iBATIS Tutorial

IBATIS TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

iBATIS Tutorial

iBATIS is a persistence framework which automates the mapping between SQL databases and objects in Java, .NET, and Ruby on Rails. iBATIS makes it easier to build better database oriented application more quickly and with less code.

Audience

This tutorial is designed for Java programmers with a need to understand the iBATIS framework in detail along with its architecture and actual usage. This tutorial will bring you at intermediate level of expertise from where you can take yourself at higher level of expertise.

Prerequisites

Before proceeding with this tutorial you should have a good understanding of Java programming language. Because you are going to deal with SQL mapping, so it is required that you have good understanding on SQL and Database concepts.

Copyright & Disclaimer Notice

Table of Content

CHAPTER

1

iBATIS Overview

IBATIS is a persistence framework which automates the mapping between SQL databases and objects in Java, .NET, and Ruby on Rails. The mappings are decoupled from the application logic by packaging the SQL statements in XML configuration files.

iBATIS is a lightweight framework and persistence API good for persisting POJOs(Plain Old Java Objects).

iBATIS is what is known as a data mapper and takes care of mapping the parameters and results between the class properties and the columns of the database table.

A significant difference between iBATIS and other persistence frameworks such as Hibernate is that iBATIS emphasizes use of SQL, while other frameworks typically use a custom query language such has the Hibernate Query Language (HQL) or Enterprise JavaBeans Query Language (EJB QL).

iBATIS Design Philosophies:

iBATIS comes with the following design philosophies:

- **Simplicity:** iBATIS is widely regarded as being one of the simplest persistence frameworks available today.
- **Fast Development:** iBATIS's philosophy is to do all it can to facilitate hyper-fast development.
- **Portability:** iBATIS can be implemented for nearly any language or platform like Java, Ruby, and C# for Microsoft .NET.
- **Independent Interfaces:** iBATIS provides database-independent interfaces and APIs that help the rest of the application remain independent of any persistence-related resources,
- **Open source:** iBATIS is free and an open source software.

Advantages of IBATIS

Here are few advantages of using IBATIS:

- **Suppports Stored procedures:** iBATIS encapsulates SQL in the form of stored procedures so that business logic is kept out of the database, and the application is easier to deploy and test, and is more portable.
- **Supports Inline SQL:** No precompiler is needed, and you have full access to all of the features of SQL.
- **Supports Dynamic SQL:** iBATIS provides features for dynamically building SQL queries based on parameters.

- **Supports O/RM:** iBATIS supports many of the same features as an O/RM tool, such as lazy loading, join fetching, caching, runtime code generation, and inheritance

Pre-Requisite:

Before you proceed, you make sure that you understand the basics of procedural and object-oriented programming: control structures, data structures and variables, classes, objects etc.

iBATIS makes use of JAVA programming language while developing database oriented application.

To understand JAVA in detail you can go through our JAVA Tutorial.

CHAPTER

2

iBATIS Environment

Before you start with actual development with iBATIS, you would have to setup your envrionment properly.

This tutorial would guide you with few steps to achieve a working environment.

iBATIS Installation:

Here are the simple steps you would need to carry out to install iBATIS on your Linux machine:

- Download latest version of iBATIS from Download iBATIS.
- Unzip the downloaded file to extract .jar file from the bundle and keep it in appropriate lib directory.
- Set PATH and CLASSPATH variables at the extracted .jar file(s) appropriately.

Here are the steps I carried out on my linux machine after downloading iBATIS binary file:

```
$ unzip ibatis-2.3.4.726.zip
  inflating: META-INF/MANIFEST.MF
   creating: doc/
   creating: lib/
   creating: simple_example/
   creating: simple_example/com/
   creating: simple_example/com/mydomain/
   creating: simple_example/com/mydomain/data/
   creating: simple_example/com/mydomain/domain/
   creating: src/
  inflating: doc/dev-javadoc.zip
  inflating: doc/user-javadoc.zip
  inflating: jar-dependencies.txt
  inflating: lib/ibatis-2.3.4.726.jar
  inflating: license.txt
  inflating: notice.txt
  inflating: release.txt
$pwd
/var/home/ibatis
$set PATH=$PATH:/var/home/ibatis/
$set CLASSPATH=$CLASSPATH:/var/home/ibatis\
                       /lib/ibatis-2.3.4.726.jar
```

Database Setup:

Create EMPLOYEE table in any MySQL database using the following syntax:

```
mysql> CREATE TABLE EMPLOYEE (
            id INT NOT NULL auto_increment,
            first_name VARCHAR(20) default NULL,
            last_name  VARCHAR(20) default NULL,
            salary     INT  default NULL,
            PRIMARY KEY (id)
        );
```

Create SqlMapConfig.xml

Consider the following:

- We are going to use JDBC to access the database **testdb**.
- JDBC driver for MySQL is "com.mysql.jdbc.Driver".
- Connection URL is "jdbc:mysql://localhost:3306/testdb".
- We would use username and password is "root" and "root".
- Our sql statement mappings for all the operations would be described in "Employee.xml".

Based on the above assumption we have to create an XML configuration file with name**SqlMapConfig.xml** with the following content. This is where you need to provide all configurations required for iBatis:

It is important that both the files SqlMapConfig.xml and Employee.xml should be present in the class path. For now we would keep Employee.xml file empty and we would conver its content in subsequent chapters.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
     <settings useStatementNamespaces="true"/>
     <transactionManager type="JDBC">
        <dataSource type="SIMPLE">
          <property name="JDBC.Driver"
               value="com.mysql.jdbc.Driver"/>
          <property name="JDBC.ConnectionURL"
               value="jdbc:mysql://localhost:3306/testdb"/>
          <property name="JDBC.Username" value="root"/>
          <property name="JDBC.Password" value="root"/>
        </dataSource>
      </transactionManager>
     <sqlMap resource="Employee.xml"/>
</sqlMapConfig>
```

There are other optional properties which you can set using SqlMapConfig.xml file:

```
<property name="JDBC.AutoCommit" value="true"/>

<property name="Pool.MaximumActiveConnections" value="10"/>
```

```
<property name="Pool.MaximumIdleConnections" value="5"/>

<property name="Pool.MaximumCheckoutTime" value="150000"/>

<property name="Pool.MaximumTimeToWait" value="500"/>

<property name="Pool.PingQuery" value="select 1 from Employee"/>

<property name="Pool.PingEnabled" value="false"/>
```

CHAPTER

3

iBATIS Create Operation

To perform any CRUD (Create, Write, Update and Delete) operation using iBATIS, you would need to create a POJOs (Plain Old Java Objects) class corresponding to the table. This class describes the objects that will "model" database table rows.

The POJO class would have implementation for all the methods required to perform desired operations.

We have following EMPLOYEE table in MySQL:

```
CREATE TABLE EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Employee POJO Class:

We would create Employee class in Employee.java file as follows:

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;

  /* Define constructors for the Employee class. */
  public Employee() {}

  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
  }
} /* End of Employee */
```

You can define methods to set individual fields in the table. Next chapter would show you how to get value of individual fields.

Employee.xml File:

To define SQL mapping statement using iBATIS, we would use <insert> tag and inside this tag definition we would define an "id" which will be used in IbatisInsert.java file for executing SQL INSERT query on database.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">
<insert id="insert" parameterClass="Employee">

   insert into EMPLOYEE(first_name, last_name, salary)
   values (#first_name#, #last_name#, #salary#)

   <selectKey resultClass="int" keyProperty="id">
      select last_insert_id() as id
   </selectKey>

</insert>

</sqlMap>
```

Here **parameterClass:** could take a value as *string, int, float, double* or any class *object* based on requirement. In this example we would pass Employee object as a parameter while calling *insert*method of SqlMap class.

If your database table uses an IDENTITY, AUTO_INCREMENT, or SERIAL column or you have defined a SEQUENCE/GENERATOR, you can use the <selectKey> element in an <insert> statement to use or return that database-generated value.

IbatisInsert.java File:

This file would have application level logic to insert records in the Employee table:

```
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisInsert{
  public static void main(String[] args)
   throws IOException,SQLException{
   Reader rd = Resources.getResourceAsReader("SqlMapConfig.xml");
   SqlMapClient smc = SqlMapClientBuilder.buildSqlMapClient(rd);

   /* This would insert one record in Employee table. */
   System.out.println("Going to insert record.....");
   Employee em = new Employee("Zara", "Ali", 5000);

   smc.insert("Employee.insert", em);

   System.out.println("Record Inserted Successfully ");

  }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.
- Create Employee.java as shown above and compile it.
- Create IbatisInsert.java as shown above and compile it.
- Execute IbatisInsert binary to run the program.

You would get following result, and a record would be created in EMPLOYEE table.

```
$java IbatisInsert
Going to insert record.....
Record Inserted Successfully
```

Go and check your EMPLOYEE table, it should have following result:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  1 | Zara       | Ali       |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)
```

CHAPTER

4

iBATIS Read Operation

Last chapter has shown how to perform CREATE operation on a table using iBATIS. This chapter would show you how to read a table using iBATIS.

We have following EMPLOYEE table in MySQL:

```
CREATE TABLE EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

This table is having only one record as follows:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  1 | Zara       | Ali       |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)
```

Employee POJO Class:

To perform read operation we would modify Employee class in Employee.java file as follows:

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;

  /* Define constructors for the Employee class. */
  public Employee() {}

  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
  }

  /* Here are the method definitions */
  public int getId() {
    return id;
  }
  public String getFirstName() {
    return first_name;
  }
  public String getLastName() {
    return last_name;
  }
  public int getSalary() {
    return salary;
  }
} /* End of Employee */
```

Employee.xml File:

To define SQL mapping statement using iBATIS, we would add <select> tag in Employee.xml file and inside this tag definition we would define an "id" which will be used in IbatisRead.java file for executing SQL SELECT query on database.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">
<insert id="insert" parameterClass="Employee">
   INSERT INTO EMPLOYEE(first_name, last_name, salary)
   values (#first_name#, #last_name#, #salary#)

   <selectKey resultClass="int" keyProperty="id">
      select last_insert_id() as id
   </selectKey>

</insert>
<select id="getAll" resultClass="Employee">
   SELECT * FROM EMPLOYEE
</select>
</sqlMap>
```

Here we did not use WHERE clause with SQL SELECT statement. Subsequent chapter would demonstrate how you can use WHERE clause with SELECT statement and how you can pass values to that WHERE clause.

IbatisRead.java File:

This file would have application level logic to read records from the Employee table:

```
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisRead{
  public static void main(String[] args)
```

```
    throws IOException,SQLException{
    Reader rd = Resources.getResourceAsReader("SqlMapConfig.xml");
    SqlMapClient smc = SqlMapClientBuilder.buildSqlMapClient(rd);

    /* This would read all records from the Employee table. */
    System.out.println("Going to read records.....");
    List <Employee> ems = (List<Employee>)
                          smc.queryForList("Employee.getAll", null);
    Employee em = null;
    for (Employee e : ems) {
       System.out.print("  " + e.getId());
       System.out.print("  " + e.getFirstName());
       System.out.print("  " + e.getLastName());
       System.out.print("  " + e.getSalary());
       em = e;
       System.out.println("");
    }

    System.out.println("Records Read Successfully ");

   }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.
- Create Employee.java as shown above and compile it.
- Create IbatisRead.java as shown above and compile it.
- Execute IbatisRead binary to run the program.

You would get following result, and a record would be read from the EMPLOYEE table.

```
Going to read records.....
   1  Zara  Ali  5000
Record Reads Successfully
```

CHAPTER

5

iBATIS Update Operation

Last chapter has shown how to perform READ operation on a table using iBATIS. This chapter would show you how you can update records in a table using iBATIS.

We have following EMPLOYEE table in MySQL:

```
CREATE TABLE EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

This table is having only one record as follows:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  1 | Zara       | Ali       |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)
```

Employee POJO Class:

To perform udpate operation you would need to modify Employee.java file as follows:

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;

  /* Define constructors for the Employee class. */
  public Employee() {}

  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
  }

 /* Here are the required method definitions */
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getFirstName() {
    return first_name;
  }
  public void setFirstName(String fname) {
    this.first_name = fname;
  }
  public String getLastName() {
    return last_name;
  }
  public void setlastName(String lname) {
    this.last_name = lname;
  }
  public int getSalary() {
    return salary;
  }
  public void setSalary(int salary) {
    this.salary = salary;
  }

 } /* End of Employee */
```

Employee.xml File:

To define SQL mapping statement using iBATIS, we would add <update> tag in Employee.xml file and inside this tag definition we would define an "id" which will be used in IbatisUpdate.java file for executing SQL UPDATE query on database.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">
<insert id="insert" parameterClass="Employee">
   INSERT INTO EMPLOYEE(first_name, last_name, salary)
   values (#first_name#, #last_name#, #salary#)

   <selectKey resultClass="int" keyProperty="id">
      select last_insert_id() as id
   </selectKey>

</insert>

<select id="getAll" resultClass="Employee">
   SELECT * FROM EMPLOYEE
</select>

<update id="update" parameterClass="Employee">
   UPDATE EMPLOYEE
   SET    first_name = #first_name#
   WHERE  id = #id#
```

```
</update>
</sqlMap>
```

IbatisUpdate.java File:

This file would have application level logic to update records into the Employee table:

```
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisUpdate{
  public static void main(String[] args)
   throws IOException,SQLException{
   Reader rd = Resources.getResourceAsReader("SqlMapConfig.xml");
   SqlMapClient smc = SqlMapClientBuilder.buildSqlMapClient(rd);

   /* This would update one record in Employee table. */
   System.out.println("Going to update record.....");
   Employee rec = new Employee();
   rec.setId(1);
   rec.setFirstName( "Roma");
   smc.update("Employee.update", rec );
   System.out.println("Record updated Successfully ");

   System.out.println("Going to read records.....");
   List <Employee> ems = (List<Employee>)
                  smc.queryForList("Employee.getAll", null);
   Employee em = null;
   for (Employee e : ems) {
      System.out.print("  " + e.getId());
      System.out.print("  " + e.getFirstName());
      System.out.print("  " + e.getLastName());
      System.out.print("  " + e.getSalary());
      em = e;
      System.out.println("");
   }

   System.out.println("Records Read Successfully ");

  }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.
- Create Employee.java as shown above and compile it.
- Create IbatisUpdate.java as shown above and compile it.

- Execute IbatisUpdate binary to run the program.

You would get following result, and a record would be updated in EMPLOYEE table and later same record would be read from the EMPLOYEE table.

```
Going to update record.....
Record updated Successfully
Going to read records.....
  1  Roma  Ali  5000
Records Read Successfully
```

CHAPTER

6

Ibatis Delete Operation

This chapter would teach you how you can delete records from a table using iBATIS.

We have following EMPLOYEE table in MySQL:

```
CREATE TABLE EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Assume this table is having two records as follows:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  1 | Zara       | Ali       |   5000 |
|  2 | Roma       | Ali       |   3000 |
+----+------------+-----------+--------+
2 row in set (0.00 sec)
```

Employee POJO Class:

To perform delete operation you do need to modify Employee.java file. So let us keep it as it is in last chapter.

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;

  /* Define constructors for the Employee class. */
  public Employee() {}

  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
```

```
   }

 /* Here are the required method definitions */
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getFirstName() {
    return first_name;
  }
  public void setFirstName(String fname) {
    this.first_name = fname;
  }
  public String getLastName() {
    return last_name;
  }
  public void setlastName(String lname) {
    this.last_name = lname;
  }
  public int getSalary() {
    return salary;
  }
  public void setSalary(int salary) {
    this.salary = salary;
  }

 } /* End of Employee */
```

Employee.xml File:

To define SQL mapping statement using iBATIS, we would add <delete> tag in Employee.xml file and inside this tag definition we would define an "id" which will be used in IbatisDelete.java file for executing SQL DELETE query on database.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">
<insert id="insert" parameterClass="Employee">
   INSERT INTO EMPLOYEE(first_name, last_name, salary)
   values (#first_name#, #last_name#, #salary#)

   <selectKey resultClass="int" keyProperty="id">
      select last_insert_id() as id
   </selectKey>

</insert>

<select id="getAll" resultClass="Employee">
   SELECT * FROM EMPLOYEE
</select>

<update id="update" parameterClass="Employee">
   UPDATE EMPLOYEE
   SET    first_name = #first_name#
   WHERE  id = #id#
```

```
</update>

<delete id="delete" parameterClass="int">
   DELETE FROM EMPLOYEE
   WHERE  id = #id#
</delete>

</sqlMap>
```

IbatisDelete.java File:

This file would have application level logic to delete records from the Employee table:

```
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisDelete{
  public static void main(String[] args)
   throws IOException,SQLException{
   Reader rd = Resources.getResourceAsReader("SqlMapConfig.xml");
   SqlMapClient smc = SqlMapClientBuilder.buildSqlMapClient(rd);

   /* This would delete one record in Employee table. */
   System.out.println("Going to delete record.....");
   int id = 1;

   smc.delete("Employee.delete", id );
   System.out.println("Record deleted Successfully ");

   System.out.println("Going to read records.....");
   List <Employee> ems = (List<Employee>)
                            smc.queryForList("Employee.getAll", null);
   Employee em = null;
   for (Employee e : ems) {
      System.out.print("  " + e.getId());
      System.out.print("  " + e.getFirstName());
      System.out.print("  " + e.getLastName());
      System.out.print("  " + e.getSalary());
      em = e;
      System.out.println("");
   }

   System.out.println("Records Read Successfully ");

  }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.

- Create Employee.java as shown above and compile it.
- Create IbatisDelete.java as shown above and compile it.
- Execute IbatisDelete binary to run the program.

You would get following result, and a record with ID = 1, would be deleted in EMPLOYEE table and rest of the records would be read from the EMPLOYEE table.

```
Going to delete record.....
Record deleted Successfully
Going to read records.....
   2  Roma  Ali  3000
Records Read Successfully
```

CHAPTER

7

iBATIS Result Maps

The resultMap element is the most important and powerful element in iBATIS. You can reduce upto 90% JDBC coding by using iBATIS ResultMap and in some cases allows you to do things that JDBC does not even support.

The design of the ResultMaps is such that simple statements don't require explicit result mappings at all, and more complex statements require no more than is absolutely necessary to describe the relationships.

This chapter would give you just a simple introduction of iBATIS ResultMap.

We have following EMPLOYEE table in MySQL:

```
CREATE TABLE EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

This table is having two records as follows:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  1 | Zara       | Ali       |   5000 |
|  2 | Roma       | Ali       |   3000 |
+----+------------+-----------+--------+
2 row in set (0.00 sec)
```

Employee POJO Class:

To use iBATIS ResultMap, you do need to modify Employee.java file. So let us keep it as it is in last chapter.

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;
```

```
   /* Define constructors for the Employee class. */
   public Employee() {}

   public Employee(String fname, String lname, int salary) {
     this.first_name = fname;
     this.last_name = lname;
     this.salary = salary;
   }

  /* Here are the required method definitions */
   public int getId() {
     return id;
   }
   public void setId(int id) {
     this.id = id;
   }
   public String getFirstName() {
     return first_name;
   }
   public void setFirstName(String fname) {
     this.first_name = fname;
   }
   public String getLastName() {
     return last_name;
   }
   public void setlastName(String lname) {
     this.last_name = lname;
   }
   public int getSalary() {
     return salary;
   }
   public void setSalary(int salary) {
     this.salary = salary;
   }

  } /* End of Employee */
```

Employee.xml File:

Here we would modify Employee.xml file to introduce <resultMap></resultMap> tag. This tag would have an id which is required to run this resultMap in our <select> tag's resultMap attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">

<!-- Perform Insert Operation -->
<insert id="insert" parameterClass="Employee">
   INSERT INTO EMPLOYEE(first_name, last_name, salary)
   values (#first_name#, #last_name#, #salary#)

   <selectKey resultClass="int" keyProperty="id">
      select last_insert_id() as id
   </selectKey>

</insert>
```

```
<!-- Perform Read Operation -->
<select id="getAll" resultClass="Employee">
   SELECT * FROM EMPLOYEE
</select>

<!-- Perform Update Operation -->
<update id="update" parameterClass="Employee">
   UPDATE EMPLOYEE
   SET    first_name = #first_name#
   WHERE  id = #id#
</update>

<!-- Perform Delete Operation -->
<delete id="delete" parameterClass="int">
   DELETE FROM EMPLOYEE
   WHERE  id = #id#
</delete>

<!-- Using ResultMap -->
 <resultMap id="result" class="Employee">
    <result property="id" column="id"/>
    <result property="first_name" column="first_name"/>
    <result property="last_name" column="last_name"/>
    <result property="salary" column="salary"/>
</resultMap>
<select id="useResultMap" resultMap="result">
         SELECT * FROM EMPLOYEE
         WHERE id=#id#
</select>

</sqlMap>
```

IbatisResultMap.java File:

This file would have application level logic to read records from the Employee table using ResultMap:

```
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisResultMap{
  public static void main(String[] args)
   throws IOException,SQLException{
   Reader rd = Resources.getResourceAsReader("SqlMapConfig.xml");
   SqlMapClient smc = SqlMapClientBuilder.buildSqlMapClient(rd);

   int id = 1;
   System.out.println("Going to read record.....");
   Employee e = (Employee)smc.queryForObject
                 ("Employee.useResultMap", id);

   System.out.println("ID:  " + e.getId());
   System.out.println("First Name:  " + e.getFirstName());
   System.out.println("Last Name:  " + e.getLastName());
   System.out.println("Salary:  " + e.getSalary());

   System.out.println("Record read Successfully ");
   }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.
- Create Employee.java as shown above and compile it.
- Create IbatisResultMap.java as shown above and compile it.
- Execute IbatisResultMap binary to run the program.

You would get following result which is a read operation on the EMPLOYEE table.

```
Going to read record.....
ID:  1
First Name:  Zara
Last Name:  Ali
Salary:  5000
Record read Successfully
```

CHAPTER

8

iBATIS Stored Procedures

This is very much possible to call a stored procedure using iBATIS configuration. To understand this chapter, first you need to understand how we create a stored procedure in MySQL.

Before procedding for this chapter you can go through MySQL Stored Procedure.

We have following EMPLOYEE table in MySQL:

```
CREATE TABLE EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Let us have following stored procedure created in MySQL database.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `testdb`.`getEmp` $$
CREATE PROCEDURE `testdb`.`getEmp`
   (IN empid INT)
BEGIN
   SELECT * FROM EMPLOYEE
   WHERE ID = empid;
END $$

DELIMITER;
```

Consider EMPLOYEE table is having two records as follows:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  1 | Zara       | Ali       |   5000 |
|  2 | Roma       | Ali       |   3000 |
+----+------------+-----------+--------+
2 row in set (0.00 sec)
```

Employee POJO Class:

To use stored procedure, you do need to modify Employee.java file. So let us keep it as it is in last chapter.

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;

  /* Define constructors for the Employee class. */
  public Employee() {}

  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
  }

 /* Here are the required method definitions */
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getFirstName() {
    return first_name;
  }
  public void setFirstName(String fname) {
    this.first_name = fname;
  }
  public String getLastName() {
    return last_name;
  }
  public void setlastName(String lname) {
    this.last_name = lname;
  }
  public int getSalary() {
    return salary;
  }
  public void setSalary(int salary) {
    this.salary = salary;
  }

 } /* End of Employee */
```

Employee.xml File:

Here we would modify Employee.xml file to introduce <procedure></procedure> and <parameterMap></parameterMap> tags. Here <procedure></procedure> tag would have an id which we would use in our application to call the stored procedure.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">
```

```
<!-- Perform Insert Operation -->
<insert id="insert" parameterClass="Employee">
   INSERT INTO EMPLOYEE(first_name, last_name, salary)
   values (#first_name#, #last_name#, #salary#)

   <selectKey resultClass="int" keyProperty="id">
      select last_insert_id() as id
   </selectKey>

</insert>

<!-- Perform Read Operation -->
<select id="getAll" resultClass="Employee">
   SELECT * FROM EMPLOYEE
</select>

<!-- Perform Update Operation -->
<update id="update" parameterClass="Employee">
   UPDATE EMPLOYEE
   SET    first_name = #first_name#
   WHERE  id = #id#
</update>

<!-- Perform Delete Operation -->
<delete id="delete" parameterClass="int">
   DELETE FROM EMPLOYEE
   WHERE  id = #id#
</delete>

<!-- To call stored procedure. -->
<procedure id="getEmpInfo" resultClass="Employee"
              parameterMap="getEmpInfoCall">
   { call getEmp( #acctID# ) }
</procedure>
<parameterMap id="getEmpInfoCall" class="map">
   <parameter property="acctID" jdbcType="INT"
   javaType="java.lang.Integer" mode="IN"/>
</parameterMap>

</sqlMap>
```

IbatisSP.java File:

This file would have application level logic to read name of the employee from the Employee table using ResultMap:

```
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisSP{
  public static void main(String[] args)
   throws IOException,SQLException{
   Reader rd = Resources.getResourceAsReader("SqlMapConfig.xml");
   SqlMapClient smc = SqlMapClientBuilder.buildSqlMapClient(rd);
```

```
    int id = 1;
    System.out.println("Going to read employee name.....");
    Employee e = (Employee)smc.queryForObject
                  ("Employee.getEmpInfo", id);

    System.out.println("First Name:  " + e.getFirstName());

    System.out.println("Record name Successfully ");

   }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.
- Create Employee.java as shown above and compile it.
- Create IbatisSP.java as shown above and compile it.
- Execute IbatisSP binary to run the program.

You would get following result:

```
Going to read record.....
ID:  1
First Name:  Zara
Last Name:  Ali
Salary:  5000
Record read Successfully
```

CHAPTER

9

iBATIS Dynamic SQL

Using dynamic queries is a very powerful feature of iBatis. Sometime you have changing WHERE clause criterion based on your parameter object's state. In such situation iBATIS provides a set of dynamic SQL tags that can be used within mapped statements to enhance the reusability and flexibility of the SQL.

All the logic is put in .XML file using some additional tags. Following is an example where SELECT statement would work in two ways:

- If you would pass an ID then it would return all the records corresponding to that ID,
- otherwise it would return all the records where employee ID is set to NULL.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">
<select id="findByID" resultClass="Employee">
   SELECT * FROM EMPLOYEE
   <dynamic prepend="WHERE ">
      <isNull property="id">
         id IS NULL
      </isNull>
      <isNotNull property="id">
         id = #id#
      </isNotNull>
   </dynamic>
</select>
</sqlMap>
```

You can check condition using <isNotEmpty> tag as follows. Here condition would be added only when passed property is not empty.

```
..................
<select id="findByID" resultClass="Employee">
   SELECT * FROM EMPLOYEE
   <dynamic prepend="WHERE ">
      <isNotEmpty property="id">
           id = #id#
      </isNotEmpty>
   </dynamic>
```

```
</select>
..................
```

If you want a query where we can select on id and/or first name of an Employee. Here would be your SELECT statement:

```
..................
<select id="findByID" resultClass="Employee">
   SELECT * FROM EMPLOYEE
   <dynamic prepend="WHERE ">
      <isNotEmpty prepend="AND" property="id">
           id = #id#
      </isNotEmpty>
      <isNotEmpty prepend="OR" property="first_name">
           first_name = #first_name#
      </isNotEmpty>
   </dynamic>
</select>
..................
```

Example: Dynamic SQL

Following example would show how you can write SELECT statement with dynamic SQL. Consider, we have following EMPLOYEE table in MySQL:

```
CREATE TABLE EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

This table is having only one record as follows:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
|  1 | Zara       | Ali       |   5000 |
|  2 | Roma       | Ali       |   3000 |
|  3 | Noha       | Ali       |   7000 |
+----+------------+-----------+--------+
3 row in set (0.00 sec)
```

Employee POJO Class:

To perform read operation, let us have Employee class in Employee.java file as follows:

```
public class Employee {
  private int id;
  private String first_name;
  private String last_name;
  private int salary;

  /* Define constructors for the Employee class. */
  public Employee() {}

  public Employee(String fname, String lname, int salary) {
    this.first_name = fname;
    this.last_name = lname;
    this.salary = salary;
  }

  /* Here are the method definitions */
  public int getId() {
    return id;
  }
  public String getFirstName() {
    return first_name;
  }
  public String getLastName() {
    return last_name;
  }
  public int getSalary() {
    return salary;
  }
} /* End of Employee */
```

Employee.xml File:

To define SQL mapping statement using iBATIS, we would add following modified <select> tag in Employee.xml file and inside this tag definition we would define an "id" which will be used in IbatisReadDy.java file for executing Dynamic SQL SELECT query on database.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
"http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="Employee">
<select id="findByID" resultClass="Employee">
   SELECT * FROM EMPLOYEE
   <dynamic prepend="WHERE ">
      <isNotNull property="id">
         id = #id#
      </isNotNull>
   </dynamic>
</select>
</sqlMap>
```

Above SELECT statement would work in two ways (i) If you would pass an ID then it would return records corresponding to that ID (ii) otherwise it would return all the records.

IbatisReadDy.java File:

This file would have application level logic to read conditional records from the Employee table:

```
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisReadDy{
  public static void main(String[] args)
   throws IOException,SQLException{
    Reader rd=Resources.getResourceAsReader("SqlMapConfig.xml");
    SqlMapClient smc=SqlMapClientBuilder.buildSqlMapClient(rd);

    /* This would read all records from the Employee table.*/
    System.out.println("Going to read records.....");
    Employee rec = new Employee();
    rec.setId(1);

    List <Employee> ems = (List<Employee>)
                    smc.queryForList("Employee.findByID", rec);
    Employee em = null;
    for (Employee e : ems) {
       System.out.print("  " + e.getId());
       System.out.print("  " + e.getFirstName());
       System.out.print("  " + e.getLastName());
       System.out.print("  " + e.getSalary());
       em = e;
       System.out.println("");
    }

    System.out.println("Records Read Successfully ");

   }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.
- Create Employee.java as shown above and compile it.
- Create IbatisReadDy.java as shown above and compile it.
- Execute IbatisReadDy binary to run the program.

You would get following result, and a record would be read from the EMPLOYEE table.

```
Going to read records.....
  1  Zara  Ali  5000
Record Reads Successfully
```

Try above example by passing **null** as *smc.queryForList("Employee.findByID", null)*.

iBATIS OGNL Expressions

iBATIS provides powerful OGNL based expressions to eliminate most of the other elements.

- if Statement
- choose, when, otherwise Statement
- where Statement

- foreach Statement

The if Statement:

The most common thing to do in dynamic SQL is conditionally include a part of a where clause. For example:

```
<select id="findActiveBlogWithTitleLike"
           parameterType="Blog" resultType="Blog">
   SELECT * FROM BLOG
   WHERE state = 'ACTIVE.
   <if test="title != null">
      AND title like #{title}
   </if>
</select>
```

This statement would provide an optional text search type of functionality. If you passed in no title, then all active Blogs would be returned. But if you do pass in a title, it will look for a title with the given **like**condition.

You can include multiple **if** conditions as follows:

The most common thing to do in dynamic SQL is conditionally include a part of a where clause. For example:

```
<select id="findActiveBlogWithTitleLike"
           parameterType="Blog" resultType="Blog">
   SELECT * FROM BLOG
   WHERE state = 'ACTIVE.
   <if test="title != null">
      AND title like #{title}
   </if>
   <if test="author != null">
      AND author like #{author}
   </if>
</select>
```

The choose, when, otherwise Statement:

iBATIS offers a choose element which is similar to Java's switch statement. This helps choose only one case among many options.

Following example would search only on title if one is provided, then only by author if one is provided. If neither is provided, let's only return featured blogs:

```
<select id="findActiveBlogWithTitleLike"
           parameterType="Blog" resultType="Blog">
   SELECT * FROM BLOG
   WHERE state = 'ACTIVE.
   <choose>
      <when test="title != null">
         AND title like #{title}
      </when>
      <when test="author != null and author.name != null">
         AND author like #{author}
      </when>
      <otherwise>
         AND featured = 1
      </otherwise>
   </choose>
</select>
```

The where Statement:

If we look previous examples, What happens if none of the conditions are met? You would end up with SQL that looked like this:

```
SELECT * FROM BLOG
WHERE
```

This would fail, but iBATIS has a simple solution with one simple change, everything works fine:

```
<select id="findActiveBlogLike"
           parameterType="Blog" resultType="Blog">
   SELECT * FROM BLOG
   <where>
      <if test="state != null">
         state = #{state}
      </if>
      <if test="title != null">
         AND title like #{title}
      </if>
      <if test="author != null>
         AND author like #{author}
      </if>
   </where>
</select>
```

The **where** element knows to only insert *WHERE* if there is any content returned by the containing tags. Furthermore, if that content begins with *AND* or *OR*, it knows to strip it off.

The foreach Statement:

The foreach element is very powerful, and allows you to specify a collection, declare item and index variables that can be used inside the body of the element.

It also allows you to specify opening and closing strings, and add a separator to place in between iterations. You can build an **IN** condition as follows:

```
<select id="selectPostIn" resultType="domain.blog.Post">
   SELECT *
   FROM POST P
   WHERE ID in
   <foreach item="item" index="index" collection="list"
      open="(" separator="," close=")">
         #{item}
   </foreach>
</select>
```

CHAPTER

10

iBATIS Debugging

It is easy to debug your program while working with iBATIS. iBATIS has built-in logging support and it works with the following logging libraries and searches for them in this order.

- Jakarta Commons Logging (JCL).
- Log4J
- JDK logging

You can use any of the above listed libraries alongwith iBATIS.

Debugging with Log4J:

Assuming you are going to use Log4J, which is my favorite for logging. Before proceeding you need to cross check following points:

- The Log4J JAR file (log4j-{version}.jar) should be in the CLASSPATH.
- You have log4j.properties available in the CLASSPATH.

Following is the of a log4j.properties file. Note that some of the lines are commented out. You can uncomment them if you need additiona debugging information.

```
# Global logging configuration
log4j.rootLogger=ERROR, stdout

log4j.logger.com.ibatis=DEBUG

# shows SQL of prepared statements
#log4j.logger.java.sql.Connection=DEBUG

# shows parameters inserted into prepared statements
#log4j.logger.java.sql.PreparedStatement=DEBUG

# shows query results
#log4j.logger.java.sql.ResultSet=DEBUG

#log4j.logger.java.sql.Statement=DEBUG
```

```
# Console output
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

You can find complete documentation for Log4J from Apaches site: Log4J Documentation.

iBATIS Debugging Example:

The following Java class is a very simple example that initializes, and then uses, the Log4J logging library for Java applications. We would use above mentioned property file which lies in CLASSPATH.

```
import org.apache.log4j.Logger;

import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;
import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class IbatisUpdate{
  static Logger log = Logger.getLogger(
                      IbatisUpdate.class.getName());

  public static void main(String[] args)
   throws IOException,SQLException{
   Reader rd = Resources.getResourceAsReader("SqlMapConfig.xml");
   SqlMapClient smc = SqlMapClientBuilder.buildSqlMapClient(rd);

   /* This would insert one record in Employee table. */
   log.info("Going to update record.....");
   Employee rec = new Employee();
   rec.setId(1);
   rec.setFirstName( "Roma");
   smc.update("Employee.update", rec );
   log.info("Record updated Successfully ");

   log.debug("Going to read records.....");
   List <Employee> ems = (List<Employee>)
                       smc.queryForList("Employee.getAll", null);
   Employee em = null;
   for (Employee e : ems) {
      System.out.print("  " + e.getId());
      System.out.print("  " + e.getFirstName());
      System.out.print("  " + e.getLastName());
      System.out.print("  " + e.getSalary());
      em = e;
      System.out.println("");
   }

   log.debug("Records Read Successfully ");

  }
}
```

Compilation and Run:

Here are the steps to compile and run the above mentioned software. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create Employee.xml as shown above.
- Create Employee.java as shown above and compile it.
- Create IbatisUpdate.java as shown above and compile it.
- Create log4j.properties as shown above.
- Execute IbatisUpdate binary to run the program.

You would get following result, and a record would be updated in EMPLOYEE table and later same record would be read from the EMPLOYEE table.

```
DEBUG [main] - Created connection 28405330.
DEBUG [main] - Returned connection 28405330 to pool.
DEBUG [main] - Checked out connection 28405330 from pool.
DEBUG [main] - Returned connection 28405330 to pool.
  1  Roma  Ali  5000
  2  Zara  Ali  5000
  3  Zara  Ali  5000
```

Debug Methods:

In the above example we used only info() method but you can use any of the following methods as per your requirements:

```
public void trace(Object message);
public void debug(Object message);
public void info(Object message);
public void warn(Object message);
public void error(Object message);
public void fatal(Object message);
```

CHAPTER

11

iBATIS Hibernate

There are major differences between iBatis and Hibernate but both the solutions work well, given their specific domain. Personally I would suggest you should use iBATIS if:

- You want to create your own SQL's and are willing to maintain them.
- Your environment is driven by relational data model.
- You have to work existing and complex schema's.

And simply use Hibernate if:

- Your environment is driven by object model and wants generates SQL automatically.

To count there are few differences:

- **iBATIS is:**
    - Simpler
    - Faster development time
    - Flixable
    - Much smaller in package size
- **Hibernate:**
    - Generates SQL for you which means you don't spend time on SQL
    - Provides much more advance cache
    - Highly scalable

Other difference is that iBATIS makes use of SQL which could be database dependent where as Hibernate makes use of HQL which is relatively independent of databases and it is easier to change db in Hibernate.

Hibernate maps your Java POJO objects to the Database tables where as iBatis maps the ResultSet from JDBC API to your POJO Objets.

If you are using stored procedures, well you can do it in Hibernate but it is little difficult in comparision of iBATIS. As an alternative solution iBATIS maps results sets to objects, so no need to care about table structures. This works very well for stored procedures, works very well for reporting applications, etc

Finally, Hibernate and iBATIS both are open source Object Relational Mapping(ORM) tools available in the industry. Use of each of these tools depends on the context you are using them. Hibernate and iBatis both also have good support from SPRING framework so it should not be a problem to chose one of them.

CHAPTER

12

iBATOR Introduction

IBATOR is a code generator for iBATIS. iBATOR introspects one or more database tables and will generate iBATIS artifacts that can be used to access the table(s).

Later you can write your custom SQL code or stored procedure to meet yoru requirements. iBATOR generates following artifacts:

- SqlMap XML Files
- Java Classes to match the primary key and fields of the table(s)
- DAO Classes that use the above objects (optional)

iBATOR can run as a standalone JAR file, or as an Ant task, or as an Eclipse plugin. This tutorial would teach you simplest way of generating iBATIS configuration files form command line.

Download iBATOR:

Download the standalone JAR if you are using an IDE other than Eclipse. The standalone JAR includes an Ant task to run iBATOR, or you can run iBATOR from the command line of from Java code.

- You can download zip file from Download iBATOR.
- You can check online documentation: iBATOR Documentation.

Generating Configuration File:

To get up and running quickly with Abator, follow these steps:

Step 1:

Create and fill out a configuration file ibatorConfig.xml appropriately. At a minimum, you must specify:

- A **<jdbcConnection>** element to specify how to connect to the target database.
- A **<javaModelGenerator>** element to specify target package and target project for generated Java model objects
- A **<sqlMapGenerator>** element to specify target package and target project for generated SQL map files.

- A **<daoGenerator>** element to specify target package and target project for generated DAO interfaces and classes (you may omit the <daoGenerator> element if you don't wish to generate DAOs).
- At least one database **<table>** element

NOTE: See the XML Configuration File Reference page for an example of an Abator configuration file.

Step 2:

Save the file in some convenient location for example at: \temp\ibatorConfig.xml).

Step 3:

Now run Abator from the command line with a command line as follows:

```
java -jar abator.jar -configfile \temp\abatorConfig.xml -overwrite
```

This will tell Abator to run using your configuration file. It will also tell Abator to overwrite any existing Java files with the same name. If you want to save any existing Java files, then omit the **-overwrite**parameter.

If there is a conflict, Abator will save the newly generated file with a unique name.

After running Abator, you will need to create or modify the standard iBATIS configuration files to make use of your newly generated code. This is explained in next section.

Tasks After Running Abator:

After you run Abator, you will need to create or modify other iBATIS configuration artifacts. The main tasks are as follows:

- Create or Modify the SqlMapConfig.xml file.

- Create or modify the dao.xml file (only if using the iBATIS DAO Framework).

Each task is described in detail below:

Updating the SqlMapConfig.xml File:

iBATIS uses an XML file, commonly named SqlMapConfig.xml, to specify information for a database connection, a transaction management scheme, and SQL map XML files that will be used in an iBATIS session.

Abator cannot create this file for you because Abator knows nothing about your execution environment. However, some of the items in this file relate directly to Abator generated items.

Abator specific needs in the configuration file are as follows:

- Statement namespaces must be enabled.

- Abator generated SQL Map XML files must be listed .

For example, suppose that Abator has generated an SQL Map XML file called MyTable_SqlMap.xml, and that the file has been placed in the test.xml package of your project. The SqlMapConfig.xml file should have these entries:

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE sqlMapConfig
    PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
```

```
     "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

   <sqlMapConfig>
     <!-- Statement namespaces are required for Abator -->
     <settings useStatementNamespaces="true" />

     <!-- Setup the transaction manager and data source that are
          appropriate for your environment
     -->
     <transactionManager type="...">
       <dataSource type="...">
       </dataSource>
     </transactionManager>

     <!-- SQL Map XML files should be listed here -->
     <sqlMap resource="test/xml/MyTable_SqlMap.xml" />

   </sqlMapConfig>
```

If there is more than one SQL Map XML file (as is quite common), then the files can be listed in any order with repeated <sqlMap> elements after the <transactionManager> element.

Updating the dao.xml File:

The iBATIS DAO framework is configured by an xml file commonly called dao.xml.

The iBATIS DAO framework uses this file to control the database connection information for DAOs, and also to list the DAO implementation classes and DAO interfaces.

In this file you should specify the path to your SqlMapConfig.xml file, and all the Abator generated DAO interfaces and implementation classes.

For example, suppose that Abator has generated a DAO interface called MyTableDAO and a implementation class called MyTableDAOImpl, and that the files have been placed in the test.dao package of your project.

The dao.xml file should have these entries:

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE daoConfig
   PUBLIC "-//ibatis.apache.org//DTD DAO Configuration 2.0//EN"
   "http://ibatis.apache.org/dtd/dao-2.dtd">

  <daoConfig>
    <context>
      <transactionManager type="SQLMAP">
        <property name="SqlMapConfigResource"
                  value="test/SqlMapConfig.xml"/>
      </transactionManager>

      <!-- DAO interfaces and implementations
              should be listed here -->
      <dao interface="test.dao.MyTableDAO"
           implementation="test.dao.MyTableDAOImpl" />

    </context>
  </daoConfig>
```

NOTE: This step is only required if you generated DAOs for the iBATIS DAO framework.