Community Experience Distilled

Java Persistence with MyBatis 3

A practical guide to MyBatis, a simple yet powerful Java Persistence Framework!

K. Siva Prasad Reddy

[PACKT] PUBLISHING open source* community experience distilled

Java Persistence with MyBatis 3

A practical guide to MyBatis, a simple yet powerful Java Persistence Framework!

K. Siva Prasad Reddy

BIRMINGHAM - MUMBAI

Java Persistence with MyBatis 3

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2013

Production Reference: 1130613

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-78216-680-1

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author
K. Siva Prasad Reddy

Reviewers
Muhammad Edwin
Eduardo Macarrón

Acquisition Editor
Usha Iyer

Commissioning Editor
Ameya Sawant

Technical Editors
Jeeten Handu
Akshata Patil
Zafeer Rais

Copy Editor
Alfida Paiva
Insiya Morbiwala
Laxmi Subramanian

Project Coordinator
Suraj Bist

Proofreader
Lesley Harrison

Indexer
Monica Ajmera Mehta

Graphics
Abhinash Sahu

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

K. Siva Prasad Reddy is a Senior Software Engineer living in Hyderabad, India and has more than six years' experience in developing enterprise applications with Java and JavaEE technologies. Siva is a Sun Certified Java Programmer and has a lot of experience in server-side technologies such as Java, JavaEE, Spring, Hibernate, MyBatis, JSF (PrimeFaces), and WebServices (SOAP/REST).

Siva normally shares the knowledge he has acquired on his blog `www.sivalabs.in`. If you want to find out more information about his work, you can follow him on Twitter (`@sivalabs`) and GitHub (`https://github.com/sivaprasadreddy`).

I would like to thank my wife Neha, as she supported me in every step of the process and without her, this wouldn't have been possible. I thank my parents and my sister for their moral support in helping me complete this dream.

About the Reviewers

Muhammad Edwin is the founder and Chief Technology Officer for Baculsoft Technology, an Indonesian leading system integrator company, which provides consultancy, support, and services around open source technologies. His primary responsibility is designing and implementing solutions that use cutting-edge enterprise Java technologies to fit his customer's needs. He has held a number of positions including Software Engineer, Development Team Lead, and also as a Java Trainer. Edwin earned his Bachelor's and Master's degree from Budi Luhur University, majoring in Information Technology.

While not working or answering questions on various forums and mailing lists, he can be found traveling around beautiful beaches, scuba diving, and clicking underwater pictures.

I would like to thank my parents and my wife, Nunung Astuti, for their unwavering support while I used my personal time to review this book. I would also like to thank my colleagues at Budi Luhur University, my friends at Kaskus Programmer Community, and also people from Java User Group Indonesia. May the Source be with you.

Eduardo Macarrón has worked as an enterprise integrator and solution architect for 15 years in the electric utility industry, which focused on large projects (with more than 100 developers).

He is an open source enthusiast and has been a member of the MyBatis project since 2010.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

For many software systems, saving and retrieving data from a database is a crucial part of the process. In Java land there are many tools and frameworks for implementing the data persistence layer and each of them follow a different approach. MyBatis, a simple yet powerful Java persistence framework, took the approach of eliminating the boilerplate code and leveraging the power of SQL and Java while still providing powerful features.

This MyBatis book will take you through the process of installing, configuring, and using MyBatis. Concepts in every chapter are explained through simple and practical examples with step-by-step instructions.

By the end of the book, you will not only gain theoretical knowledge but also gain hands-on practical understanding and experience on how to use MyBatis in your real projects.

This book can also be used as a reference or to relearn the concepts that have been discussed in each chapter. It has illustrative examples, wherever necessary, to make sure it is easy to follow.

What this book covers

Chapter 1, *Getting Started with MyBatis*, introduces MyBatis persistence framework and explains the advantages of using MyBatis instead of plain JDBC. We will also look at how to create a project, install MyBatis framework dependencies with and without the Maven build tool, configure, and use MyBatis.

Chapter 2, *Bootstrapping MyBatis*, covers how to bootstrap MyBatis using XML and Java API-based configuration. We will also learn various MyBatis configuration options such as type aliases, type handlers, global settings, and so on.

Chapter 3, *SQL Mappers Using XML*, goes in-depth into writing SQL mapped statements using the Mapper XML files. We will learn how to configure simple statements, statements with one-to-one, one-to-many relationships and mapping results using ResultMaps. We will also learn how to build dynamic queries, paginated results, and custom ResultSet handling.

Chapter 4, *SQL Mappers Using Annotations*, covers writing SQL mapped statements using annotations. We will learn how to configure simple statements, statements with one-to-one and one-to-many relationships. We will also look into building dynamic queries using SqlProvider annotations.

Chapter 5, *Integration with Spring*, covers how to integrate MyBatis with Spring framework. We will learn how to install Spring libraries, register MyBatis beans in Spring ApplicationContext, inject SqlSession and Mapper beans, and use Spring's annotation-based transaction handling mechanism with MyBatis.

What you need for this book

You will need the following software to follow the examples:

- Java JDK 1.5+
- MyBatis latest version (`https://code.google.com/p/mybatis/`)
- MySQL (`http://www.mysql.com/`) or any other relational database, which has JDBC driver
- Eclipse (`http://www.eclipse.org`) or any of your favorite Java IDE
- Apache Maven build tool (`http://maven.apache.org/`)

Who this book is for

This book is for Java developers who have at least some basic experience with databases and using JDBC. You will need to have a basic familiarity with SQL. We do not assume that you have prior experience with MyBatis.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
package com.mybatis3.domain;
import java.util.Date;
public class Student
{
  private Integer studId;
  private String name;
  private String email;
  private Date dob;
  // setters and getters
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
package com.mybatis3.domain;
import java.util.Date;
public class Student
{
  private Integer studId;
  private String name;
  private String email;
  private Date dob;
  // setters and getters
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with MyBatis

In this chapter, we will cover the following topics:

- What is MyBatis?
- Why MyBatis?
- Installing and configuring MyBatis
- Sample domain model

What is MyBatis?

MyBatis is an open source persistence framework that simplifies the implementation of the persistence layer by abstracting a lot of JDBC boilerplate code and provides a simple and easy-to-use API to interact with the database.

MyBatis was formerly known as iBATIS and was started by Clinton Begin in 2002. MyBatis 3 is a complete redesign of iBATIS, with annotations and Mapper support.

The main reason for the popularity of MyBatis is its simplicity and ease of use. In Java applications, the persistence layer involves populating Java objects with data loaded from the database using SQL queries, and persisting the data in Java objects into the database using SQL.

MyBatis makes using SQL easy by abstracting low-level JDBC code, automating the process of populating the SQL result set into Java objects, and persisting data into tables by extracting the data from Java objects.

If you are currently using iBATIS and want to migrate to MyBatis, you can find the step-by-step instructions on the official MyBatis website at `https://code.google.com/p/mybatis/wiki/DocUpgrade3`.

Why MyBatis?

There are many Java-based persistence frameworks, however MyBatis became popular because of the following reasons:

- It Eliminates a lot of JDBC boilerplate code
- It has a low learning curve
- It works well with legacy databases
- It embraces SQL
- It provides support for integration with Spring and Guice frameworks
- It provides support for integration with third-party cache libraries
- It induces better performance

Eliminates a lot of JDBC boilerplate code

Java has a **Java DataBase Connectivity** (**JDBC**) API to work with relational databases. But JDBC is a very low-level API, and we need to write a lot of code to perform database operations.

Let us examine how we can implement simple `insert` and `select` operations on a `STUDENTS` table using plain JDBC.

Assume that the `STUDENTS` table has `STUD_ID`, `NAME`, `EMAIL`, and `DOB` columns.

The corresponding `Student` JavaBean is as follows:

```
package com.mybatis3.domain;
import java.util.Date;
public class Student
{
  private Integer studId;
  private String name;
  private String email;
  private Date dob;
  // setters and getters
}
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The following `StudentService.java` program implements the `SELECT` and `INSERT` operations on the `STUDENTS` table using JDBC.

```
public Student findStudentById(int studId)
{
  Student student = null;
  Connection conn = null;
  try{
//obtain connection
    conn = getDatabaseConnection();
    String sql = "SELECT * FROM STUDENTS WHERE STUD_ID=?";
//create PreparedStatement
    PreparedStatement pstmt = conn.prepareStatement(sql);
//set input parameters
    pstmt.setInt(1, studId);
    ResultSet rs = pstmt.executeQuery();
//fetch results from database and populate into Java objects
    if(rs.next())  {
      student = new Student();
      student.setStudId(rs.getInt("stud_id"));
      student.setName(rs.getString("name"));
      student.setEmail(rs.getString("email"));
      student.setDob(rs.getDate("dob"));
    }
  } catch (SQLException e){
    throw new RuntimeException(e);
  }finally{
//close connection
    if(conn!= null){
      try {
        conn.close();
      } catch (SQLException e){ }
    }
  }
  return student;
}
```

```
public void createStudent(Student student)
{
  Connection conn = null;
  try{
//obtain connection
    conn = getDatabaseConnection();
    String sql = "INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL,DOB)
    VALUES(?,?,?,?)";
//create a PreparedStatement
    PreparedStatement pstmt = conn.prepareStatement(sql);
//set input parameters
    pstmt.setInt(1, student.getStudId());
    pstmt.setString(2, student.getName());
    pstmt.setString(3, student.getEmail());
    pstmt.setDate(4, new
    java.sql.Date(student.getDob().getTime()));
    pstmt.executeUpdate();

  } catch (SQLException e){
    throw new RuntimeException(e);
  }finally{
//close connection
    if(conn!= null){
      try {
        conn.close();
      } catch (SQLException e){ }
    }
  }
}

protected Connection getDatabaseConnection() throws SQLException
{
  try{
    Class.forName("com.mysql.jdbc.Driver");
    return DriverManager.getConnection
    ("jdbc:mysql://localhost:3306/test", "root", "admin");
  } catch (SQLException e){
    throw e;
  } catch (Exception e){
    throw new RuntimeException(e);
  }
}
```

There is a lot of duplicate code in each of the preceding methods, for creating a connection, creating a statement, setting input parameters, and closing the resources, such as the connection, statement, and result set.

MyBatis abstracts all these common tasks so that the developer can focus on the really important aspects, such as preparing the SQL statement that needs to be executed and passing the input data as Java objects.

In addition to this, MyBatis automates the process of setting the query parameters from the input Java object properties and populates the Java objects with the SQL query results as well.

Now let us see how we can implement the preceding methods using MyBatis:

1. Configure the queries in a SQL Mapper config file, say `StudentMapper.xml`.

```
<select id="findStudentById" parameterType="int"
resultType=" Student">
  SELECT STUD_ID AS studId, NAME, EMAIL, DOB
  FROM STUDENTS WHERE STUD_ID=#{Id}
</select>

<insert id="insertStudent" parameterType="Student">
  INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL,DOB)
  VALUES(#{studId},#{name},#{email},#{dob})
</insert>
```

2. Create a `StudentMapper` interface.

```
public interface StudentMapper
{
  Student findStudentById(Integer id);
  void insertStudent(Student student);
}
```

3. In Java code, you can invoke these statements as follows:

```
SqlSession session = getSqlSessionFactory().openSession();
StudentMapper mapper =
session.getMapper(StudentMapper.class);
// Select Student by Id
Student student = mapper.selectStudentById(1);
//To insert a Student record
mapper.insertStudent(student);
```

That's it! You don't need to create the `Connection`, `PrepareStatement`, `extract`, and `set` parameters and close the connection by yourself for every database operation. Just configure the database connection properties and SQL statements, and MyBatis will take care of all the ground work.

Don't worry about what `SqlSessionFactory`, `SqlSession`, and Mapper XML files are. These concepts will be explained in detail in the coming chapters.

Along with these, MyBatis provides many other features that simplify the implementation of persistence logic.

- It supports the mapping of complex SQL result set data to nested object graph structures
- It supports the mapping of one-to-one and one-to-many results to Java objects
- It supports building dynamic SQL queries based on the input data

Low learning curve

One of the primary reasons for MyBatis' popularity is that it is very simple to learn and use because it depends on your knowledge of Java and SQL. If developers are familiar with Java and SQL, they will find it fairly easy to get started with MyBatis.

Works well with legacy databases

Sometimes we may need to work with legacy databases that are not in a normalized form. It is possible, but difficult, to work with these kinds of legacy databases with fully-fledged ORM frameworks such as Hibernate because they attempt to statically map Java objects to database tables.

MyBatis works by mapping query results to Java objects; this makes it easy for MyBatis to work with legacy databases. You can create Java domain objects following the object-oriented model, execute queries against the legacy database, and map the query results to the Java objects.

Embraces SQL

Full-fledged ORM frameworks such as Hibernate encourage working with entity objects and generate SQL queries under the hood. Because of this SQL generation, we may not be able to take advantage of database-specific features. Hibernate allows to execute native SQLs, but that might defeat the promise of a database-independent persistence.

The MyBatis framework embraces SQL instead of hiding it from developers. As MyBatis won't generate any SQLs and developers are responsible for preparing the queries, you can take advantage of database-specific features and prepare optimized SQL queries. Also, working with stored procedures is supported by MyBatis.

Supports integration with Spring and Guice frameworks

MyBatis provides out-of-the-box integration support for the popular dependency injection frameworks Spring and Guice; this further simplifies working with MyBatis.

Supports integration with third-party cache libraries

MyBatis has inbuilt support for caching `SELECT` query results within the scope of `SqlSession` level ResultSets. In addition to this, MyBatis also provides integration support for various third-party cache libraries, such as EHCache, OSCache, and Hazelcast.

Better performance

Performance is one of the key factors for the success of any software application. There are lots of things to consider for better performance, but for many applications, the persistence layer is a key for overall system performance.

- MyBatis supports database connection pooling that eliminates the cost of creating a database connection on demand for every request.
- MyBatis has an in-built cache mechanism which caches the results of SQL queries at the SqlSession level. That is, if you invoke the same mapped select query, then MyBatis returns the cached result instead of querying the database again.
- MyBatis doesn't use proxying heavily and hence yields better performance compared to other ORM frameworks that use proxies extensively.

There are no one-size-fits-all solutions in software development. Each application has a different set of requirements, and we should choose our tools and frameworks based on application needs. In the previous section, we have seen various advantages of using MyBatis. But there will be cases where MyBatis may not be the ideal or best solution.

If your application is driven by an object model and wants to generate SQL dynamically, MyBatis may not be a good fit for you. Also, if you want to have a transitive persistence mechanism (saving the parent object should persist associated child objects as well) for your application, Hibernate will be better suited for it.

Installing and configuring MyBatis

We are assuming that the JDK 1.6+ and MySQL 5 database servers have been installed on your system. The installation process of JDK and MySQL is outside the scope of this book.

At the time of writing this book, the latest version of MyBatis is MyBatis 3.2.2. Throughout this book, we will use the MyBatis 3.2.2 version.

Even though it is not mandatory to use IDEs, such as Eclipse, NetBeans IDE, or IntelliJ IDEA for coding, they greatly simplify development with features such as handy autocompletion, refactoring, and debugging. You can use any of your favorite IDEs for this purpose.

This section explains how to develop a simple Java project using MyBatis:

- By creating a `STUDENTS` table and inserting sample data
- By creating a Java project and adding `mybatis-3.2.2.jar` to the classpath
- By creating the `mybatis-config.xml` and `StudentMapper.xml` configuration files
- By creating the `MyBatisSqlSessionFactory` singleton class
- By creating the `StudentMapper` interface and the `StudentService` classes
- By creating a JUnit test for testing `StudentService`

Creating a STUDENTS table and inserting sample data

Create a `STUDENTS` table and insert sample records in the MySQL database using the following SQL script:.

```
CREATE TABLE STUDENTS
(
stud_id int(11) NOT NULL AUTO_INCREMENT,
  name varchar(50) NOT NULL,
  email varchar(50) NOT NULL,
  dob date DEFAULT NULL,
  PRIMARY KEY (stud_id)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;

/*Sample Data for the students table */
insert  into students(stud_id,name,email,dob)
values (1,'Student1','student1@gmail.com','1983-06-25');

insert  into students(stud_id,name,email,dob)
values (2,'Student2','student2@gmail.com','1983-06-25');
```

Creating a Java project and adding mybatis-3.2.2.jar to the classpath

Let us create a Java project and configure MyBatis JAR dependencies.

1. Create a Java project named `mybatis-demo`.
2. If you are not using a build tool, such as Maven or Gradle, with dependency management capabilities, you need to download the JAR files and add them to the classpath manually.
3. You can download the MyBatis distribution `mybatis-3.2.2.zip` from `http://code.google.com/p/mybatis/`. This bundle contains the `mybatis-3.2.2.jar` file and its optional dependent jars such as the `slf4j/log4j` logging jars.
4. We will use the SLF4J logging framework along with the `log4j` binding for logging. The `mybatis-3.2.2.zip` file contains the `slf4j` dependency jars as well.

5. Extract the ZIP file and add the `mybatis-3.2.2.jar`, `lib/slf4j-api-1.7.5.jar`, `lib/slf4j-log4j12-1.7.5.jar`, and `lib/log4j-1.2.17.jar` JARS to the classpath.
6. You can download the JUnit JAR file from `http://junit.org/` and the driver from `http://www.mysql.com/downloads/connector/j/`.
7. Add `junit-4.11.jar` and `mysql-connector-java-5.1.22.jar` to the classpath.
8. If you are using Maven, configuring these jar dependencies is much simpler. In your `pom.xml` file add the following dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.2.2</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.22</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
```

```
    <scope>test</scope>
  </dependency>
</dependencies>
```

9. Create the `log4j.properties` file and put it in the classpath.

```
log4j.rootLogger=DEBUG, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d [%-5p] %c
- %m%n
```

Creating the mybatis-config.xml and StudentMapper.xml configuration files

Let us create MyBatis' main configuration file `mybatis-config.xml` with database connection properties, type aliases, and so on, and create the `StudentMapper.xml` file containing mapped SQL statements.

1. Create the `mybatis-config.xml` file to configure database connection properties, SQL Mapper files, type aliases, and so on, and put it in the classpath.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
<typeAliases>
  <typeAlias alias="Student"
  type="com.mybatis3.domain.Student"/>
</typeAliases>
<environments default="development">
<environment id="development">
<transactionManager type="JDBC"/>
<dataSource type="POOLED">
<property name="driver" value="com.mysql.jdbc.Driver"/>
<property name="url"
value="jdbc:mysql://localhost:3306/test"/>
<property name="username" value="root"/>
<property name="password" value="admin"/>
</dataSource>
</environment>
</environments>
<mappers>
```

```
<mapper resource="com/mybatis3/mappers/StudentMapper.xml"/>
</mappers>
</configuration>
```

2. Create the SQL Mapper XML file `StudentMapper.xml` and put it in the classpath under the `com.mybatis3.mappers` package.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
  <mapper namespace="com.mybatis3.mappers.StudentMapper">
  <resultMap type="Student" id="StudentResult">
    <id property="studId" column="stud_id"/>
    <result property="name" column="name"/>
    <result property="email" column="email"/>
    <result property="dob" column="dob"/>
  </resultMap>

  <select id="findAllStudents" resultMap="StudentResult">
    SELECT * FROM STUDENTS
</select>

<select id="findStudentById" parameterType="int"
resultType="Student">
  SELECT STUD_ID AS STUDID, NAME, EMAIL, DOB
  FROM STUDENTS WHERE STUD_ID=#{Id}
</select>

<insert id="insertStudent" parameterType="Student">
  INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL,DOB)
  VALUES(#{studId },#{name},#{email},#{dob})
</insert>
</mapper>
```

The preceding `StudentMapper.xml` file contains the mapped statements that can be invoked using the statement ID along with the namespace.

Creating the MyBatisSqlSessionFactory singleton class

Create the `MyBatisSqlSessionFactory.java` class to instantiate and hold the `SqlSessionFactory` singleton object.

```
package com.mybatis3.util;
import java.io.*;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.*;
public class MyBatisSqlSessionFactory
{
  private static SqlSessionFactory sqlSessionFactory;

  public static SqlSessionFactory getSqlSessionFactory() {
    if(sqlSessionFactory==null) {
      InputStream inputStream;
      try {
        inputStream = Resources.
        getResourceAsStream("mybatis-config.xml");
        sqlSessionFactory = new
        SqlSessionFactoryBuilder().build(inputStream);
      } catch (IOException e) {
        throw new RuntimeException(e.getCause());
      }
    }
    return sqlSessionFactory;
  }

  public static SqlSession openSession() {
    return getSqlSessionFactory().openSession();
  }

}
```

In the preceding code snippet, we have created the `SqlSessionFactory` object that will be used to get `SqlSession` and execute mapped statements.

Creating the StudentMapper interface and the StudentService classes

Let us create a `StudentMapper` interface with method signatures similar to mapped SQL statements and a `StudentService.java` class that contains the implementation of business operations.

1. First, create the JavaBean `Student.java`.

```
package com.mybatis3.domain;
import java.util.Date;
public class Student
{
  private Integer studId;
  private String name;
  private String email;
  private Date dob;
  // setters and getters
}
```

2. Create a Mapper interface `StudentMapper.java` with the same method signatures as the mapped statements in `StudentMapper.xml`.

```
package com.mybatis3.mappers;
import java.util.List;
import com.mybatis3.domain.Student;
public interface StudentMapper
{
  List<Student> findAllStudents();
  Student findStudentById(Integer id);
  void insertStudent(Student student);
}
```

3. Now create `StudentService.java` to implement database operations on the `STUDENTS` table.

```
package com.mybatis3.services;
import java.util.List;
import org.apache.ibatis.session.SqlSession;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.mybatis3.domain.Student;
import com.mybatis3.mappers.StudentMapper;
import com.mybatis3.util.MyBatisSqlSessionFactory;
public class StudentService
{
```

```
  private Logger logger =
  LoggerFactory.getLogger(getClass());

  public List<Student> findAllStudents()
  {
    SqlSession sqlSession =
    MyBatisSqlSessionFactory.openSession();
      try {
      StudentMapper studentMapper =
      sqlSession.getMapper(StudentMapper.class);
      return studentMapper.findAllStudents();
    } finally {
//If sqlSession is not closed
//then database Connection associated this sqlSession will not be
returned to pool
//and application may run out of connections.
      sqlSession.close();
    }
  }

  public Student findStudentById(Integer studId)
  {
    logger.debug("Select Student By ID :{}", studId);
    SqlSession sqlSession =
    MyBatisSqlSessionFactory.openSession();
    try {
      StudentMapper studentMapper =
      sqlSession.getMapper(StudentMapper.class);
      return studentMapper.findStudentById(studId);
    } finally {
      sqlSession.close();
    }
  }

  public void createStudent(Student student)
  {
    SqlSession sqlSession =
    MyBatisSqlSessionFactory.openSession();
    try {
      StudentMapper studentMapper =
      sqlSession.getMapper(StudentMapper.class);
      studentMapper.insertStudent(student);
      sqlSession.commit();
    } finally {
      sqlSession.close();
```

```
        }
      }
    }
```

You can also execute mapped SQL statements without using Mapper interfaces. An example is as follows:

```
Student student = (Student)sqlSession.
selectOne("com.mybatis3.mappers.StudentMapper.findStudentById",
studId);
```

However, it is best practice to use Mapper interfaces so that we invoke mapped statements in a type-safe manner.

Creating a JUnit test for testing StudentService

Create a JUnit test class `StudentServiceTest.java` to test the `StudentService` methods.

```
package com.mybatis3.services;
import java.util.*;
import org.junit.*;
import com.mybatis3.domain.Student;
public class StudentServiceTest
{
  private static StudentService studentService;

  @BeforeClass
  public static void setup(){
    studentService = new StudentService();
  }
  @AfterClass
  public static void teardown(){
    studentService = null;
  }

  @Test
  public void testFindAllStudents() {
    List<Student> students = studentService.findAllStudents();
    Assert.assertNotNull(students);
    for (Student student : students) {
      System.out.println(student);
    }
```

```
    }

    @Test
    public void testFindStudentById() {
      Student student = studentService.findStudentById(1);
      Assert.assertNotNull(student);
      System.out.println(student);
    }

    @Test
    public void testCreateStudent() {
      Student student = new Student();
      int id = 3;
      student.setStudId(id);
      student.setName("student_"+id);
      student.setEmail("student_"+id+"gmail.com");
      student.setDob(new Date());
      studentService.createStudent(student);
      Student newStudent = studentService.findStudentById(id);
      Assert.assertNotNull(newStudent);
    }
  }
```

How it works

First, we have configured the main MyBatis configuration file, `mybatis-config.xml`, with the JDBC connection parameters and configured the Mapper XML files that contain the SQL statement's mappings.

We have created the `SqlSessionFactory` object using the `mybatis-config.xml` file. There should be only one instance of `SqlSessionFactory` per database environment, so we have used a singleton pattern to have only one instance of `SqlSessionFactory`.

We have created a Mapper interface, `StudentMapper`, with method signatures that are the same as those of the mapped statements in `StudentMapper.xml`. Note that the `StudentMapper.xml` namespace value is set to `com.mybatis3.mappers.StudentMapper`, which is a fully qualified name of the `StudentMapper` interface. This enables us to invoke mapped statements using the Mapper interface.

In `StudentService.java`, we have created a new `SqlSession` in each method and closed it after the method completes. Each thread should have its own instance of `SqlSession`. Instances of `SqlSession` objects are not thread safe and should not be shared. So the best scope for `SqlSession` is the method scope. From a web application perspective, `SqlSession` should have a request scope.

Sample domain model

In this section, we will discuss the sample domain model that represents an e-learning application that will be used throughout the book.

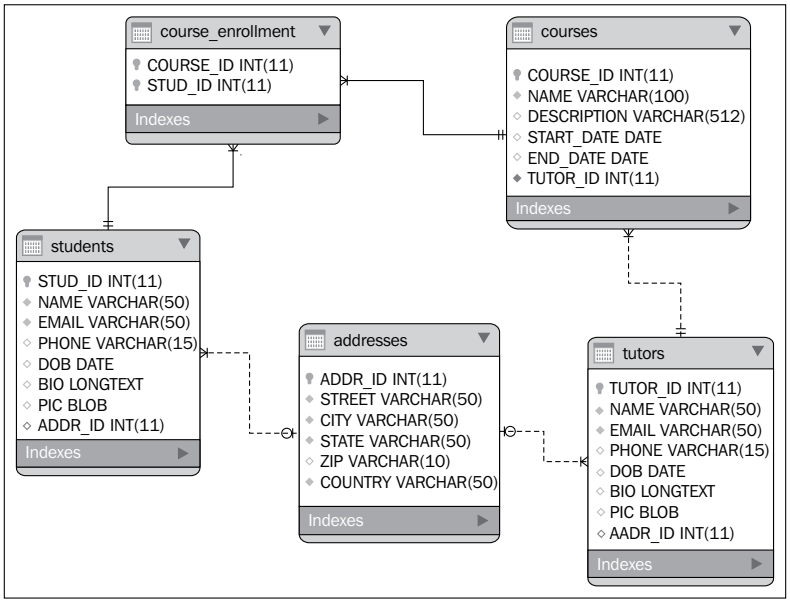
An e-learning system enables students to enroll for courses and take lessons through web-based mediums, such as virtual classes or desktop-sharing systems.

The tutors who are interested in teaching courses through an e-learning system can register with the system and announce the course details that they are going to teach.

The course details include course name, description, and duration. Students from across the globe can register and enroll for the courses that they want to learn.

The e-learning system provides a course search functionality where you can search for the available courses by course name, tutor, start date, or end date.

The following diagram represents the **database schema** for our e-learning application:



Summary

In this chapter, we discussed about MyBatis and the advantages of using MyBatis instead of plain JDBC for database access. We learned how to create a project, install MyBatis jar dependencies, create a MyBatis configuration file, and configure SQL mapped statements in Mapper XML files. We created a `Service` class to insert and get data from the database using MyBatis. We created a JUnit test case for testing `Service`.

In the next chapter, we will discuss bootstrapping MyBatis using XML and Java-API-based approaches in detail.

2
Bootstrapping MyBatis

The key component of MyBatis is `SqlSessionFactory` from which we get `SqlSession` and execute the mapped SQL statements. The `SqlSessionFactory` object can be created using XML-based configuration or Java API.

We will explore various MyBatis configuration elements, such as `dataSource`, environments, global settings, `typeAliases`, `typeHandlers`, and SQL mappers, and instantiate `SqlSessionFactory`.

In this chapter, we will cover:

- Configuring MyBatis using XML
- Configuring MyBatis using Java API
- Customizing MyBatis logging

Configuring MyBatis using XML

The most commonly used approach for building `SqlSessionFactory` is XML-based configuration. The following `mybatis-config.xml` file shows how a typical MyBatis configuration file looks:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

  <properties resource="application.properties">
    <property name="username" value="db_user"/>
    <property name="password" value="verysecurepwd"/>
  </properties>
  <settings>
```

```
    <setting name="cacheEnabled" value="true"/>
  </settings>
    <typeAliases>
      <typeAlias alias="Tutor" type="com.mybatis3.domain.Tutor"/>
      <package name="com.mybatis3.domain"/>
    </typeAliases>

    <typeHandlers>
      <typeHandler handler="com.mybatis3.typehandlers.
  PhoneTypeHandler"/>
      <package name="com.mybatis3.typehandlers"/>
    </typeHandlers>

    <environments default="development">
    <environment id="development">
    <transactionManager type="JDBC"/>
    <dataSource type="POOLED">
    <property name="driver" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
    </dataSource>
    </environment>

    <environment id="production">
    <transactionManager type="MANAGED"/>
    <dataSource type="JNDI">
    <property name="data_source" value="java:comp/jdbc/MyBatisDemoDS"/>
    </dataSource>
    </environment>
    </environments>

    <mappers>
    <mapper resource="com/mybatis3/mappers/StudentMapper.xml"/>
    <mapper url="file:///D:/mybatisdemo/mappers/TutorMapper.xml"/>
    <mapper class="com.mybatis3.mappers.TutorMapper"/>
    </mappers>

  </configuration>
```

Let us discuss each part of the preceding configuration file, starting with the most important part, that is, **environments**.

Environment

MyBatis supports configuring multiple dataSource environments so that deploying the application in various environments, such as DEV, TEST, QA, UAT, and PRODUCTION, can be easily achieved by changing the default environment value to the desired `environment id` value. In the preceding configuration, the default environment has been set to `development`. When deploying the application on to production servers, you don't need to change the configuration much; just set the default environment to the production `environment id` attribute.

Sometimes, we may need to work with multiple databases within the same application. For example, we may have the `SHOPPINGCART` database to store all the order details and the `REPORTS` database to store the aggregates of the order details for reporting purposes.

If your application needs to connect to multiple databases, you'll need to configure each database as a separate environment and create a separate `SqlSessionFactory` object for each database.

```
<environments default="shoppingcart">
  <environment id="shoppingcart">
  <transactionManager type="MANAGED"/>
  <dataSource type="JNDI">
    <property name="data_source" value="java:comp/jdbc/
ShoppingcartDS"/>
  </dataSource>
  </environment>

  <environment id="reports">
  <transactionManager type="MANAGED"/>
  <dataSource type="JNDI">
    <property name="data_source" value="java:comp/jdbc/ReportsDS"/>
  </dataSource>
  </environment>
</environments>
```

We can create `SqlSessionFactory` for a given environment as follows:

```
inputStream = Resources.getResourceAsStream("mybatis-config.xml");
defaultSqlSessionFactory = new SqlSessionFactoryBuilder().
build(inputStream);
cartSqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStre
am,"shoppingcart");
reportSqlSessionFactory = new SqlSessionFactoryBuilder().
build(inputStream,"reports");
```

When we create `SqlSessionFactory` without explicitly defining `environment id`, `SqlSessionFactory` will be created using the default environment. In the preceding code, `defaultSqlSessionFactory` was created using the `shoppingcart` environment settings.

For each environment, we need to configure the `dataSource` and `transactionManager` elements.

DataSource

The `dataSource` element is used to configure the database connection properties.

```
<dataSource type="POOLED">
  <property name="driver" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</dataSource>
```

The `dataSource` type can be one of the built-in types such as `UNPOOLED`, `POOLED`, or `JNDI`.

- If you set the type to `UNPOOLED`, MyBatis will open a new connection and close that connection for every database operation. This method can be used for simple applications that have a small number of concurrent users.
- If you set the type to `POOLED`, MyBatis will create a pool of database connections, and one of these connections will be used for the database operation. Once this is complete, MyBatis will return the connection to the pool. This is a commonly used method for developing/testing environments.
- If you set the type to `JNDI`, MyBatis will get the connection from the `JNDI` dataSource that has typically been configured in the application server. This is a preferred method in production environments.

TransactionManager

MyBatis supports two types of transaction managers: `JDBC` and `MANAGED`.

- The `JDBC` transaction manager is used where the application is responsible for managing the connection life cycle, that is, `commit`, `rollback`, and so on. When you set the `TransactionManager` property to `JDBC`, behind the scenes MyBatis uses the `JdbcTransactionFactory` class to create `TransactionManager`. For example, an application deployed on Apache Tomcat should manage the transactions by itself.

- The `MANAGED` transaction manager is used where the application server is responsible for managing the connection life cycle. When you set the `TransactionManager` property to `MANAGED`, behind the scenes MyBatis uses the `ManagedTransactionFactory` class to create `TransactionManager`. For example, a JavaEE application deployed on an application server, such as JBoss, WebLogic, or GlassFish, can leverage the application server's transaction management capabilities using EJB. In these managed environments, you can use the `MANAGED` transaction manager.

Properties

The properties configuration element can be used to externalize the configuration values into a properties file and use the properties' key names as placeholders. In the preceding configuration, we have externalized the database connection properties into the `application.properties` file and used placeholders for the driver, URL, and so on.

1. Configure the database connection parameters in `application.properties` as follows:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatisdemo
jdbc.username=root
jdbc.password=admin
```

2. In `mybatis-config.xml`, use the placeholders for the properties defined in `application.properties`.

```
<properties resource="application.properties">
  <property name="jdbc.username" value="db_user"/>
  <property name="jdbc.password" value="verysecurepwd"/>
</properties>

<dataSource type="POOLED">
  <property name="driver" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</dataSource>
```

Also, you can configure the default parameter values inside the `<properties>` element; they will be overridden by the values in the properties file if there are properties with the same key name.

```
<properties resource="application.properties">
  <property name="jdbc.username" value="db_user"/>
  <property name="jdbc.password" value="verysecurepwd"/>
</properties>
```

Here, the username and password values `db_user` and `verysecurepwd` will be overridden by the values in `application.properties` if the `application.peroperties` file contains the key names `jdbc.username` and `jdbc.password`.

typeAliases

In the SQL Mapper configuration file, we need to give the fully qualified name of the JavaBeans for the `resultType` and `parameterType` attributes.

An example is as follows:

```
<select id="findStudentById" parameterType="int"
    resultType="com.mybatis3.domain.Student">
    SELECT STUD_ID AS ID, NAME, EMAIL, DOB
  FROM STUDENTS WHERE STUD_ID=#{Id}
</select>

<update id="updateStudent" parameterType="com.mybatis3.domain.
Student">
  UPDATE STUDENTS SET NAME=#{name}, EMAIL=#{email}, DOB=#{dob}
  WHERE STUD_ID=#{id}
</update>
```

Here we are giving the fully qualified name of the `Student` type `com.mybatis3.domain.Student` for the `resultType` and `parameterType` attributes.

Instead of typing the fully qualified names everywhere, we can give the alias names and use these alias names in all the other places where we need to give the fully qualified names.

An example is as follows:

```
<typeAliases>
  <typeAlias alias="Student" type="com.mybatis3.domain.Student"/>
  <typeAlias alias="Tutor" type="com.mybatis3.domain.Tutor"/>
  <package name="com.mybatis3.domain"/>
</typeAliases>
```

Now in the SQL Mapper file, we can use the alias name `Student` as follows:

```
<select id="findStudentById" parameterType="int" resultType="Student">
    SELECT STUD_ID AS ID, NAME, EMAIL, DOB
  FROM STUDENTS WHERE STUD_ID=#{id}
</select>

<update id="updateStudent" parameterType="Student">
  UPDATE STUDENTS SET NAME=#{name}, EMAIL=#{email}, DOB=#{dob}
  WHERE STUD_ID=#{id}
</update>
```

Instead of giving an alias name for each JavaBeans separately, you can give the package name where MyBatis can scan and register aliases using uncapitalized, nonqualified class names of the Bean.

An example is as follows:

```
<typeAliases>
<package name="com.mybatis3.domain"/>
</typeAliases>
```

If there are `Student.java` and `Tutor.java` Beans in the `com.mybatis3.domain` package, `com.mybatis3.domain.Student` will be registered as `student` and `com.mybatis3.domain.Tutor` will be registered as `tutor`.

An example is as follows:

```
<typeAliases>
  <typeAlias alias="Student" type="com.mybatis3.domain.Student"/>
  <typeAlias alias="Tutor" type="com.mybatis3.domain.Tutor"/>
  <package name="com.mybatis3.domain"/>
  <package name="com.mybatis3.webservices.domain"/>
</typeAliases>
```

There is another way of aliasing JavaBeans, using the `@Alias` annotation.

```
@Alias("StudentAlias")
public class Student
{
}
```

The `@Alias` annotation overrides the `<typeAliases>` configuration.

typeHandlers

As discussed in the previous chapter, MyBatis simplifies the persistent logic implementation by abstracting JDBC. MyBatis uses JDBC under the hood and provides simpler ways to implement database operations.

When MyBatis executes an `INSERT` statement by taking a Java object as an input parameter, it will create `PreparedStatement` and set the parameter values for the placeholders using the `setXXX()` methods.

Here `XXX` can be any one of `Int`, `String`, `Date`, and so on, based on the type of Java property.

An example is as follows:

```
<insert id="insertStudent" parameterType="Student">
  INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL,DOB)
VALUES(#{studId},#{name},#{email},#{dob})
</insert>
```

To execute this statement, MyBatis will perform the following sequence of actions.

1. Create a `PreparedStatement` interface with placeholders as follows:

   ```
   PreparedStatement pstmt = connection.prepareStatement
   ("INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL,DOB) VALUES(?,?,?,?)");
   ```

2. Check the property type of `studId` in the `Student` object and use the appropriate `setXXX` method to set the value. Here `studId` is of the type integer, so it will use the `setInt()` method.

   ```
   pstmt.setInt(1,student.getStudId());
   ```

3. Similarly, for the `name` and `email` attributes MyBatis will use the `setString()` methods because they are of the type `String`.

   ```
   pstmt.setString(2, student.getName());
   pstmt.setString(3, student.getEmail());
   ```

4. And for the `dob` property, MyBatis will use the `setDate()` method for setting the `dob` place holder value.
5. MyBatis first converts `java.util.Date` into `java.sql.Timestamp` and sets the value.

   ```
   pstmt.setTimestamp(4, new Timestamp((student.getDob()).
   getTime()));
   ```

Cool. But how does MyBatis know to use `setInt()` for the Integer and `setString` for the String type properties? MyBatis determines all these things using type handlers.

MyBatis comes with built-in type handlers for all primitive types, primitive wrapper types, `byte[]`, `java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `java enums`, and so on. So when MyBatis finds one of these types of properties, it uses the corresponding type handler to set the value on `PreparedStatement`, while at the same time populating the JavaBeans from the SQL Result Set.

What if we give a custom object type value to store into the database?

An example is as follows:

Assume that the `STUDENTS` table has a `PHONE` column that is of the type `VARCHAR(15)`. The JavaBeans `Student` has the `phoneNumber` property of the `PhoneNumber` class.

```
public class PhoneNumber
{
  private String countryCode;
  private String stateCode;
  private String number;

  public PhoneNumber() {
  }

  public PhoneNumber(String countryCode, String stateCode, String
number) {
    this.countryCode = countryCode;
    this.stateCode = stateCode;
    this.number = number;
  }

  public PhoneNumber(String string) {
    if(string != null){
      String[] parts = string.split("-");
      if(parts.length>0) this.countryCode=parts[0];
      if(parts.length>1) this.stateCode=parts[1];
      if(parts.length>2) this.number=parts[2];
    }
  }

  public String getAsString() {
    return countryCode+"-"+stateCode+"-"+number;
```

```
  }
  // Setters and getters
}

public class Student
{
  private Integer id;
  private String name;
  private String email;
  private PhoneNumber phone;
  // Setters and getters
}

<insert id="insertStudent" parameterType="Student">
  insert into students(name,email,phone)
  values(#{name},#{email},#{phone})
</insert>
```

Here, for the `phone` parameter we have given the value `#{phone}`; this gives the `phone` object that is of the type `PhoneNumber`. However, MyBatis doesn't know how to handle this type of object.

To let MyBatis understand how to handle custom Java object types, such as `PhoneNumber`, we can create a custom type handler as follows:

1. MyBatis provides an abstract class `BaseTypeHandler<T>` that we can extend to create custom type handlers.

```
packagecom.mybatis3.typehandlers;

importjava.sql.CallableStatement;
importjava.sql.PreparedStatement;
importjava.sql.ResultSet;
importjava.sql.SQLException;
importorg.apache.ibatis.type.BaseTypeHandler;
importorg.apache.ibatis.type.JdbcType;
importcom.mybatis3.domain.PhoneNumber;

public class PhoneTypeHandler extends BaseTypeHandler<PhoneNumber>
{

  @Override
  public void setNonNullParameter(PreparedStatement ps, int i,
      PhoneNumber parameter, JdbcType jdbcType) throws
SQLException {
    ps.setString(i, parameter.getAsString());
```

```
  }

  @Override
  public PhoneNumber getNullableResult(ResultSet rs, String
columnName)
      throws SQLException {
    return new PhoneNumber(rs.getString(columnName));
  }

  @Override
  public PhoneNumber getNullableResult(ResultSet rs, int
columnIndex)
      throws SQLException {
    return new PhoneNumber(rs.getString(columnIndex));
  }

  @Override
  public PhoneNumber getNullableResult(CallableStatement cs, int
columnIndex)
      throws SQLException {
    return new PhoneNumber(cs.getString(columnIndex));
  }
}
```

2. We are using the `ps.setString()` and `rs.getString()` methods because the phone number is being stored in a `VARCHAR` type column.
3. Once the custom type handler is implemented, we need to register it in `mybatis-config.xml`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <properties resource="application.properties"/>

  <typeHandlers>
  <typeHandler handler="com.mybatis3.typehandlers.
PhoneTypeHandler"/>
  </typeHandlers>

</configuration>
```

After registering `PhoneTypeHandler`, MyBatis will be able to store the `Phone` type object value into any `VARCHAR` type column.

Settings

The default MyBatis global settings, which can be overridden to better suit application-specific needs, are as follows:

```
<settings>
<setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
  <setting name="useColumnLabel" value="true"/>
  <setting name="useGeneratedKeys" value="false"/>
  <setting name="autoMappingBehavior" value="PARTIAL"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25000"/>
  <setting name="safeRowBoundsEnabled" value="false"/>
  <setting name="mapUnderscoreToCamelCase" value="false"/>
  <setting name="localCacheScope" value="SESSION"/>
  <setting name="jdbcTypeForNull" value="OTHER"/>
  <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode
,toString"/>
</settings>
```

Mappers

Mapper XML files contain the mapped SQL statements that will be executed by the application using `statement id`. We need to configure the locations of the SQL Mapper files in `mybatis-config.xml`.

```
<mappers>
  <mapper resource="com/mybatis3/mappers/StudentMapper.xml"/>
  <mapper url="file:///D:/mybatisdemo/app/mappers/TutorMapper.xml"/>
  <mapper class="com.mybatis3.mappers.TutorMapper"/>
  <package name="com.mybatis3.mappers"/>
  </mappers>
```

Each of the `<mapper>` tag attributes facilitates to load mappers from different kinds of sources:

- The attribute `resource` can be used to point to a mapper file that is in the classpath
- The attribute `url` can be used to point to a mapper file by its fully qualified filesystem path or web URL

- The attribute `class` can be used to point to a Mapper interface
- The `package` element can be used to point to a package name where Mapper interfaces can be found

Configuring MyBatis using Java API

In the previous section, we have discussed various MyBatis configuration elements, such as environments, typeAliases, and typeHandlers, and how to configure them using XML. Even though you want to use the Java-API-based MyBatis configuration, it would be good to go through the previous section to have a better idea about these configuration elements. In this section, we will be referring to some of the classes described in the previous section.

MyBatis' `SqlSessionFactory` interface can be created programmatically using the Java API instead of using the XML-based configuration. Each configuration element used in an XML-based configuration can be created programmatically.

We can create the `SqlSessionFactory` object using the Java API as follows:

```
public static SqlSessionFactory getSqlSessionFactory()
{
  SqlSessionFactory sqlSessionFactory = null;
  try{
    DataSource dataSource = DataSourceFactory.getDataSource();
    TransactionFactory transactionFactory = new
JdbcTransactionFactory();
    Environment environment = new Environment("development",
transactionFactory, dataSource);
    Configuration configuration = new Configuration(environment);
    configuration.getTypeAliasRegistry().registerAlias("student",
Student.class);
    configuration.getTypeHandlerRegistry().register(PhoneNumber.
class,PhoneTypeHandler.class);
    configuration.addMapper(StudentMapper.class);
    sqlSessionFactory = new SqlSessionFactoryBuilder().
build(configuration);

  }catch (Exception e){
    throw new RuntimeException(e);
  }
  return sqlSessionFactory;
}
```

Environment

We need to create an `Environment` object for each database that we want to connect to using MyBatis. To work with multiple databases, we'll need to create a `SqlSessionFactory` object for each environment. To create an instance of `Environment`, we'll need the `javax.sql.DataSource` and `TransactionFactory` instances. Let us see how to create the `DataSource` and `TransactionFactory` objects.

DataSource

MyBatis supports three built-in `DataSource` types: `UNPOOLED`, `POOLED`, and `JNDI`.

- The `UNPOOLED` dataSource creates a new database connection every time for each user request and is not advisable for concurrent multiuser applications.
- The `POOLED` dataSource creates a pool of `Connection` objects, and for every user request, it will use one of the `Connection` objects readily available in the pool, thereby increasing performance. MyBatis provides `org.apache.ibatis.datasource.pooled.PooledDataSource` that implements `javax.sql.DataSource` to create a `Connection` pool.
- The `JNDI` dataSource uses the `Connection` pool configured in the application server and obtains a connection using a `JNDI` lookup.

Let us see how we can get a `DataSource` object using MyBatis' `PooledDataSource` interface:

```
public class DataSourceFactory
{
  public static DataSource getDataSource()
  {
    String driver = "com.mysql.jdbc.Driver";
    String url = "jdbc:mysql://localhost:3306/mybatisdemo";
    String username = "root";
    String password = "admin";
    PooledDataSource dataSource = new PooledDataSource(driver, url,
username, password);
    return dataSource;
  }
}
```

Generally in production environments, `DataSource` will be configured in the application server and get the `DataSource` object using `JNDI` as follows:

```
public class DataSourceFactory
{
  public static DataSource getDataSource()
```

```
    {
      String jndiName = "java:comp/env/jdbc/MyBatisDemoDS";
      try {
        InitialContext ctx = new InitialContext();
        DataSource dataSource = (DataSource) ctx.lookup(jndiName);
        return dataSource;
      }
      catch (NamingException e) {
        throw new RuntimeException(e);
      }
    }
  }
```

There are many popular third-party libraries, such as `commons-dbcp` and `c3p0`, implementing `javax.sql.DataSource`, and you can use any of these libraries to create a dataSource.

TransactionFactory

MyBatis supports the following two types of `TransactionFactory` implementations:

- `JdbcTransactionFactory`
- `ManagedTransactionFactory`

If the application is running in a non-managed environment, you should use `JdbcTransactionFactory`.

```
DataSource dataSource = DataSourceFactory.getDataSource();
TransactionFactory txnFactory = new JdbcTransactionFactory();
Environment environment = new Environment("development", txnFactory,
dataSource);
```

If the application is running in a managed environment and uses container-supported transaction management services, you should use `ManagedTransactionFactory`.

```
DataSource dataSource = DataSourceFactory.getDataSource();
TransactionFactory txnFactory = new ManagedTransactionFactory();
Environment environment = new Environment("development", txnFactory,
dataSource);
```

typeAliases

MyBatis provides several ways of registering Type Aliases with the `Configuration` object.

- To register an alias for a single class with an uncapitalized, nonqualified class name according to the default alias rule, use the following code:

  ```
  configuration.getTypeAliasRegistry().registerAlias(Student.class);
  ```

- To register a single class alias with a given alias name, use the following code:

  ```
  configuration.getTypeAliasRegistry().registerAlias("Student",
  Student.class);
  ```

- To register a single class alias name for the given fully qualified class name, use the following code:

  ```
  configuration.getTypeAliasRegistry().registerAlias("Student",
  "com.mybatis3.domain.Student");
  ```

- To register aliases for all the classes in the `com.mybatis3.domain` package, use the following code:

  ```
  configuration.getTypeAliasRegistry().registerAliases("com.
  mybatis3.domain");
  ```

- To register aliases for the classes that extend the `Identifiable` type in the `com.mybatis3.domain` package, use the following code

  ```
  configuration.getTypeAliasRegistry().registerAliases("com.
  mybatis3.domain", Identifiable.class);
  ```

typeHandlers

MyBatis provides several ways of registering type handlers with the `Configuration` object. We can register custom type handlers using the `Configuration` object as follows:

- To register a type handler for a specific Java class:

  ```
  configuration.getTypeHandlerRegistry().register(PhoneNumber.
  class,PhoneTypeHandler.class);
  ```

- To register a type handler:

  ```
  configuration.getTypeHandlerRegistry().register(PhoneTypeHandler.
  class);
  ```

- To register all the type handlers in the `com.mybatis3.typehandlers` package:

```
configuration.getTypeHandlerRegistry().register("com.mybatis3.
typehandlers");
```

Settings

MyBatis comes with a set of default global settings that suit well for most applications. However, you can tweak these settings to better suit your application needs. You can use the following methods to set the values of the global settings to the desired values.

```
configuration.setCacheEnabled(true);
configuration.setLazyLoadingEnabled(false);
configuration.setMultipleResultSetsEnabled(true);
configuration.setUseColumnLabel(true);
configuration.setUseGeneratedKeys(false);
configuration.setAutoMappingBehavior(AutoMappingBehavior.PARTIAL);
configuration.setDefaultExecutorType(ExecutorType.SIMPLE);
configuration.setDefaultStatementTimeout(25);
configuration.setSafeRowBoundsEnabled(false);
configuration.setMapUnderscoreToCamelCase(false);
configuration.setLocalCacheScope(LocalCacheScope.SESSION);
configuration.setAggressiveLazyLoading(true);
configuration.setJdbcTypeForNull(JdbcType.OTHER);
Set<String> lazyLoadTriggerMethods = new HashSet<String>();
lazyLoadTriggerMethods.add("equals");
lazyLoadTriggerMethods.add("clone");
lazyLoadTriggerMethods.add("hashCode");
lazyLoadTriggerMethods.add("toString");
configuration.setLazyLoadTriggerMethods(lazyLoadTriggerMethods );
```

Mappers

MyBatis provides several ways of registering Mapper XML files and Mapper interfaces with the `Configuration` object.

- To add a single Mapper interface, use the following code:

```
configuration.addMapper(StudentMapper.class);
```

- To add all the Mapper XML files or interfaces in the `com.mybatis3.mappers` package, use the following code:

```
configuration.addMappers("com.mybatis3.mappers");
```

- To add all the Mapper interfaces that extend an interface, say `BaseMapper`, in the `com.mybatis3.mappers` package, use the following code:

```
configuration.addMappers("com.mybatis3.mappers", BaseMapper.
class);
```

Mappers should be added to the configuration only after registering typeAliases and typeHandlers if they have been used.

Customizing MyBatis logging

MyBatis uses its internal LoggerFactory as a facade to actual logging libraries. The internal LoggerFactory will delegate the logging task to one of the following actual logger implementations, with the priority decreasing from top to bottom in the given order:

- SLF4J
- Apache Commons Logging
- Log4j 2
- Log4j
- JDK logging

If MyBatis finds none of the previous implementations, logging will be disabled.

If your application is running in an environment where multiple logging libraries are available in its classpath and you want MyBatis to use a specific logging implementation, you can do this by calling one of the following methods:

- `org.apache.ibatis.logging.LogFactory.useSlf4jLogging();`
- `org.apache.ibatis.logging.LogFactory.useLog4JLogging();`
- `org.apache.ibatis.logging.LogFactory.useLog4J2Logging();`
- `org.apache.ibatis.logging.LogFactory.useJdkLogging();`
- `org.apache.ibatis.logging.LogFactory.useCommonsLogging();`
- `org.apache.ibatis.logging.LogFactory.useStdOutLogging();`

If you want to customize MyBatis logging, you should call one of these methods before calling any other MyBatis methods. If the logging library that you want to switch is not available at runtime, MyBatis will ignore the request.

Summary

In this chapter, we learned how to bootstrap MyBatis using XML and Java-API-based configuration. We also learned about various configuration options, such as type aliases, type handlers, and global settings. In the next chapter, we will discuss SQL Mappers; they are the key elements of MyBatis.

3
SQL Mappers Using XML

Relational databases and SQL are time-tested and proven data storage mechanisms. Unlike other ORM frameworks such as Hibernate, MyBatis encourages the use of SQL instead of hiding it from developers, thereby utilizing the full power of SQL provided by the database server. At the same time, MyBatis eliminates the pain of writing boilerplate code and makes using SQL easy.

Embedding SQL queries directly inside the code is a bad practice and hard to maintain. MyBatis configures SQL statements using Mapper XML files or annotations. In this chapter, we will see how to configure mapped statements in Mapper XML files in detail; we will cover the following topics:

- Mapper XMLs and Mapper interfaces
- Mapped statements
    - Configuring `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements
- ResultMaps
    - Simple ResultMaps
    - One-to-one mapping using a nested `Select` query
    - One-to-one mapping using nested results mapping
    - One-to-many mapping using a nested `Select` query
    - One-to-many mapping using nested results mapping
- Dynamic SQL
    - The `If` condition
    - The `choose` (`when`, `otherwise`) condition
    - The `trim` (`where`, `set`) condition
    - The `foreach` loop
- MyBatis recipes

Mapper XMLs and Mapper interfaces

In the previous chapters, we have seen some basic examples of how to configure mapped statements in Mapper XML files and how to invoke them using the `SqlSession` object.

Let us now see how the `findStudentById` mapped statement can be configured in `StudentMapper.xml`, which is in the `com.mybatis3.mappers` package, using the following code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

  <mapper namespace="com.mybatis3.mappers.StudentMapper">
  <select id="findStudentById" parameterType="int"
  resultType="Student">
  select stud_id as studId, name, email, dob from Students where
stud_id=#{studId}
  </select>
</mapper>
```

We can invoke the mapped statement as follows:

```
public Student findStudentById(Integer studId)
{
  SqlSession sqlSession = MyBatisUtil.getSqlSession();
  try
  {
    Student student =
sqlSession.selectOne("com.mybatis3.mappers.StudentMapper.
findStudentById", studId);
    return student;
  } finally {
    sqlSession.close();
  }
}
```

We can invoke mapped statements such as the previous one using string literals (`namespace` and `statement id`), but this exercise is error prone. You need to make sure to pass the valid input type parameter and assign the result to a valid return type variable by checking it in the Mapper XML file.

MyBatis provides a better way of invoking mapped statements by using Mapper interfaces. Once we have configured the mapped statements in the Mapper XML file, we can create a Mapper interface with a fully qualified name that is the same as the namespace and add the method signatures with matching statement IDs, input parameters, and return types.

For the preceding `StudentMapper.xml` file, we can create a Mapper interface `StudentMapper.java` as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  Student findStudentById(Integer id);
}
```

In the `StudentMapper.xml` file, the namespace should be the same as the fully qualified name of the `StudentMapper` interface that is `com.mybatis3.mappers.StudentMapper`. Also, the statement `id`, `parameterType`, and `returnType` values in `StudentMapper.xml` should be the same as the method name, argument type, and return type in the `StudentMapper` interface respectively.

Using Mapper interfaces, you can invoke mapped statements in a type safe manner as follows:

```
public Student findStudentById(Integer studId)
{
  SqlSession sqlSession = MyBatisUtil.getSqlSession();
  try {
    StudentMapper studentMapper =
sqlSession.getMapper(StudentMapper.class);
    return studentMapper.findStudentById(studId);
  } finally {
    sqlSession.close();
  }
}
```

Even though Mapper interfaces are enabled to invoke mapped statements in a type safe manner, it is our responsibility to write Mapper interfaces with correct, matching method names, argument types, and return types. If the Mapper interface methods do not match the mapped statements in XML, you will get exceptions at runtime. Actually, specifying `parameterType` is optional; MyBatis can determine `parameterType` by using `Reflection API`. But from a readability perspective, it would be better to specify the `parameterType` attribute. If the `parameterType` attribute has not been mentioned, the developer will have to switch between Mapper XML and Java code to know what type of input parameter is being passed to that statement.

Mapped statements

MyBatis provides various elements to configure different types of statements, such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. Let us see how to configure mapped statements in detail.

The INSERT statement

An `INSERT` query can be configured in a Mapper XML file using the `<insert>` element as follows:

```
<insert id="insertStudent" parameterType="Student">
  INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL, PHONE)
  VALUES(#{studId},#{name},#{email},#{phone})
</insert>
```

Here, we are giving an ID `insertStudent` that can be uniquely identified along with the namespace `com.mybatis3.mappers.StudentMapper.insertStudent`. The `parameterType` attribute value should be a fully qualified class name or type alias name.

We can invoke this statement as follows:

```
int count =
sqlSession.insert("com.mybatis3.mappers.StudentMapper.insertStuden
t", student);
```

The `sqlSession.insert()` method returns the number of rows affected by the `INSERT` statement.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe manner as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  int insertStudent(Student student);
}
```

You can invoke the `insertStudent` mapped statement as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int count = mapper.insertStudent(student);
```

Autogenerated keys

In the preceding `INSERT` statement, we are inserting the value for the `STUD_ID` column that is an `auto_generated` primary key column. We can use the `useGeneratedKeys` and `keyProperty` attributes to let the database generate the `auto_increment` column value and set that generated value into one of the input object properties as follows:

```
<insert id="insertStudent" parameterType="Student"
useGeneratedKeys="true" keyProperty="studId">
  INSERT INTO STUDENTS(NAME, EMAIL, PHONE)
  VALUES(#{name},#{email},#{phone})
</insert>
```

Here the `STUD_ID` column value will be autogenerated by MySQL database, and the generated value will be set to the `studId` property of the `student` object.

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
mapper.insertStudent(student);
```

Now you can obtain the `STUD_ID` value of the inserted `STUDENT` record as follows:

```
int studentId = student.getStudId();
```

Some databases such as Oracle don't support `AUTO_INCREMENT` columns and use `SEQUENCE` to generate the primary key values.

Assume we have a `SEQUENCE` called `STUD_ID_SEQ` to generate the `STUD_ID` primary key values. Use the following code to generate the primary key:

```
<insert id="insertStudent" parameterType="Student">
  <selectKey keyProperty="studId" resultType="int" order="BEFORE">
    SELECT ELEARNING.STUD_ID_SEQ.NEXTVAL FROM DUAL
```

```
    </selectKey>
    INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL, PHONE)
    VALUES(#{studId},#{name},#{email},#{phone})
  </insert>
```

Here we used the `<selectKey>` subelement to generate the primary key value and stored it in the `studId` property of the `Student` object. The attribute `order="BEFORE"` indicates that MyBatis will get the primary key value, that is, the next value from the sequence and store it in the `studId` property before executing the `INSERT` query.

We can also set the primary key value using a trigger where we will obtain the next value from the sequence and set it as the primary key column value before executing the `INSERT` query.

If you are using this approach, the `INSERT` mapped statement will be as follows:

```
<insert id="insertStudent" parameterType="Student">
  INSERT INTO STUDENTS(NAME,EMAIL, PHONE)
  VALUES(#{name},#{email},#{phone})
  <selectKey keyProperty="studId" resultType="int" order="AFTER">
    SELECT ELEARNING.STUD_ID_SEQ.CURRVAL FROM DUAL
  </selectKey>
</insert>
```

The UPDATE statement

An `UPDATE` statement can be configured in the Mapper XML file using the `<update>` element as follows:

```
<update id="updateStudent" parameterType="Student">
  UPDATE STUDENTS SET NAME=#{name}, EMAIL=#{email}, PHONE=#{phone}
  WHERE STUD_ID=#{studId}
</update>
```

We can invoke this statement as follows:

```
int noOfRowsUpdated =
sqlSession.update("com.mybatis3.mappers.StudentMapper.updateStudent",
student);
```

The `sqlSession.update()` method returns the number of rows affected by this `UPDATE` statement.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe way as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  int updateStudent(Student student);
}
```

You can invoke the `updateStudent` statement using the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int noOfRowsUpdated = mapper.updateStudent(student);
```

The DELETE statement

A `DELETE` statement can be configured in the Mapper XML file using the `<delete>` element as follows:

```
<delete id="deleteStudent" parameterType="int">
  DELETE FROM STUDENTS WHERE STUD_ID=#{studId}
</delete>
```

We can invoke this statement as follows:

```
int studId =1;
int noOfRowsDeleted =
sqlSession.delete("com.mybatis3.mappers.StudentMapper.deleteStuden
t", studId);
```

The `sqlSession.delete()` method returns the number of rows affected by this `delete` statement.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe way as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  int deleteStudent(int studId);
}
```

You can invoke the `deleteStudent` statement using the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int noOfRowsDeleted = mapper.deleteStudent(studId);
```

The SELECT statement

The true power of MyBatis will be known only by finding out how flexible MyBatis is for mapping `SELECT` query results to JavaBeans.

Let us see how a simple `select` query can be configured, using the following code:

```
<select id="findStudentById" parameterType="int"
resultType="Student">
  SELECT STUD_ID, NAME, EMAIL, PHONE
  FROM STUDENTS
  WHERE STUD_ID=#{studId}
</select>
```

We can invoke this statement as follows:

```
int studId =1;
Student student = sqlSession.selectOne("com.mybatis3.mappers.
StudentMapper.findStudentById", studId);
```

The `sqlSession.selectOne()` method returns the object of the type configured for the `resultType` attribute. If the query returns multiple rows for the `sqlSession.selectOne()` method, `TooManyResultsException` will be thrown.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe manner as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  Student findStudentById(Integer studId);
}
```

You can invoke the `findStudentById` statement using the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
Student student = mapper.findStudentById(studId);
```

If you check the property values of the `Student` object, you will observe that the `studId` property value is not populated with the `stud_id` column value. This is because MyBatis automatically populates the JavaBeans properties with the column values that have a matching column name. That is why, the properties `name`, `email`, and `phone` get populated but the `studId` property does not get populated.

To resolve this, we can give alias names for the columns to match with the Java Beans property names as follows:

```
<select id="findStudentById" parameterType="int"
resultType="Student">
  SELECT STUD_ID AS studId, NAME,EMAIL, PHONE
  FROM STUDENTS
  WHERE STUD_ID=#{studId}
</select>
```

Now the `Student` bean will get populated with all the `stud_id`, `name`, `email`, and `phone` columns properly.

Now let us see how to execute a `SELECT` query that returns multiple rows as shown in the following code:

```
<select id="findAllStudents" resultType="Student">
  SELECT STUD_ID AS studId, NAME,EMAIL, PHONE
  FROM STUDENTS
</select>


List<Student> students =
sqlSession.selectList("com.mybatis3.mappers.StudentMapper.findAllS
tudents");
```

The Mapper interface `StudentMapper` can also be used as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  List<Student> findAllStudents();
}
```

Using the previous code, you can invoke the `findAllStudents` statement with the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
List<Student> students = mapper.findAllStudents();
```

If you observe the preceding `SELECT` query mappings, we are giving the alias name for `stud_id` in all the mapped statements.

Instead of repeating the alias names everywhere, we can use `ResultMaps`, which we are going to discuss in a moment.

Instead of `java.util.List`, you can also use other types of collections, such as `Set`, `Map`, and `SortedSet`. Based on the type of the collection, MyBatis will use an appropriate collection implementation as follows:

- For the `List`, `Collection`, or `Iterable` types, `java.util.ArrayList` will be returned
- For the `Map` type, `java.util.HashMap` will be returned
- For the `Set` type, `java.util.HashSet` will be returned
- For the `SortedSet` type, `java.util.TreeSet` will be returned

ResultMaps

ResultMaps are used to map the SQL `SELECT` statement's results to JavaBeans properties. We can define ResultMaps and reference this `resultMap` query from several `SELECT` statements. The MyBatis ResultMaps feature is so powerful that you can use it for mapping simple `SELECT` statements to complex `SELECT` statements with one-to-one and one-to-many associations.

Simple ResultMaps

A simple `resultMap` query that maps query results to the `Student` JavaBeans is as follows:

```
<resultMap id="StudentResult" type="com.mybatis3.domain.Student">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <result property="phone" column="phone"/>
</resultMap>

<select id="findAllStudents" resultMap="StudentResult" >
  SELECT * FROM STUDENTS
</select>

<select id="findStudentById" parameterType="int"
resultMap="StudentResult">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}
</select>
```

The `id` attribute of `resultMap` `StudentResult` should be unique within the `namespace`, and the type should be a fully qualified name or alias name of the return type.

The <result> sub-elements are used to map a resultset column to a JavaBeans property.

The <id> element is similar to <result> but is used to map the identifier property that is used for comparing objects.

In the <select> statement, we have used the resultMap attribute instead of resultType to refer the StudentResult mapping. When a resultMap attribute is configured for a <select> statement, MyBatis uses the column for property mappings in order to populate the JavaBeans properties.

We can use either resultType or resultMap for a SELECT mapped statement, but not both.

Let us see another example of a <select> mapped statement showing how to populate query results into HashMap as follows:

```
<select id="findStudentById" parameterType="int" resultType="map">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}
</select>
```

In the preceding <select> statement, we configured resultType to be map, that is, the alias name for java.util.HashMap. In this case, the column names will be the key and the column value will be the value.

```
HashMap<String,Object> studentMap = sqlSession.selectOne("com.
mybatis3.mappers.StudentMapper.findStudentById", studId);
System.out.println("stud_id :"+studentMap.get("stud_id"));
System.out.println("name :"+studentMap.get("name"));
System.out.println("email :"+studentMap.get("email"));
System.out.println("phone :"+studentMap.get("phone"));
```

Let us see another example using resultType="map" that returns multiple rows.

```
<select id="findAllStudents" resultType="map">
  SELECT STUD_ID, NAME, EMAIL, PHONE FROM STUDENTS
</select>
```

As resultType="map" and the statement return multiple rows, the final return type would be List<HashMap<String,Object>> as shown in the following code:

```
List<HashMap<String,Object>> studentMapList =
sqlSession.selectList("com.mybatis3.mappers.StudentMapper.findAllS
tudents");

for(HashMap<String,Object> studentMap : studentMapList)
```

```
{
  System.out.println("studId :"+studentMap.get("stud_id"));
  System.out.println("name :"+studentMap.get("name"));
  System.out.println("email :"+studentMap.get("email"));
  System.out.println("phone :"+studentMap.get("phone"));
}
```

Extending ResultMaps

We can extend one `<resultMap>` query from another `<resultMap>` query, thereby inheriting the column to do property mappings from the one that is being extended.

```
<resultMap type="Student" id="StudentResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <result property="phone" column="phone"/>
</resultMap>

<resultMap type="Student" id="StudentWithAddressResult"
extends="StudentResult">
  <result property="address.addrId" column="addr_id"/>
  <result property="address.street" column="street"/>
  <result property="address.city" column="city"/>
  <result property="address.state" column="state"/>
  <result property="address.zip" column="zip"/>
  <result property="address.country" column="country"/>
</resultMap>
```

The `resultMap` query with the ID `StudentWithAddressResult` extends the `resultMap` with the ID `StudentResult`.

Now you can use `StudentResult resultMap` if you want to map only the `Student` data as shown in the following code:

```
<select id="findStudentById" parameterType="int"
resultMap="StudentResult">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}
</select>
```

If you want to map the query results with `Student` along with the `Address` data, you can use `resultMap` with the ID `StudentWithAddressResult` as follows:

```
<select id="selectStudentWithAddress" parameterType="int"
resultMap="StudentWithAddressResult">
  SELECT STUD_ID, NAME, EMAIL, PHONE, A.ADDR_ID, STREET, CITY,
```

```
  STATE, ZIP, COUNTRY
  FROM STUDENTS S LEFT OUTER JOIN ADDRESSES A ON
  S.ADDR_ID=A.ADDR_ID
  WHERE STUD_ID=#{studId}
</select>
```

One-to-one mapping

In our sample domain model, each student has an associated address. The `STUDENTS` table has an `ADDR_ID` column that is a foreign key to the `ADDRESSES` table.

The `STUDENTS` table's sample data is as follows:

STUD_ID	NAME	E-MAIL	PHONE	ADDR_ID
1	John	john@gmail.com	123-456-7890	1
2	Paul	paul@gmail.com	111-222-3333	2

The `ADDRESSES` table's sample data is as follows:

ADDR_ID	STREET	CITY	STATE	ZIP	COUNTRY
1	Naperville	CHICAGO	IL	60515	USA
2	Elgin	CHICAGO	IL	60515	USA

Let us see how to fetch `Student` details along with `Address` details.

The `Student` and `Address` JavaBeans are created as follows:

```
public class Address
{
  private Integer addrId;
  private String street;
  private String city;
  private String state;
  private String zip;
  private String country;
  // setters & getters
}
public class Student
{
  private Integer studId;
  private String name;
```

```
  private String email;
  private PhoneNumber phone;
  private Address address;
  //setters & getters
}

<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <result property="phone" column="phone"/>
  <result property="address.addrId" column="addr_id"/>
  <result property="address.street" column="street"/>
  <result property="address.city" column="city"/>
  <result property="address.state" column="state"/>
  <result property="address.zip" column="zip"/>
  <result property="address.country" column="country"/>
</resultMap>

<select id="selectStudentWithAddress" parameterType="int"
  resultMap="StudentWithAddressResult">
  SELECT STUD_ID, NAME, EMAIL, A.ADDR_ID, STREET, CITY, STATE,
  ZIP, COUNTRY
  FROM STUDENTS S LEFT OUTER JOIN ADDRESSES A ON
  S.ADDR_ID=A.ADDR_ID
  WHERE STUD_ID=#{studId}
</select>
```

We can set the properties of a nested object using the dot notation. In the preceding `resultMap`, Student's address property values are set by `address` column values using dot notation. Likewise, we can refer the properties of nested objects to any depth. We can access the nested object properties as follows:

```
public interface StudentMapper
{
  Student selectStudentWithAddress(int studId);
}

int studId = 1;
StudentMapper studentMapper =
sqlSession.getMapper(StudentMapper.class);
Student student = studentMapper.selectStudentWithAddress(studId);
System.out.println("Student :"+student);
System.out.println("Address :"+student.getAddress());
```

The preceding example shows one way of mapping a one-to-one association. However with this approach, if the `address` results need to be mapped to the `Address` object values in other `Select` mapped statements, we'll need to repeat the mappings for each statement.

MyBatis provides better approaches for mapping one-to-one associations using the `Nested ResultMap` and `Nested Select` statements, which is what we are going to discuss next.

One-to-one mapping using nested ResultMap

We can get `Student` along with the `Address` details using a nested ResultMap as follows:

```
<resultMap type="Address" id="AddressResult">
  <id property="addrId" column="addr_id"/>
  <result property="street" column="street"/>
  <result property="city" column="city"/>
  <result property="state" column="state"/>
  <result property="zip" column="zip"/>
  <result property="country" column="country"/>
</resultMap>

<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <association property="address" resultMap="AddressResult"/>
</resultMap>

<select id="findStudentWithAddress" parameterType="int"
resultMap="StudentWithAddressResult">
  SELECT STUD_ID, NAME, EMAIL, A.ADDR_ID, STREET, CITY, STATE,
ZIP, COUNTRY
  FROM STUDENTS S LEFT OUTER JOIN ADDRESSES A ON
S.ADDR_ID=A.ADDR_ID
  WHERE STUD_ID=#{studId}
</select>
```

The `<association>` element can be used to load the `has-one` type of associations. In the preceding example, we used the `<association>` element, referencing another `<resultMap>` that is declared in the same XML file.

We can also use `<association>` with an inline `resultMap` query as follows:

```
<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <association property="address" javaType="Address">
    <id property="addrId" column="addr_id"/>
    <result property="street" column="street"/>
    <result property="city" column="city"/>
    <result property="state" column="state"/>
    <result property="zip" column="zip"/>
    <result property="country" column="country"/>
  </association>
</resultMap>
```

Using the nested ResultMap approach, the association data will be loaded using a single query (along with `joins` if required).

One-to-one mapping using nested Select

We can get `Student` along with the `Address` details using a nested `Select` query as follows:

```
<resultMap type="Address" id="AddressResult">
  <id property="addrId" column="addr_id"/>
  <result property="street" column="street"/>
  <result property="city" column="city"/>
  <result property="state" column="state"/>
  <result property="zip" column="zip"/>
  <result property="country" column="country"/>
</resultMap>

<select id="findAddressById" parameterType="int"
resultMap="AddressResult">
  SELECT * FROM ADDRESSES WHERE ADDR_ID=#{id}
</select>

<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <association property="address" column="addr_id"
select="findAddressById"/>
</resultMap>
```

```
<select id="findStudentWithAddress" parameterType="int"
resultMap="StudentWithAddressResult">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{Id}
</select>
```

In this approach, the `<association>` element's `select` attribute is set to the statement `id` `findAddressById`. Here, two separate SQL statements will be executed against the database, the first one called `findStudentById` to load student details and the second one called `findAddressById` to load its address details.

The `addr_id` column value will be passed as input to the `selectAddressById` statement.

We can invoke the `findStudentWithAddress` mapped statement as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
Student student = mapper.selectStudentWithAddress(studId);
System.out.println(student);
System.out.println(student.getAddress());
```

One-to-many mapping

In our sample domain model, a tutor can teach one or more courses. This means that there is a one-to-many relationship between the tutor and course.

We can map one-to-many types of results to a collection of objects using the `<collection>` element.

The `TUTORS` table's sample data is as follows:

TUTOR_ID	NAME	EMAIL	PHONE	ADDR_ID
1	John	john@gmail.com	123-456-7890	1
2	Ying	ying@gmail.com	111-222-3333	2

The `COURSES` table's sample data is as follows:

COURSE_ID	NAME	DESCRIPTION	START_DATE	END_DATE	TUTOR_ID
1	JavaSE	Java SE	2013-01-10	2013-02-10	1
2	JavaEE	JavaEE6	2013-01-10	2013-03-10	2
3	MyBatis	MyBatis	2013-01-10	2013-02-20	2

In the preceding table data, the tutor John teaches one course whereas the tutor Ying teaches two courses.

The JavaBeans for `Course` and `Tutor` are as follows:

```
public class Course
{
  private Integer courseId;
  private String name;
  private String description;
  private Date startDate;
  private Date endDate;
  private Integer tutorId;

  //setters & getters
}

public class Tutor
{
  private Integer tutorId;
  private String name;
  private String email;
  private Address address;
  private List<Course> courses;
  /setters & getters
}
```

Now let us see how we can get the tutor's details along with the list of courses he/she teaches.

The `<collection>` element can be used to map multiple course rows to a list of course objects. Similar to one-to-one mapping, we can map one-to-many relationships using a nested ResultMap and nested `Select` approaches.

One-to-many mapping with nested ResultMap

We can get the tutor along with the courses' details using a nested ResultMap as follows:

```
<resultMap type="Course" id="CourseResult">
  <id column="course_id" property="courseId"/>
  <result column="name" property="name"/>
  <result column="description" property="description"/>
  <result column="start_date" property="startDate"/>
  <result column="end_date" property="endDate"/>
```

```
</resultMap>

<resultMap type="Tutor" id="TutorResult">
  <id column="tutor_id" property="tutorId"/>
  <result column="tutor_name" property="name"/>
  <result column="email" property="email"/>
  <collection property="courses" resultMap="CourseResult"/>
</resultMap>

<select id="findTutorById" parameterType="int"
resultMap="TutorResult">
  SELECT T.TUTOR_ID, T.NAME AS TUTOR_NAME, EMAIL, C.COURSE_ID,
  C.NAME, DESCRIPTION, START_DATE, END_DATE
  FROM TUTORS T LEFT OUTER JOIN ADDRESSES A ON T.ADDR_ID=A.ADDR_ID
  LEFT OUTER JOIN COURSES C ON T.TUTOR_ID=C.TUTOR_ID
  WHERE T.TUTOR_ID=#{tutorId}
</select>
```

Here we are fetching the tutor along with the courses' details using a single `Select` query with `JOINS`. The `<collection>` element's `resultMap` is set to the `resultMap` ID `CourseResult` that contains the mapping for the `Course` object's properties.

One-to-many mapping with nested select

We can get the tutor along with the courses' details using a nested `select` query as follows:

```
<resultMap type="Course" id="CourseResult">
  <id column="course_id" property="courseId"/>
  <result column="name" property="name"/>
  <result column="description" property="description"/>
  <result column="start_date" property="startDate"/>
  <result column="end_date" property="endDate"/>
</resultMap>

<resultMap type="Tutor" id="TutorResult">
  <id column="tutor_id" property="tutorId"/>
  <result column="tutor_name" property="name"/>
  <result column="email" property="email"/>
  <association property="address" resultMap="AddressResult"/>
  <collection property="courses" column="tutor_id"
select="findCoursesByTutor"/>
  </resultMap>

<select id="findTutorById" parameterType="int"
```

```
resultMap="TutorResult">
  SELECT T.TUTOR_ID, T.NAME AS TUTOR_NAME, EMAIL
  FROM TUTORS T WHERE T.TUTOR_ID=#{tutorId}
  </select>

<select id="findCoursesByTutor" parameterType="int"
resultMap="CourseResult">
  SELECT * FROM COURSES WHERE TUTOR_ID=#{tutorId}
</select>
```

In this approach, the `<association>` element's `select` attribute is set to the statement ID `findCoursesByTutor` that triggers a separate SQL query to load the courses' details. The `tutor_id` column value will be passed as input to the `findCoursesByTutor` statement.

```
public interface TutorMapper
{
  Tutor findTutorById(int tutorId);
}
TutorMapper mapper = sqlSession.getMapper(TutorMapper.class);
Tutor tutor = mapper.findTutorById(tutorId);
System.out.println(tutor);
List<Course> courses = tutor.getCourses();
for (Course course : courses)
{
  System.out.println(course);
}
```

A nested select approach may result in N+1 select problems. First, the main query will be executed (1), and for every row returned by the first query, another select query will be executed (*N* queries for *N* rows). For large datasets, this could result in poor performance.

Dynamic SQL

Sometimes, static SQL queries may not be sufficient for application requirements. We may have to build queries dynamically, based on some criteria.

For example, in web applications there could be search screens that provide one or more input options and perform searches based on the chosen criteria. While implementing this kind of search functionality, we may need to build a dynamic query based on the selected options. If the user provides any value for input criteria, we'll need to add that field in the `WHERE` clause of the query.

MyBatis provides first-class support for building dynamic SQL queries using elements such as `<if>`, `<choose>`, `<where>`, `<foreach>`, and `<trim>`.

The If condition

The `<if>` element can be used to conditionally embed SQL snippets. If the test condition is evaluated to `true`, then only the SQL snippet will be appended to the query.

Assume we have a Search Courses Screen that has a `Tutor` dropdown, the `CourseName` text field, and the `StartDate` and `End Date` input fields as the search criteria.

Assume that `Tutor` is a mandatory field and that the rest of the fields are optional.

When the user clicks on the search button, we need to display a list of courses that meet the following criteria:

- Courses by the selected `Tutor`
- Courses whose name contain the entered course name; if nothing has been provided, fetch all the courses
- Courses whose start date and end date are in between the provided `StartDate` and `EndDate` input fields

We can create the mapped statement for searching the courses as follows:

```
<resultMap type="Course" id="CourseResult">
  <id column="course_id" property="courseId"/>
  <result column="name" property="name"/>
  <result column="description" property="description"/>
  <result column="start_date" property="startDate"/>
  <result column="end_date" property="endDate"/>
</resultMap>

<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
<![CDATA[
  SELECT * FROM COURSES
  WHERE TUTOR_ID= #{tutorId}
  <if test="courseName != null">
    AND NAME LIKE #{courseName}
  </if>
  <if test="startDate != null">
    AND START_DATE >= #{startDate}
  </if>
```

```
    <if test="endDate != null">
      AND END_DATE <= #{endDate}
    </if>
  ]]>
  </select>

  public interface CourseMapper
  {
    List<Course> searchCourses(Map<String, Object> map);
  }

  public void searchCourses()
  {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("tutorId", 1);
    map.put("courseName", "%java%");
    map.put("startDate", new Date());
    CourseMapper mapper = sqlSession.getMapper(CourseMapper.class);
    List<Course> courses = mapper.searchCourses(map);
    for (Course course : courses) {
      System.out.println(course);

  }
```

This will generate the query `SELECT * FROM COURSES WHERE TUTOR_ID= ? AND NAME like ? AND START_DATE >= ?`. This will come in handy while preparing a dynamic SQL query based on the given criteria.

MyBatis uses **OGNL** (**Object Graph Navigation Language**) expressions for building dynamic queries.

The choose, when, and otherwise conditions

Sometimes, search functionality could be based on the search type. First, the user needs to choose whether he wants to search by `Tutor` or `Course Name` or `Start Dates` and `End Dates`, and then based on the selected search type, the input field will appear. In such scenarios, we should apply only one of the conditions.

MyBatis provides the `<choose>` element to support this kind of dynamic SQL preparation.

Now let us write a SQL mapped statement to get the courses by applying the search criteria. If no search criteria is selected, the courses starting from today onwards should be fetched as follows:

```
<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <choose>
    <when test="searchBy == 'Tutor'">
      WHERE TUTOR_ID= #{tutorId}
    </when>
    <when test="searchBy == 'CourseName'">
      WHERE name like #{courseName}
    </when>
    <otherwise>
      WHERE TUTOR start_date  &gt;= now()
    </otherwise>
  </choose>
</select>
```

MyBatis evaluates the `<choose>` test conditions and uses the clause with the first condition that evaluates to `TRUE`. If none of the conditions are true, the `<otherwise>` clause will be used.

The where condition

At times, all the search criteria might be optional. In cases where at least one of the search conditions needs to be applied, then only the `WHERE` clause should be appended. Also, we need to append `AND` or `OR` to the conditions only if there are multiple conditions. MyBatis provides the `<where>` element to support building these kinds of dynamic SQL statements.

In our example Search Courses screen, we assume that all the search criteria is optional. So, the `WHERE` clause should be there only if any of the search criteria has been provided.

```
<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <where>
    <if test=" tutorId != null ">
      TUTOR_ID= #{tutorId}
    </if>
    <if test="courseName != null">
      AND name like #{courseName}
```

```
      </if>
      <if test="startDate != null">
        AND start_date  &gt;= #{startDate}
      </if>
      <if test="endDate != null">
        AND end_date  &lt;= #{endDate}
      </if>
    </where>
  </select>
```

The `<where>` element inserts `WHERE` only if any content is returned by the inner conditional tags. Also, it removes the `AND` or `OR` prefixes if the `WHERE` clause begins with `AND` or `OR`.

In the preceding example, if none of the `<if>` conditions are `True`, `<where>` won't insert the `WHERE` clause. If at least one of the `<if>` conditions is `True`, `<where>` will insert the `WHERE` clause followed by the content returned by the `<if>` tags.

If the `tutor_id` parameter is `null` and the `courseName` parameter is not `null`, `<where>` will take care of stripping out the `AND` prefix and adding `NAME like #{courseName}`.

The trim condition

The `<trim>` element works similar to `<where>` but provides additional flexibility on what prefix/suffix needs to be prefixed/suffixed and what prefix/suffix needs to be stripped off.

```
<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <trim prefix="WHERE" prefixOverrides="AND | OR">
    <if test=" tutorId != null ">
      TUTOR_ID= #{tutorId}
    </if>
    <if test="courseName != null">
      AND name like #{courseName}
    </if>
  </trim>
</select>
```

Here `<trim>` will insert `WHERE` if any of the `<if>` conditions are `true` and remove the `AND` or `OR` prefixes just after `WHERE`.

The foreach loop

Another powerful dynamic SQL builder tag is `<foreach>`. It is a very common requirement for iterating through an array or list and for building `AND/OR` conditions or an `IN` clause.

Suppose we want to find out all the courses taught by the tutors whose `tutor_id` IDs are `1`, `3`, and `6`. We can pass a list of `tutor_id` IDs to the mapped statement and build a dynamic query by iterating through the list using `<foreach>`.

```
<select id="searchCoursesByTutors" parameterType="map"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <if test="tutorIds != null">
    <where>
      <foreach item="tutorId" collection="tutorIds">
        OR tutor_id=#{tutorId}
      </foreach>
    </where>
  </if>
</select>

public interface CourseMapper
{
  List<Course> searchCoursesByTutors(Map<String, Object> map);
}

public void searchCoursesByTutors()
{
  Map<String, Object> map = new HashMap<String, Object>();
  List<Integer> tutorIds = new ArrayList<Integer>();
  tutorIds.add(1);
  tutorIds.add(3);
  tutorIds.add(6);
  map.put("tutorIds", tutorIds);
  CourseMapper mapper =
  sqlSession.getMapper(CourseMapper.class);
  List<Course> courses = mapper.searchCoursesByTutors(map);
  for (Course course : courses)
  {
    System.out.println(course);
  }
}
```

Let us see how to use `<foreach>` to generate the `IN` clause:

```
<select id="searchCoursesByTutors" parameterType="map"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <if test="tutorIds != null">
    <where>
      tutor_id IN
      <foreach item="tutorId" collection="tutorIds"
        open="(" separator="," close=")">
        #{tutorId}
      </foreach>
    </where>
  </if>
</select>
```

The set condition

The `<set>` element is similar to the `<where>` element and will insert `SET` if any content is returned by the inner conditions.

```
<update id="updateStudent" parameterType="Student">
  update students
  <set>
    <if test="name != null">name=#{name},</if>
    <if test="email != null">email=#{email},</if>
    <if test="phone != null">phone=#{phone},</if>
  </set>
  where stud_id=#{id}
</update>
```

Here, `<set>` inserts the `SET` keyword if any of the `<if>` conditions return text and also strips out the tailing commas at the end.

In the preceding example, if `phone != null`, `<set>` will take care of removing the comma after `phone=#{phone}`.

MyBatis recipes

In addition to simplifying the database programming, MyBatis provides various features that are very useful for implementing some common tasks, such as loading the table rows page by page, storing and retrieving `CLOB/BLOB` type data, and handling enumerated type values, among others. Let us have a look at a few of these features.

Handling enumeration types

MyBatis supports persisting `enum` type properties out of the box. Assume that the `STUDENTS` table has a column `gender` of the type `varchar` to store either `MALE` or `FEMALE` as the value. And, the `Student` object has a `gender` property that is of the type `enum` as shown in the following code:

```
public enum Gender
{
  FEMALE,
  MALE
}
```

By default, MyBatis uses `EnumTypeHandler` to handle `enum` type Java properties and stores the name of the `enum` value. You don't need any extra configuration to do this. You can use `enum` type properties just like primitive type properties as shown in the following code:

```
public class Student
{
  private Integer id;
  private String name;
  private String email;
  private PhoneNumber phone;
  private Address address;
  private Gender gender;
  //setters and getters
}

<insert id="insertStudent" parameterType="Student"
useGeneratedKeys="true" keyProperty="id">
  insert into students(name,email,addr_id, phone,gender)
  values(#{name},#{email},#{address.addrId},#{phone},#{gender})
</insert>
```

When you execute the `insertStudent` statement, MyBatis takes the name of the `Gender` enum (`FEMALE/MALE`) and stores it in the `GENDER` column.

If you want to store the ordinal position of the enum instead of the enum name, you will need to explicitly configure it.

So if you want to store `0` for `FEMALE` and `1` for `MALE` in the `gender` column, you'll need to register `EnumOrdinalTypeHandler` in the `mybatis-config.xml` file.

```
<typeHandler
handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"
javaType="com.mybatis3.domain.Gender"/>
```

Be careful to use ordinal values to store in the DB. Ordinal values are assigned to `enum` values based on their order of declaration. If you change the declaration order in `Gender enum`, the data in the database and ordinal values will be mismatched.

Handling the CLOB/BLOB types

MyBatis provides built-in support for mapping `CLOB/BLOB` type columns.

Assume we have the following table to store the `Students` and `Tutors` photographs and their biodata:

```
CREATE TABLE USER_PICS
(
  ID INT(11) NOT NULL AUTO_INCREMENT,
  NAME VARCHAR(50) DEFAULT NULL,
  PIC BLOB,
  BIO LONGTEXT,
  PRIMARY KEY (ID)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=LATIN1;
```

Here, the photograph can be an image of type PNG, JPG, and so on, and the biodata can be a lengthy history about the student/tutor.

By default, MyBatis maps `CLOB` type columns to the `java.lang.String` type and `BLOB` type columns to the `byte[]` type.

```
public class UserPic
{
  private int id;
  private String name;
  private byte[] pic;
  private String bio;
  //setters & getters
}
```

Create the `UserPicMapper.xml` file and configure the mapped statements as follows:

```
<insert id="insertUserPic" parameterType="UserPic">
  INSERT INTO USER_PICS(NAME, PIC,BIO)
  VALUES(#{name},#{pic},#{bio})
</insert>

<select id="getUserPic" parameterType="int" resultType="UserPic">
  SELECT * FROM USER_PICS WHERE ID=#{id}
</select>
```

The following method `insertUserPic()` shows how to insert data into `CLOB/BLOB` type columns:

```
public void insertUserPic()
{
  byte[] pic = null;
  try {
    File file = new File("C:\\Images\\UserImg.jpg");
    InputStream is = new FileInputStream(file);
    pic = new byte[is.available()];
    is.read(pic);
    is.close();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
  String name = "UserName";
  String bio = "put some lenghty bio here";
  UserPic userPic = new UserPic(0, name, pic , bio);

  SqlSession sqlSession = MyBatisUtil.openSession();
  try {
    UserPicMapper mapper =
    sqlSession.getMapper(UserPicMapper.class);
    mapper.insertUserPic(userPic);
    sqlSession.commit();
  }
  finally {
    sqlSession.close();
  }

}
```

The following method `getUserPic()` shows how to read `CLOB` type data into `String` and `BLOB` type data into `byte[]` properties:

```
public void getUserPic()
{
  UserPic userPic = null;
  SqlSession sqlSession = MyBatisUtil.openSession();
  try {
    UserPicMapper mapper =
sqlSession.getMapper(UserPicMapper.class);
    userPic =  mapper.getUserPic(1);
  }
  finally {
    sqlSession.close();
  }
  byte[] pic = userPic.getPic();
  try {
    OutputStream os = new FileOutputStream(new
File("C:\\Images\\UserImage_FromDB.jpg"));
    os.write(pic);
    os.close();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Passing multiple input parameters

MyBatis's mapped statements have the `parameterType` attribute to specify the type of input parameter. If we want to pass multiple input parameters to a mapped statement, we can put all the input parameters in a `HashMap` and pass it to that mapped statement.

MyBatis provides another way of passing multiple input parameters to a mapped statement. Suppose we want to find students with the given `name` and `email`.

```
Public interface StudentMapper
{
  List<Student> findAllStudentsByNameEmail(String name, String
email);
}
```

MyBatis supports passing multiple input parameters to a mapped statement and referencing them using the `#{param}` syntax.

```
<select id="findAllStudentsByNameEmail" resultMap="StudentResult"
>
  select stud_id, name,email, phone from Students
  where name=#{param1} and email=#{param2}
 </select>
```

Here `#{param1}` refers to the first parameter `name` and `#{param2}` refers to the second parameter `email`.

```
StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.
class);
studentMapper.findAllStudentsByNameEmail(name, email);
```

Multiple results as a map

If we have a mapped statement that returns multiple rows and we want the results in a `HashMap` with some property value as the key and the resulting object as the value, we can use `sqlSession.selectMap()` as follows:

```
<select id=" findAllStudents" resultMap="StudentResult">
  select * from Students
</select>

Map<Integer, Student> studentMap =
sqlSession.selectMap("com.mybatis3.mappers.StudentMapper.
findAllStudents", "studId");
```

Here `studentMap` will contain `studId` values as keys and `Student` objects as values.

Paginated ResultSets using RowBounds

Sometimes, we may need to work with huge volumes of data, such as with tables with millions of records. Loading all these records may not be possible due to memory constraints, or we may need only a fragment of data. Typically in web applications, pagination is used to display large volumes of data in a page-by-page style.

MyBatis can load table data page by page using `RowBounds`. The `RowBounds` object can be constructed using the `offset` and `limit` parameters. The parameter `offset` refers to the starting position and `limit` refers to the number of records.

Suppose if you want to load and display 25 student records per page, you can use the following query:

```
<select id="findAllStudents" resultMap="StudentResult">
  select * from Students
</select>
```

Then, you can load the first page (first 25 records) as follows:

```
int offset =0 , limit =25;
RowBounds rowBounds = new RowBounds(offset, limit);
List<Student> = studentMapper.getStudents(rowBounds);
```

To display the second page, use `offset=25` and `limit=25`; for the third page, use `offset=50` and `limit=25`.

Custom ResultSet processing using ResultSetHandler

MyBatis provides great support with plenty of options for mapping the query results to JavaBeans. But sometimes, we may come across scenarios where we need to process the SQL query results by ourselves for special purposes. MyBatis provides `ResultHandler` plugin that enables the processing of the ResultSet in whatever way we like.

Suppose that we want to get the student details in a `HashMap` where `stud_id` is used as a key and name is used as a value.

As of `mybatis-3.2.2`, MyBatis doesn't have support for getting the result as `HashMap`, with one property value as the key and another property value as the value, using the `resultMap` element. `sqlSession.selectMap()` returns a map with the given property value as the key and the result object as the value. We can't configure it to use one property as the key and another property as the value.

For `sqlSession.select()` methods, we can pass an implementation of `ResultHandler` that will be invoked for each record in the ResultSet.

```
public interface ResultHandler
{
  void handleResult(ResultContext context);
}
```

Now let us see how we can use `ResultHandler` to process the ResultSet and return customized results.

```
public Map<Integer, String> getStudentIdNameMap()
{
  final Map<Integer, String> map = new HashMap<Integer, String>();
  SqlSession sqlSession = MyBatisUtil.openSession();
  try {

sqlSession.select("com.mybatis3.mappers.StudentMapper.findAllStude
  nts",
  new ResultHandler() {
    @Override
    public void handleResult(ResultContext context) {
      Student student = (Student) context.getResultObject();
      map.put(student.getStudId(), student.getName());
    }
  }
);
} finally {
  sqlSession.close();
}
return map;
}
```

In the preceding code, we are providing an inline implementation of `ResultHandler`. Inside the `handleResult()` method, we are getting the current result object using `context.getResultObject()` that is a `Student` object because we configured `resultMap="StudentResult"` for the `findAllStudents` mapped statement. As the `handleResult()` method will be called for every row returned by the query, we are extracting the `studId` and `name` values from the `Student` object and populating the map.

Cache

Caching data that is loaded from the database is a common requirement for many applications to improve their performance. MyBatis provides in-built support for caching the query results loaded by mapped `SELECT` statements. By default, the first-level cache is enabled; this means that if you'll invoke the same `SELECT` statement within the same `SqlSession` interface, results will be fetched from the cache instead of the database.

We can add global second-level caches by adding the `<cache/>` element in SQL Mapper XML files.

When you'll add the `<cache/>` element the following will occur:

- All results from the `<select>` statements in the mapped statement file will be cached
- All the `<insert>`, `<update>`, and `<delete>` statements in the mapped statement file will flush the cache
- The cache will use a Least Recently Used (`LRU`) algorithm for eviction
- The cache will not flush on any sort of time-based schedule (no Flush Interval)
- The cache will store `1024` references to lists or objects (whatever the query method returns)
- The cache will be treated as a read/write cache; this means that the objects retrieved will not be shared and can safely be modified by the caller without it interfering with other potential modifications by other callers or threads

You can also customize this behavior by overriding the default attribute values as follows:

```
<cache eviction="FIFO" flushInterval="60000" size="512"
readOnly="true"/>
```

A description for each of the attributes is as follows:

- `eviction`: This is the cache eviction policy to be used. The default value is `LRU`. The possible values are `LRU` (least recently used), `FIFO`(first in first out), `SOFT`(soft reference), `WEAK`(weak reference).
- `flushInterval`: This is the cache flush interval in milliseconds. The default is not set. So, no flush interval is used and the cache is only flushed by calls to the statements.
- `size`: This represents the maximum number of elements that can be held in the cache. The default is `1024`, and you can set it to any positive integer.
- `readOnly`: A read-only cache will return the same instance of the cached object to all the callers. A read-write cache will return a copy (via serialization) of the cached object. The default is `false` and the possible values are `true` and `false`.

A cache configuration and cache instance are bound to the namespace of the SQL Mapper file, so all the statements in the same `namespace` table as the cache are bound by it.

The default cache configuration for a mapped statement is:

```
<select ... flushCache="false" useCache="true"/>
<insert ... flushCache="true"/>
<update ... flushCache="true"/>
<delete ... flushCache="true"/>
```

You can override this default behavior for any specific mapped statements; for example, by not using a cache for a `select` statement by setting the `useCache="false"` attribute.

In addition to in-built Cache support, MyBatis provides support for integration with popular third-party Cache libraries, such as `Ehcache`, `OSCache`, and `Hazelcast`. You can find more information on integrating third-party Cache libraries on the official MyBatis website `https://code.google.com/p/mybatis/wiki/Caches`.

Summary

In this chapter, we learned how to write SQL mapped statements using the Mapper XML files. We discussed how to configure simple statements, statements with one-to-one and one-to-many relationships, and how to map the results using `ResultMap`. We also looked into building dynamic queries, paginated results, and custom ResultSet handling. In the next chapter, we will discuss how to write mapped statements using annotations.

4
SQL Mappers Using Annotations

In the previous chapter, we had seen how we can configure mapped statements in XML Mapper files. MyBatis supports configuring mapped statements using annotations also. When using annotation-based Mapper interfaces you don't need to configure SQL queries in XML files. If you want, you can use XML and annotation-based mapped statements together.

In this chapter we will cover the following topics:

- Mapper interfaces using annotations
- Mapped statements
    - `@Insert`, `@Update`, `@Delete`, and `@SelectStatements`
- Resultmaps
    - Simple resultmaps
    - One-to-one mapping
    - One-to-many mapping
- Dynamic SQL
    - `@SelectProvider`
    - `@InsertProvider`
    - `@UpdateProvider`
    - `@DeleteProvider`

Mapper interfaces using annotations

MyBatis provides annotation-based configuration options for most of the XML-based mapper elements, including `<select>` and`<update>`. However, there are few cases where there are no annotation-based equivalents for some of the XML-based elements.

Mapped statements

MyBatis provides various annotations to configure different types of statements such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. Let us see how to configure mapped statements in detail.

@Insert

We can define an `INSERT` mapped statement using the `@Insert` annotation.

```
package com.mybatis3.mappers;
public interface StudentMapper
{
@Insert("INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL,ADDR_ID, PHONE)
VALUES(#{studId},#{name},#{email},#{address.addrId},#{phone})")
int insertStudent(Student student);
}
```

The `insertStudent()` method with the `@Insert` annotation returns the number of rows affected by this `insert` statement.

Autogenerated keys

As discussed in the previous chapter, there can be autogenerated primary key columns. We can use the `useGeneratedKeys` and `keyProperty` attributes of the `@Options` annotation to let the database server generate the `auto_increment` column value and set that generated value as one of the input object properties.

```
@Insert("INSERT INTO STUDENTS(NAME,EMAIL,ADDR_ID, PHONE)
VALUES(#{name},#{email},#{address.addrId},#{phone})")
@Options(useGeneratedKeys=true, keyProperty="studId")
int insertStudent(Student student);
```

Here the `STUD_ID` column value will be autogenerated by MySQL database and the generated value will be set to the `studId` property of the `student` object.

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
mapper.insertStudent(student);
int studentId = student.getStudId();
```

Some of the databases, such as Oracle, don't support the `AUTO_INCREMENT` columns and generally we use `SEQUENCE` to generate the primary key values.

We can use the `@SelectKey` annotation to specify any SQL statement that will give the primary key value, which can be used as the primary key column value.

Assume we have a sequence called `STUD_ID_SEQ` to generate the `STUD_ID` primary key values.

```
@Insert("INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL,ADDR_ID, PHONE)
VALUES(#{studId},#{name},#{email},#{address.addrId},#{phone})")
@SelectKey(statement="SELECT STUD_ID_SEQ.NEXTVAL FROM DUAL",
keyProperty="studId", resultType=int.class, before=true)
int insertStudent(Student student);
```

Here we have used `@SelectKey` to generate the primary key value and stored it in the `studId` property of the `Student` object using the `keyProperty` attribute. This gets executed before executing the `INSERT` statement, because we specified it via the `before=true` attribute.

If you are setting the primary key value through triggers using `SEQUENCE`, we can obtain the database-generated primary key value from `sequence_name.currval` after the `INSERT` statement is executed.

```
@Insert("INSERT INTO STUDENTS(NAME,EMAIL,ADDR_ID, PHONE)
VALUES(#{name},#{email},#{address.addrId},#{phone})")
@SelectKey(statement="SELECT STUD_ID_SEQ.CURRVAL FROM DUAL",
keyProperty="studId", resultType=int.class, before=false)
int insertStudent(Student student);
```

@Update

We can define an `UPDATE` mapped statement using the `@Update` annotation as follows:

```
@Update("UPDATE STUDENTS SET NAME=#{name}, EMAIL=#{email},
PHONE=#{phone} WHERE STUD_ID=#{studId}")
int updateStudent(Student student);
```

The `updateStudent()` method with `@Update` returns the number of rows affected by this `update` statement.

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int noOfRowsUpdated = mapper.updateStudent(student);
```

@Delete

We can define a `DELETE` mapped statement using the `@Delete` annotation as follows:

```
@Delete("DELETE FROM STUDENTS WHERE STUD_ID=#{studId}")
int deleteStudent(int studId);
```

The `deleteStudent()` method with `@Delete` returns the number of rows affected by this delete statement.

@Select

We can define the `SELECT` mapped statements using the `@Select` annotation.

Let us see how a simple select query can be configured.

```
package com.mybatis3.mappers;
public interface StudentMapper
{
@Select("SELECT STUD_ID AS STUDID, NAME, EMAIL, PHONE FROM
STUDENTS WHERE STUD_ID=#{studId}")
Student findStudentById(Integer studId);
}
```

To match the column names with the `Student` bean property names, we gave an alias name for `stud_id` as `studId`. If the query returns multiple rows, `TooManyResultsException` will be thrown.

Result maps

We can map query results to JavaBean properties using inline aliases or using an explicit `@Results` annotation.

Now let us see how to execute a `SELECT` query with explicit column to property mappings using the `@Results` annotation.

```
package com.mybatis3.mappers;
public interface StudentMapper
{
@Select("SELECT * FROM STUDENTS")
@Results({
@Result(id=true, column="stud_id", property="studId"),
@Result(column="name", property="name"),
@Result(column="email", property="email"),
@Result(column="addr_id", property="address.addrId")
```

```
})
List<Student> findAllStudents();
}
```

The `@Results` annotation is a counterpart of the Mapper XML element `<resultMap>`. However, as of MyBatis 3.2.2 we can't give an ID for the `@Results` annotation. So unlike the `<resultMap>`XML element, we can't reuse the `@Results` declaration across different mapped statements. What this means is that you need to duplicate the `@Results` configuration even though it is the same.

For example, see the following `findStudentBy()` and `findAllStudents()` methods:

```
@Select("SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}")
@Results({
@Result(id=true, column="stud_id", property="studId"),
@Result(column="name", property="name"),
@Result(column="email", property="email"),
@Result(column="addr_id", property="address.addrId")
})
Student findStudentById(int studId);

@Select("SELECT * FROM STUDENTS")
@Results({
@Result(id=true, column="stud_id", property="studId"),
@Result(column="name", property="name"),
@Result(column="email", property="email"),
@Result(column="addr_id", property="address.addrId")
})
List<Student> findAllStudents();
```

Here the `@Results` configuration is same for both the statements, but we need to duplicate it. There is also a work around for this problem. We can create a Mapper XML file and configure the `<resultMap>` element and reference that `resultMap` using the `@ResultMap` annotation.

Define `<resultMap>` with ID `StudentResult` in `StudentMapper.xml`.

```
<mapper namespace="com.mybatis3.mappers.StudentMapper">
<resultMap type="Student" id="StudentResult">
<id property="studId" column="stud_id"/>
<result property="name" column="name"/>
<result property="email" column="email"/>
<result property="phone" column="phone"/>
</resultMap>
```

```
</mapper>
```

In `StudentMapper.java`, reference the `resultMap` attribute `StudentResult` using `@ResultMap`.

```
public interface StudentMapper
{
@Select("SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}")
@ResultMap("com.mybatis3.mappers.StudentMapper.StudentResult")
Student findStudentById(int studId);

@Select("SELECT * FROM STUDENTS")
@ResultMap("com.mybatis3.mappers.StudentMapper.StudentResult")
List<Student> findAllStudents();
}
```

One-to-one mapping

MyBatis provides the `@One` annotation to load a one-to-one association using a `Nested-Select` statement.

Let us see how we can get student details along with their address details using the `@One` annotation.

```
public interface StudentMapper
{
@Select("SELECT ADDR_ID AS ADDRID, STREET, CITY, STATE, ZIP, COUNTRY
FROM ADDRESSES WHERE ADDR_ID=#{id}")
Address findAddressById(int id);

@Select("SELECT * FROM STUDENTS WHERE STUD_ID=#{studId} ")
@Results({
@Result(id=true, column="stud_id", property="studId"),
@Result(column="name", property="name"),
@Result(column="email", property="email"),
@Result(property="address", column="addr_id",
one=@One(select="com.mybatis3.mappers.StudentMapper.
findAddressById"))
})
Student selectStudentWithAddress(int studId);
}
```

Here we have used the `@One` annotation's `select` attribute to point to fully qualified method names, which return an `Address` object. With the attribute `column="addr_id"`, the column value `addr_id` from the table row `STUDENTS` will be passed as an input to the `findAddressById()` method. If the query `@One SELECT` returns multiple rows, `TooManyResultsException` will be thrown.

```
int studId = 1;
StudentMapper studentMapper =
sqlSession.getMapper(StudentMapper.class);
Student student = studentMapper.selectStudentWithAddress(studId);
System.out.println("Student :"+student);
System.out.println("Address :"+student.getAddress());
```

As discussed in *Chapter 3, SQL Mappers using XML,* we can load a one-to-one association using nested ResultMap using XML-based Mapper configuration. But as of MyBatis-3.2.2, there is no annotation-based counterpart for this kind of mapping. However, we can define `<resultMap>` in Mapper XML file and reference it using the `@ResultMap` annotation.

Configure `<resultMap>` in `StudentMapper.xml` as follows:

```
<mapper namespace="com.mybatis3.mappers.StudentMapper">
<resultMap type="Address" id="AddressResult">
<id property="addrId" column="addr_id"/>
<result property="street" column="street"/>
<result property="city" column="city"/>
<result property="state" column="state"/>
<result property="zip" column="zip"/>
<result property="country" column="country"/>
</resultMap>

<resultMap type="Student" id="StudentWithAddressResult">
<id property="studId" column="stud_id"/>
<result property="name" column="name"/>
<result property="email" column="email"/>
<association property="address" resultMap="AddressResult"/>
</resultMap>
</mapper>

public interface StudentMapper
{
@Select("select stud_id, name, email, a.addr_id, street, city,
state, zip, country"+" FROM students s left outer join addresses a
on s.addr_id=a.addr_id"+" where stud_id=#{studId} ")
@ResultMap("com.mybatis3.mappers.StudentMapper.
StudentWithAddressResult")
Student selectStudentWithAddress(int id);
}
```

One-to-many mapping

MyBatis provides the `@Many` annotation to load a one-to-many association using a `Nested-Select` statement.

Now let us see how we can get a `Tutor` with a list of courses he/she teaches using the `@Many` annotation.

```
public interface TutorMapper
{
@Select("select addr_id as addrId, street, city, state, zip,
country from addresses where addr_id=#{id}")
Address findAddressById(int id);

@Select("select * from courses where tutor_id=#{tutorId}")
@Results({
@Result(id=true, column="course_id", property="courseId"),
@Result(column="name", property="name"),
@Result(column="description", property="description"),
@Result(column="start_date" property="startDate"),
@Result(column="end_date" property="endDate")

})
List<Course> findCoursesByTutorId(int tutorId);

@Select("SELECT tutor_id, name as tutor_name, email, addr_id
FROM tutors where tutor_id=#{tutorId}")
@Results({
@Result(id=true, column="tutor_id", property="tutorId"),
@Result(column="tutor_name", property="name"),
@Result(column="email", property="email"),
@Result(property="address", column="addr_id",
one=@One(select=" com.mybatis3.
mappers.TutorMapper.findAddressById")),
@Result(property="courses", column="tutor_id",
many=@Many(select="com.mybatis3.mappers.TutorMapper.
findCoursesByTutorId"))
})
Tutor findTutorById(int tutorId);
}
```

Here we have used the `@Many` annotation's select attribute to point to a fully qualified method name, which returns the `List<Course>` objects. With the attribute `column="tutor_id"`, the `tutor_id` column value from the `TUTORS` table row will be passed as an input to the `findCoursesByTutorId()` method.

Using an XML-based Mapper configuration, we can load a one-to-many association using a nested ResultMap as discussed in *Chapter3, SQL Mappers Using XML*. As of MyBatis 3.2.2, there is no annotation-based counterpart for this kind of mapping. But we can define the `<resultMap>` in the Mapper XML file and reference it using the `@ResultMap` annotation.

Configure `<resultMap>` in `TutorMapper.xml` as follows:

```
<mapper namespace="com.mybatis3.mappers.TutorMapper">
<resultMap type="Address" id="AddressResult">
<id property="addrId" column="addr_id"/>
<result property="street" column="street"/>
<result property="city" column="city"/>
<result property="state" column="state"/>
<result property="zip" column="zip"/>
<result property="country" column="country"/>
</resultMap>

<resultMap type="Course" id="CourseResult">
<id column="course_id" property="courseId"/>
<result column="name" property="name"/>
<result column="description" property="description"/>
<result column="start_date" property="startDate"/>
<result column="end_date" property="endDate"/>
</resultMap>

<resultMap type="Tutor" id="TutorResult">
<id column="tutor_id" property="tutorId"/>
<result column="tutor_name" property="name"/>
<result column="email" property="email"/>
<association property="address" resultMap="AddressResult"/>
<collection property="courses"  resultMap="CourseResult"/>
</resultMap>
</mapper>

public interface TutorMapper
{
@Select("SELECT T.TUTOR_ID, T.NAME AS TUTOR_NAME, EMAIL,
A.ADDR_ID, STREET, CITY, STATE, ZIP, COUNTRY, COURSE_ID, C.NAME,
DESCRIPTION, START_DATE, END_DATE    FROM TUTORS T LEFT OUTER
JOIN ADDRESSES A ON T.ADDR_ID=A.ADDR_ID LEFT OUTER JOIN COURSES
C ON T.TUTOR_ID=C.TUTOR_ID WHERE T.TUTOR_ID=#{tutorId}")
@ResultMap("com.mybatis3.mappers.TutorMapper.TutorResult")
Tutor selectTutorById(int tutorId);
}
```

Dynamic SQL

Sometimes we may need to build queries dynamically based on input criteria. MyBatis provides various annotations such as `@InsertProvider`, `@UpdateProvider`, `@DeleteProvider`, and `@SelectProvider`, which facilitates building dynamic queries and lets MyBatis execute those queries.

Now let us look at an example of how to create a simple `SELECT` mapped statement using `@SelectProvider`.

Create the `TutorDynaSqlProvider.java` class with the `findTutorByIdSql()` method as follows:

```
package com.mybatis3.sqlproviders;
import org.apache.ibatis.jdbc.SQL;

public class TutorDynaSqlProvider
{
public String findTutorByIdSql(int tutorId)
{
return "SELECT TUTOR_ID AS tutorId, NAME, EMAIL FROM TUTORS
WHERE TUTOR_ID="+tutorId;
}
}
```

Create a mapped statement in the `TutorMapper.java` interface as follows:

```
@SelectProvider(type=TutorDynaSqlProvider.class,
method="findTutorByIdSql")
Tutor findTutorById(int tutorId);
```

Here we have used `@SelectProvider`to specify the class and method name, which provide the SQL statement to be executed.

But constructing SQL queries using string concatenation is difficult and error-prone. So MyBatis provides an `SQL` utility which simplifies building dynamic SQL queries without the need of string concatenations.

Now let us see how we can prepare the same query using the `org.apache.ibatis.jdbc.SQL` utility.

```
package com.mybatis3.sqlproviders;
import org.apache.ibatis.jdbc.SQL;

public class TutorDynaSqlProvider
{
public String findTutorByIdSql(final int tutorId)
```

```
{
return new SQL() {{
SELECT("tutor_id as tutorId, name, email");
FROM("tutors");
WHERE("tutor_id="+tutorId);
}}.toString();
}
}
```

The `SQL` utility will take care of constructing the query with proper space prefix and suffixes if required.

The dynamic SQL provider methods can have one of the following parameters:

- No parameter
- A single parameter with same type of Mapper interface method
- `java.util.Map`

If the SQL query preparation doesn't depend on an input argument, you can use the no-argument SQL provider method.

For example:

```
public String findTutorByIdSql()
{

return new SQL() {{
SELECT("tutor_id as tutorId, name, email");
FROM("tutors");
WHERE("tutor_id = #{tutorId}");
}}.toString();
}
```

Here we are not using any input parameter to construct the query, so it can be a no-argument method.

If the Mapper interface method has only one parameter, we can use a method that has only one parameter of the same type as the SQL provider method.

```
Tutor findTutorById(int tutorId);
```

Here the `findTutorById(int)` method has only one input parameter of type `int`. We can have the `findTutorByIdSql(int)` method as an SQL provider method as follows:

```
public String findTutorByIdSql(final int tutorId)
{

return new SQL() {{
SELECT("tutor_id as tutorId, name, email");
FROM("tutors");
WHERE("tutor_id="+tutorId);
}}.toString();
}
```

If the Mapper interface has multiple input parameters, we can use a method with the `java.util.Map` parameter type as the SQL provider method. Then all the input argument values will be placed in `map` with `param1`, `param2`, and so on as keys, and the input arguments as values. You can also get those input argument values using 0, 1, 2, and so on as keys.

```
@SelectProvider(type=TutorDynaSqlProvider.class,
method="findTutorByNameAndEmailSql")
Tutor findTutorByNameAndEmail(String name, String email);

public String findTutorByNameAndEmailSql(Map<String, Object> map)
{
String name = (String) map.get("param1");
String email = (String) map.get("param2");
//you can also get those values using 0,1 keys
//String name = (String) map.get("0");
//String email = (String) map.get("1");
return new SQL() {{
SELECT("tutor_id as tutorId, name, email");
FROM("tutors");
WHERE("name=#{name} AND email=#{email}");
}}.toString();
}
```

The `SQL` utility also provides various other methods to perform `JOINS`, `ORDER_BY`, `GROUP_BY`, and so on.

Let us look at an example of using `LEFT_OUTER_JOIN`:

```
public class TutorDynaSqlProvider
{
public String selectTutorById()
```

```
{

return new SQL() {{
SELECT("t.tutor_id, t.name as tutor_name, email");
SELECT("a.addr_id, street, city, state, zip, country");
SELECT("course_id, c.name as course_name, description,
start_date, end_date");
FROM("TUTORS t");
LEFT_OUTER_JOIN("addresses a on t.addr_id=a.addr_id");
LEFT_OUTER_JOIN("courses c on t.tutor_id=c.tutor_id");
WHERE("t.TUTOR_ID = #{id}");
}}.toString();
}
}

public interface TutorMapper
{
@SelectProvider(type=TutorDynaSqlProvider.class,
method="selectTutorById")
@ResultMap("com.mybatis3.mappers.TutorMapper.TutorResult")
Tutor selectTutorById(int tutorId);
}
```

As there is no annotation support for mapping one-to-many results using nested`ResultMap`, we can use the XML-based `<resultMap>` configuration and map with `@ResultMap`.

```
<mapper namespace="com.mybatis3.mappers.TutorMapper">

<resultMap type="Address" id="AddressResult">
<id property="id" column="addr_id"/>
<result property="street" column="street"/>
<result property="city" column="city"/>
<result property="state" column="state"/>
<result property="zip" column="zip"/>
<result property="country" column="country"/>
</resultMap>

<resultMap type="Course" id="CourseResult">
<id column="course_id" property="id"/>
<result column="course_name" property="name"/>
<result column="description" property="description"/>
<result column="start_date" property="startDate"/>
<result column="end_date" property="endDate"/>
</resultMap>
```

```
<resultMap type="Tutor" id="TutorResult">
<id column="tutor_id" property="id"/>
<result column="tutor_name" property="name"/>
<result column="email" property="email"/>
<association property="address" resultMap="AddressResult"/>
<collection property="courses"
resultMap="CourseResult"></collection>
</resultMap>
</mapper>
```

With this dynamic SQL provider we can fetch `Tutor` details along with `Address` and `Courses details`.

@InsertProvider

We can create dynamic `INSERT` queries using `@InsertProvider` as follows:

```
public class TutorDynaSqlProvider
{
public String insertTutor(final Tutor tutor)
{

return new SQL() {{
INSERT_INTO("TUTORS");

if (tutor.getName() != null) {
VALUES("NAME", "#{name}");
}

if (tutor.getEmail() != null) {
VALUES("EMAIL", "#{email}");
}
}}.toString();
}
}

public interface TutorMapper
{
@InsertProvider(type=TutorDynaSqlProvider.class,
method="insertTutor")
@Options(useGeneratedKeys=true, keyProperty="tutorId")
int insertTutor(Tutor tutor);
}
```

@UpdateProvider

We can create dynamic `UPDATE` queries using `@UpdateProvider` as follows:

```
public class TutorDynaSqlProvider
{
public String updateTutor(final Tutor tutor)
{

return new SQL() {{
UPDATE("TUTORS");

if (tutor.getName() != null) {
SET("NAME = #{name}");
}

if (tutor.getEmail() != null) {
SET("EMAIL = #{email}");
}
WHERE("TUTOR_ID = #{tutorId}");
}}.toString();
}
}

public interface TutorMapper
{
@UpdateProvider(type=TutorDynaSqlProvider.class,
method="updateTutor")
int updateTutor(Tutor tutor);
}
```

@DeleteProvider

We can create dynamic `DELETE` queries using `@DeleteProvider` as follows:

```
public class TutorDynaSqlProvider
{
public String deleteTutor(int tutorId)
{

return new SQL() {{
DELETE_FROM("TUTORS");
WHERE("TUTOR_ID = #{tutorId}");
}}.toString();
}
```

```
}

public interface TutorMapper
{
@DeleteProvider(type=TutorDynaSqlProvider.class,
method="deleteTutor")
int deleteTutor(int tutorId);
}
```

Summary

In this chapter, we learned how to write SQL mapped statements using annotations. We discussed how to configure simple statements, statements with one-to-one and one-to-many relationships. We also looked into building dynamic queries using `SqlProvider` annotations. In the next chapter, we will discuss how to integrate MyBatis with the Spring framework.

5
Integration with Spring

MyBatis-Spring is a submodule of the MyBatis framework, which provides seamless integration with the popular dependency injection framework, Spring.

The Spring framework is a **Dependency Injection** and **Aspect Oriented Programming (AOP)** based Java application framework which encourages POJO-based programming model. Also, Spring provides declarative and programmatic transaction management capabilities, which greatly simplify the implementation of the **data access layer** of the application. In this chapter, we will see how to use MyBatis in a Spring-based application and use Spring's annotation-based transaction management strategy.

In this chapter we will cover the following topics:

- Configuring MyBatis in a Spring application
    - Installation
    - Configuring MyBatis beans
- Working with SqlSession
- Working with mappers
- Transaction management using Spring

Configuring MyBatis in a Spring application

This section describes how to install and configure MyBatis in a Spring-based application.

Installation

If you are using the Maven build tool, you can configure Mybatis' spring dependency as follows:

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
<version>1.2.0</version>

</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>3.1.3.RELEASE</version>
  <exclusions>
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>3.1.3.RELEASE</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.6.8</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.6.8</version>
</dependency>
```

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib-nodep</artifactId>
  <version>2.2</version>
</dependency>

<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.4</version>
</dependency>
```

If you are not using Maven, you can download `mybatis-spring-1.2.0-bundle.zip` from `http://code.google.com/p/mybatis/`. Extract and add `mybatis-spring-1.2.0.jar` to the classpath.

You can download the Spring framework bundle `spring-framework-3.1.3.RELEASE.zip` from `http://www.springsource.org/download/community/` and add Spring and its dependent JAR files to the classpath.

If we are using MyBatis without Spring then we need to create the `SqlSessionFactory` object by ourselves and need to create a `SqlSession` object from `SqlSessionFactory` in every method. Also, we are responsible for committing or rolling back the transaction and closing the `SqlSession` object.

By using the MyBatis-Spring module, we can configure the MyBatis beans in Spring ApplicationContext, and Spring will take care of instantiating the `SqlSessionFactory` object and create a `SqlSession` object and inject it into the `DAO` or `Service` classes. Also, you can use Spring's annotation-based transaction management capabilities, without writing transaction-handling code inside the data access logic.

Configuring MyBatis beans

To let Spring instantiate MyBatis components such as `SqlSessionFactory`, `SqlSession`, and the `Mapper` objects, we need to configure them in the Spring bean definition file, say `applicationContext.xml`, as follows:

```
<beans>
  <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/elearning"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
    </bean>

    <bean id="sqlSessionFactory"
    class="org.mybatis.spring.SqlSessionFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="typeAliases"
      value="com.mybatis3.domain.Student,
      com.mybatis3.domain.Tutor"/>
      <property name="typeAliasesPackage"
      value="com.mybatis3.domain"/>
      <property name="typeHandlers"
      value="com.mybatis3.typehandlers.PhoneTypeHandler"/>
      <property name="typeHandlersPackage"
      value="com.mybatis3.typehandlers"/>
      <property name="mapperLocations"
      value="classpath*:com/mybatis3/**/*.xml" />
      <property name="configLocation" value="WEB-INF/mybatis-
      config.xml"/>
    </bean>
  </beans>
```

With the preceding bean definitions, Spring will create a `SqlSessionFactory` object using the configured properties as follows:

- `dataSource`: It refers to the `dataSource` bean
- `typeAliases`: It specifies the comma-separated list of fully qualified class names for which aliases should be created using the default aliasing rule
- `typeAliasesPackage`: It specifies the comma-separated list of package names that needs to be scanned and creates aliases for JavaBeans
- `typeHandlers`: It specifies the comma-separated list of fully qualified class names of the type handler classes
- `typeHandlersPackage`: It specifies the comma-separated list of package names, which needs to scanned for the type handler classes
- `mapperLocations`: It specifies the location of the SQL Mapper XML files
- `configLocation`: It specifies the location of the MyBatis `SqlSessionFactory` config file

Working with SqlSession

Once the `SqlSessionFactory` bean is configured, we need to configure the `SqlSessionTemplate` bean, which is a thread-safe Spring bean from which we can obtain the thread-safe `SqlSession` objects. Because `SqlSessionTemplate` provides the thread-safe `SqlSession` objects, you can share the same `SqlSessionTemplate` instance with multiple Spring beans. Conceptually `SqlSessionTemplate` is similar to `JdbcTemplate` of the Spring DAO module.

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
  <constructor-arg index="0" ref="sqlSessionFactory" />
</bean>
```

Now we can inject the `SqlSession` bean into any Spring bean and use the `SqlSession` object to invoke the mapped SQL statements.

```
public class StudentDaoImpl implements StudentDao
{
  private SqlSession sqlSession;
  public void setSqlSession(SqlSession session)
  {
    this.sqlSession = session;
  }
  public void createStudent(Student student)
  {
    StudentMapper mapper =
    this.sqlSession.getMapper(StudentMapper.class);
    mapper.insertStudent(student);
  }
}
```

If you are using an XML-based configuration for configuring the Spring beans, you can inject the `SqlSession` bean into the `StudentDaoImpl` beans as follows:

```
<bean id="studentDao" class="com.mybatis3.dao.StudentDaoImpl">
  <property name="sqlSession" ref="sqlSession" />
</bean>
```

If you are using the annotation-based configuration for wiring the Spring beans, you can inject the `SqlSession` bean into the `StudentDaoImpl` beans as follows:

```
@Repository
public class StudentDaoImpl implements StudentDao
{
  private SqlSession sqlSession;
  @Autowired
```

```
  public void setSqlSession(SqlSession session)
  {
    this.sqlSession = session;
  }
  public void createStudent(Student student)
  {
    StudentMapper mapper =
    this.sqlSession.getMapper(StudentMapper.class);
    mapper.insertStudent(student);
  }
}
```

There is another way to inject the `SqlSession` object, that is, by extending `SqlSessionDaoSupport`. This approach enables us to perform any custom logic in addition to executing the mapped statements.

```
public class StudentMapperImpl extends SqlSessionDaoSupport implements
StudentMapper
{
  public void createStudent(Student student)
  {
    StudentMapper mapper =
    getSqlSession().getMapper(StudentMapper.class);
    mapper.insertAddress(student.getAddress());
    //Custom logic
    mapper.insertStudent(student);
  }
}

<bean id="studentMapper" class="com.mybatis3.dao.StudentMapperImpl">
  <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

In these approaches we are injecting the `SqlSession` object, getting the Mapper instance, and executing the mapped statements. Here, Spring will take care of providing a thread-safe `SqlSession` object and close `SqlSession` once the method is complete.

However, the MyBatis-Spring module provides a better approach, which we will discuss in the next section, where you can inject the Sql Mapper beans directly instead of getting the mappers from `SqlSession`.

Working with mappers

We can configure the Mapper interface as a Spring bean using `MapperFactoryBean` as follows:

```
public interface StudentMapper
{
  @Select("select stud_id as studId, name, email, phone from
  students where stud_id=#{id}")
  Student findStudentById(Integer id);
}

<bean id="studentMapper" class="org.mybatis.spring.mapper.
MapperFactoryBean">
<property name="mapperInterface" value="com.mybatis3.mappers.
StudentMapper" />
<property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

Now the `StudentMapper` bean can be injected into any Spring bean and can invoke the mapped statement methods as follows:

```
public class StudentService
{
  private StudentMapper studentMapper;
  public void setStudentMapper (StudentMapperstudentMapper)
  {
    this. studentMapper = studentMapper;
  }
  public void createStudent(Student student)
  {
    this.studentMapper.insertStudent(student);
  }
}

<bean id="studentService" class="com.mybatis3.services.
StudentService">
  <property name="studentMapper" ref="studentMapper" />
</bean>
```

Configuring each Mapper interface individually is a tedious process. Instead of configuring each mapper separately, we can use `MapperScannerConfigurer` to scan packages for the Mapper interfaces and register them automatically.

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="basePackage" value="com.mybatis3.mappers" />
</bean>
```

If the Mapper interfaces are in different packages, you can specify a comma-separated list of package names for the `basePackage` attribute.

MyBatis-Spring-1.2.0 introduced two new ways for scanning the Mapper interfaces:

- Using the `<mybatis:scan/>` element
- Using the `@MapperScan` annotation (requires Spring 3.1+)

<mybatis:scan/>

The `<mybatis:scan>` element will search for Mapper interfaces in the specified comma-separated list of package names. To use this new MyBatis-Spring namespace you need to add the following schema declarations:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://mybatis.org/schema/mybatis-spring
  http://mybatis.org/schema/mybatis-spring.xsd">
  <mybatis:scan base-package="com.mybatis3.mappers" />
</beans>
```

The `<mybatis:scan>` element provides the following attributes, which can be used to customize the scanning process:

- `annotation`: The scanner will register all the interfaces in the base package that also have the specified interface class as a parent.
- `factory-ref`: Specifies which `SqlSessionFactory` to use in case there is more than one in the spring context. Usually this is only needed when you have more than one datasource.
- `marker-interface`: The scanner will register all the interfaces in the base package that also have the specified annotation.

- `template-ref`: Specifies which `SqlSessionTemplate` to use in case there is more than one in the spring context. Usually this is only needed when you have more than one datasource.
- `name-generator`: It is the fully-qualified class name of `BeanNameGenerator` to be used for naming the detected components.

@MapperScan

The Spring framework 3.x+ provides Java-based configurations using the `@Configuration` and `@Bean` annotations. If you prefer a Java-based configuration then you can use the `@MapperScan` annotation to scan for Mapper interfaces. `@MapperScan` works in the same way as `<mybatis:scan/>` and also provides all of its customization options as annotation attributes.

```
@Configuration
@MapperScan("com.mybatis3.mappers")
public class AppConfig
{

  @Bean
  public DataSource dataSource() {
    return new PooledDataSource("com.mysql.jdbc.Driver",
    "jdbc:mysql://localhost:3306/elearning", "root", "admin");
  }

  @Bean
  public SqlSessionFactory sqlSessionFactory() throws Exception {
  SqlSessionFactoryBeansessionFactory = new
  SqlSessionFactoryBean();
  sessionFactory.setDataSource(dataSource());
  return sessionFactory.getObject();
  }
}
```

The `@MapperScan` annotation has the following attributes for customizing the scanning process:

- `annotationClass`: The scanner will register all the interfaces in the base package that also have the specified annotation.
- `markerInterface`: The scanner will register all the interfaces in the base package that also have the specified interface class as a parent.
- `sqlSessionFactoryRef`: Specifies which `SqlSessionFactory` to use in case there is more than one in the spring context.

- `sqlSessionTemplateRef`: Specifies which `SqlSessionTemplate` to use in case there is more than one in the spring context.
- `nameGenerator`: The `BeanNameGenerator` class is to be used for naming the detected components within the Spring container.
- `basePackageClasses`: A type-safe alternative to `basePackages()` for specifying the packages to scan for annotated components. The package of each class specified will be scanned.
- `basePackages`: Base packages to scan for MyBatis interfaces. Note that only interfaces with at least one method will be registered; concrete classes will be ignored.

Injecting mappers is a preferred approach for injecting `SqlSession` beans because it removes the dependency on MyBatis API from the Java code.

Transaction management using Spring

Using plain MyBatis, you need to write the code for transaction handling, such as committing or rolling back the database operations.

```
public Student createStudent(Student student)
{
  SqlSession sqlSession = MyBatisUtil.getSqlSessionFactory().
openSession();
  try {
    StudentMapper mapper =
    sqlSession.getMapper(StudentMapper.class);
    mapper.insertAddress(student.getAddress());
    mapper.insertStudent(student);
    sqlSession.commit();
    return student;
  }
  catch (Exception e) {
    sqlSession.rollback();
    throw new RuntimeException(e);
  }
  finally {
    sqlSession.close();
  }
}
```

Instead of writing this boiler plate for each method, we can use the Spring's annotation-based transaction-handling mechanism.

To be able to utilize the Spring's transaction management capabilities, we should configure the `TransactionManager` bean in Spring application context.

```
<bean id="transactionManager" class="org.springframework.jdbc.
datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

The `dataSource` reference for the transaction manager should be the same `dataSource`, which is used for the `SqlSessionFactory` bean.

Enable the annotation-based transaction management feature in Spring as follows:

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Now you can annotate the Spring service beans with the `@Transactional` annotation, indicating that each method in this service should run within a transaction. Spring will commit the operation if the method is completed successfully and will rollback if any runtime exception occurs. Also, Spring will take care of converting MyBatis `Exceptions` into appropriate `DataAccessExceptions`, thereby providing additional details on specific error conditions.

```
@Service
@Transactional
public class StudentService
{
  @Autowired
  private StudentMapper studentMapper;

  public Student createStudent(Student student)
  {
    studentMapper.insertAddress(student.getAddress());
    if(student.getName().equalsIgnoreCase("")){
      throw new RuntimeException("Student name should not be
      empty.");
    }
    studentMapper.insertStudent(student);

    return student;
  }
}
```

The following is the complete configuration of Spring's `applicationContext.xml`:

```
<beans>

  <context:annotation-config />

  <context:component-scan base-package="com.mybatis3" />

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

<tx:annotation-driven transaction-manager="transactionManager"/>

  <bean id="transactionManager"
  class="org.springframework.jdbc.datasource.
  DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="basePackage" value="com.mybatis3.mappers" />
  </bean>

  <bean id="sqlSession"
  class="org.mybatis.spring.SqlSessionTemplate">
  <constructor-arg index="0" ref="sqlSessionFactory" />
  </bean>

  <bean id="sqlSessionFactory"
  class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="typeAliases"
    value="com.mybatis3.domain.Student,
    com.mybatis3.domain.Tutor"/>
    <property name="typeAliasesPackage"
    value="com.mybatis3.domain"/>
    <property name="typeHandlers"
    value="com.mybatis3.typehandlers.PhoneTypeHandler"/>
    <property name="typeHandlersPackage"
    value="com.mybatis3.typehandlers"/>
    <property name="mapperLocations"
    value="classpath*:com/mybatis3/**/*.xml" />
  </bean>

<bean id="dataSource"
class="org.springframework.jdbc.datasource.
```

```
DriverManagerDataSource">
    <property name="driverClassName"
    value="${jdbc.driverClassName}"></property>
    <property name="url" value="${jdbc.url}"></property>
    <property name="username" value="${jdbc.username}"></property>
    <property name="password" value="${jdbc.password}"></property>
  </bean>

</beans>
```

Now let us write a standalone test client for testing `StudentService` as follows:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:applicationContext.xml"
)
public class StudentServiceTest
{
  @Autowired
  private StudentService studentService;

  @Test
  public void testCreateStudent() {
    Address address = new Address(0,"Quaker Ridge
    Rd.","Bethel","Brooklyn","06801","USA");

    Student stud = new Student();
    long ts = System.currentTimeMillis();
    stud.setName("stud_"+ts);
    stud.setEmail("stud_"+ts+"@gmail.com");
    stud.setAddress(address);
    Student student = studentService.createStudent(stud);
    assertNotNull(student);
    assertEquals("stud_"+ts, student.getName());
    assertEquals("stud_"+ts+"@gmail.com", student.getEmail());
    System.err.println("CreatedStudent: "+student);
  }

  @Test(expected=DataAccessException.class)
  public void testCreateStudentForException() {
    Address address = new Address(0,"Quaker Ridge
    Rd.","Bethel","Brooklyn","06801","USA");

    Student stud = new Student();
    long ts = System.currentTimeMillis();
    stud.setName("Timothy");
```

```
        stud.setEmail("stud_"+ts+"@gmail.com");
        stud.setAddress(address);
        studentService.createStudent(stud);
        fail("You should not reach here");
      }
    }
```

Here in the `testCreateStudent()` method, we are giving proper data, so both the `Address` and `Student` records should be inserted into the `ADDRESSES` and `STUDENTS` tables. In the `testCreateStudentForException()` method we are setting `Name` to `Timothy`, which is already in the database. So when you try to insert this student record into the database, MySQL throws an exception as there is a `UNIQUE KEY` constraint on the `NAME` column. Spring will convert that exception into `DataAccessException` and the record inserted into the `ADDRESSES` table will also be rolled back.

Summary

In this chapter we learned how to integrate MyBatis with the Spring framework. We also learned how to install the Spring libraries and register the MyBatis beans in Spring ApplicationContext. We saw how to configure and inject the SqlSession and Mapper beans and invoke the mapped statements. We also learned how to utilize Spring's annotation-based transaction-handling mechanism with MyBatis.

You have finished reading this book, congratulations! By now, you should know how to use MyBatis effectively to work with databases. You learned how to be more productive with MyBatis by leveraging your Java and SQL skills. You know how to write database persistence code using MyBatis in a much cleaner way, leaving all the low-level details to be handled by the MyBatis framework. In addition, you learned how to use MyBatis with the most popular dependency injection framework, Spring.

The MyBatis framework is very easy to work with, yet it provides powerful features, thereby making it a good database persistence solution for the Java-based projects. MyBatis also provides tools such as MyBatis Generator (`http://www.mybatis.org/generator/`), which can be used to generate the persistence code artifacts such as database entities, Mapper interfaces, and Mapper XML files, from an existing database schema, which comes in very handy to start with MyBatis. Also, MyBatis has sister projects such as **MyBatis.NET** and **MyBatis-Scala**, providing the same powerful features for the **.NET** and **Scala** programming languages.

MyBatis is getting better and better with each release, with lots of new features. To learn more about these new features, you can visit the official MyBatis website at `https://code.google.com/p/mybatis/`. It is a good idea to subscribe to the MyBatis user mailing list. We wish you all the best, and happy coding!

Index

Symbols

A

B

C

D

E

F

H

I

J

K

M

N

O

P

R

S

T

U

W

Thank you for buying

Java Persistence with MyBatis 3

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

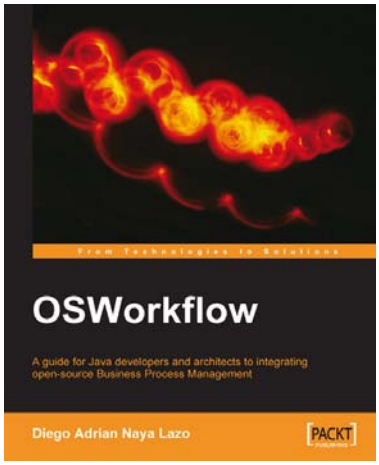
Spring Persistence with Hibernate

ISBN: 978-1-849510-56-1 Paperback: 460 pages

Build robust and reliable persistence solutions for your enterprise Java application

1. Get to grips with Hibernate and its configuration manager, mappings, types, session APIs, queries, and much more
2. Integrate Hibernate and Spring as part of your enterprise Java stack development
3. Work with Spring IoC (Inversion of Control), Spring AOP, transaction management, web development, and unit testing considerations and features

---



OSWorkflow

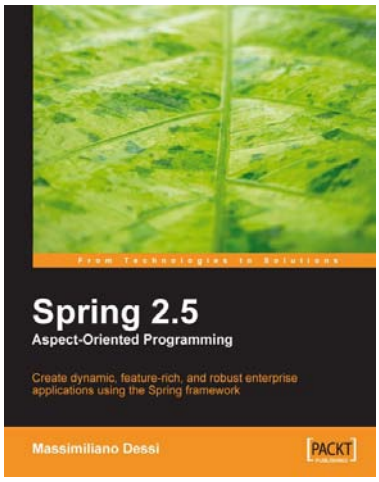
ISBN: 978-1-84719-152-6 Paperback: 212 pages

A guide for Java developers and architects to integrating open-source Business Process Management

1. Basics of OSWorkflow
2. Integrating business rules with Drools
3. Task scheduling with Quartz

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING



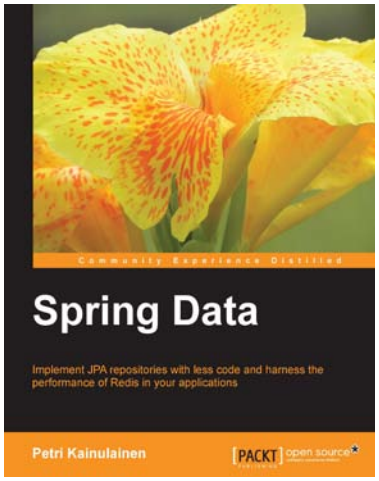
Spring 2.5 Aspect Oriented Programming

ISBN: 978-1-84719-402-2 Paperback: 332 pages

Create dynamic, feature-rich, and robust enterprise applications using the Spring framework

1. Master Aspect-Oriented Programming and its solutions to implementation issues in Object-Oriented Programming
2. A practical, hands-on book for Java developers rich with code, clear explanations, and interesting examples
3. Includes Domain-Driven Design and Test-Driven Development of an example online shop using AOP in a three-tier Spring application

---



Spring Data

ISBN: 978-1-84951-904-5 Paperback: 160 pages

Implement JPA repositories and harness the performance of Redis in your applications

1. Implement JPA repositories with lesser code
2. Includes functional sample projects that demonstrate the described concepts in action and help you start experimenting right away
3. Provides step-by-step instructions and a lot of code examples that are easy to follow and help you to get started from page one

Please check **www.PacktPub.com** for information on our titles