



Quick Start Guide

Added by [Gilles Bayon](#), last edited by [David W. Robinson](#) on Feb 02, 2008 ([view change](#))

Labels: [update](#), [schema](#), [definition](#)

A Quick Start Guide to iBATIS DataMapper

Overview

iBATIS DataMapper is a Java/.NET framework that will help you design and implement better persistence layers for your Java/.NET applications. iBATIS couples objects with stored procedures or SQL statements using a XML descriptor. Simplicity is the biggest advantage of the iBATIS Data Mapper over object relational mapping tools.

To use iBATIS Data Mapper you rely on your own objects, XML, and SQL. There is little to learn that you don't already know. With iBATIS Data Mapper you have the full power of both SQL and stored procedures at your fingertips.

Our sample project

This sample takes an "over-the-shoulder" Cookbook approach. We'll define a simple data access problem and use iBATIS to solve it for us. Let's say our client has a database and one of the tables in the database is a list of people. The client tells us:

"We would like to use a web application to display the people in this table and to add, edit, and delete individual records."

Not a complicated story, but it will cover the CRUD most developers want to learn first. 😊

Step 1 : The SQL

Let's start with the people table client mentioned. Since we're keeping it simple, we'll say it's a table in an Access database. The table definition is shown as

Name	Type	Size
PER_ID	Long Integer	4
PER_FIRST_NAME	Text	40
PER_LAST_NAME	Text	40
PER_BIRTH_DATE	Date/Time	8
PER_WEIGHT_KG	Double	8
PER_HEIGHT_M	Double	8

Step 2 :The.NET Class

Person.cs

```
using System;
namespace iBatisTutorial.Model
{
    public class Person
    {
        private int _id;
        private string _firstName;
        private string _lastName;
        private DateTime _birthDate;
        private double _weightInKilograms;
        private double _heightInMeters;

        public int Id
        {
            get{ return _id; }
            set{ _id = value; }
        }

        // Other public properties for the private fields ...
    }
}
```

Step 3 :The Nunit test

The first thing our story says is that client would like to display a list of people.

PersonTest.cs

```

using System.Collections;
using IBatisNet.DataMapper;
using NUnit.Framework;

namespace iBatisTutorial.Model
{
    [TestFixture]
    public class PersonTest
    {
        [Test]
        public void PersonList ()
        {
            // try it
            IList people = Mapper.Instance().QueryForList("SelectAll",null);

            // test it
            Assert.IsNotNull(people,"Person list not returned");
            Assert.IsTrue(people.Count>0,"Person list is empty");
            Person person = (Person) people[0];
            Assert.IsNotNull(person,"Person not returned");
        }
    }
}

```

OK, that was fun! The Assert class is built into NUnit, so to compile, we just need the Mapper object and QueryForList method. The Mapper is built into the iBATIS framework, so we don't need to write that either. The iBATIS **QueryForList** method executes our SQL statement (or stored procedure) and returns the result as a list.

Each row in the result becomes an entry in the list. Along with QueryForList, there is also Delete, Insert, Select, QueryForObject, and a couple of other methods in the iBATIS API. (See the Developers Guide for details.)

Looking at **PersonTest** class, we see that the **QueryForList** method takes the name of the statement we want to run and any runtime values the statement may need. Since a "SelectAll" statement wouldn't need any runtime values, we pass null in our test.

OK. Easy enough. But where does iBATIS get the "**SelectAll**" statement?

Some systems try to generate SQL statements for you, but iBATIS specializes in data mapping, not code generation. It's our job (or the job of our database administrator) to craft the SQL or provide a stored procedure. We then describe the statement in an XML element, like the one shown below.

Step 4 :The mapping file**Person.xml**

```

<alias>
  <typeAlias alias="Person" type="iBatisTutorial.Model.Person, iBatisTutorial.Model" />
</alias>

<resultMap id="SelectAllResult" class="Person">
  <result property="Id" column="PER_ID" />
  <result property="FirstName" column="PER_FIRST_NAME" />
  <result property="LastName" column="PER_LAST_NAME" />
  <result property="BirthDate" column="PER_BIRTH_DATE" />
  <result property="WeightInKilograms" column="PER_WEIGHT_KG" />
  <result property="HeightInMeters" column="PER_HEIGHT_M" />
</resultMap>

<select id="SelectAll" resultMap="SelectAllResult">
  select
    PER_ID,
    PER_FIRST_NAME,
    PER_LAST_NAME,
    PER_BIRTH_DATE,
    PER_WEIGHT_KG,
    PER_HEIGHT_M
  from PERSON
</select>

```

The iBATIS mapping documents can hold several sets of related elements, like those shown. We can also have as many mapping documents as we need. Having multiple mapping documents is handy when several developers are working on the project at once.

So, the framework gets the SQL code for the query from the mapping, and plugs it into a prepared statement. But, how does iBATIS know where to find the table's datasource?

Surprise! More XML! You can define a configuration file for each datasource your application uses. The example below shows a configuration file for our Access database.

Step 5 :The configuration file**SqlMap.Config - A configuration file for our Access database**

```

<?xml version="1.0" encoding="UTF-8" ?>
<sqlMapConfig
  xmlns="http://ibatis.apache.org/dataMapper"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

```

```

<database>
  <provider name="OleDb1.1"/>
  <dataSource name="iBatisTutorial"
    connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=../Resources/iBatisTutorial.mdb"/>
</database>

<sqlMaps>
  <sqlMap resource="Resources/Person.xml" />
</sqlMaps>

</sqlMapConfig>

```

Of course, besides Access, other ADO.NET providers are supported, including SqlServer, Oracle, MySQL, PostgreSQL, DB2 and generic OLEDB, ODBC providers. (See the Developers Guide for details.)

The last part of the configuration file ("sqlMaps") is where we list our mapping documents, like the one shown back in Step 4. We can list as many documents as we need here, and they will all be read when the configuration is parsed.

OK, so how does the configuration get parsed?

Look back at PersonTest class in step 3. The heart of the code is the call to the "Mapper" object (under the remark "try it"). The Mapper object is a singleton. The first time it's called, it reads in the configuration documents to create the Mapper object. On subsequent calls, it reuses the Mapper object, so that the configuration is re-read only if the file changes.

The framework comes bundled with a default Mapper class. If you want to use a different name for the configuration file, or need to use more than one database, you can also use your own class, by copying and modifying the standard version.

The following example shows the code for the standard Mapper class that comes with the framework.

Mapper.cs

```

using IBatisNet.Common.Utilities;
using IBatisNet.DataMapper;

namespace IBatisNet.DataMapper
{
    public class Mapper
    {
        private static volatile SqlMapper _mapper = null;

        protected static void Configure (object obj)
        {
            _mapper = (SqlMapper) obj;
        }
        protected static void InitMapper()
        {
            ConfigureHandler handler = new ConfigureHandler (Configure);
            _mapper = SqlMapper.ConfigureAndWatch (handler);
        }

        public static SqlMapper Instance()
        {
            if (_mapper == null)
            {
                lock (typeof (SqlMapper))
                {
                    if (_mapper == null) // double-check
                        InitMapper();
                }
            }
            return _mapper;
        }

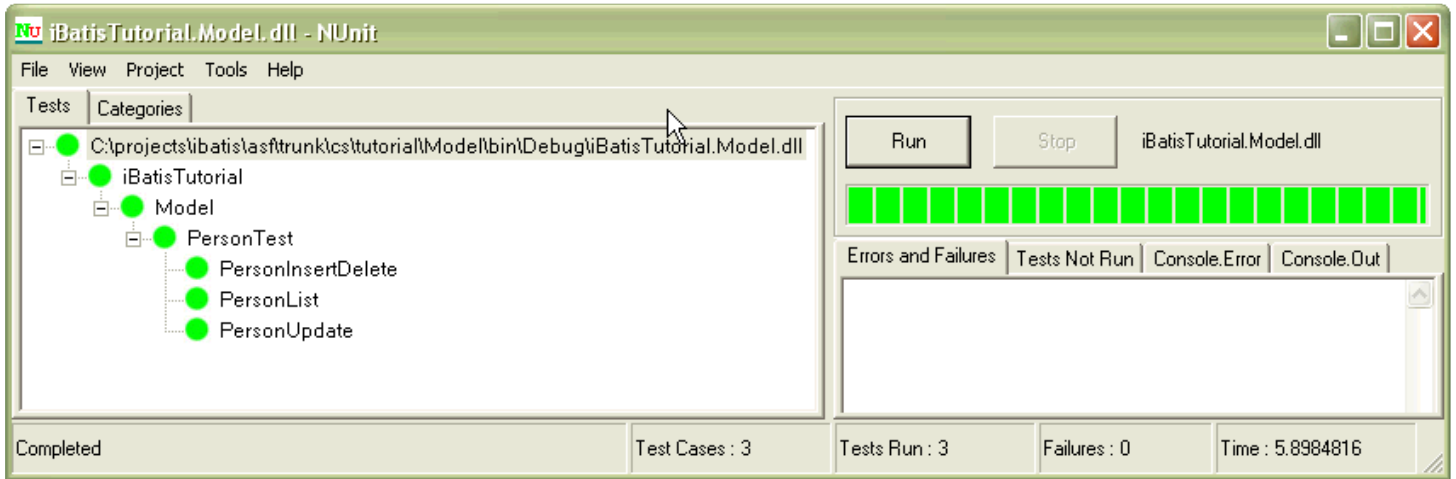
        public static SqlMapper Get()
        {
            return Instance();
        }
    }
}

```

You can access as many different database configurations as you need, just by setting up additional Mapper classes. Each Mapper configuration is a facade for a database. As far as our application knows, the "Mapper" is the database. Behind the Mapper facade, you can change the location of the database, or switch between SQL statements and stored procedures, with zero-changes to your application code.

Step 5 :Running the test

If we put this all together into a solution, we can "green bar" our test, as shown by figure.



I tried to test this example with postgres and using stored procedures and not work for me. Procedures must be used because to the architecture is based on a database that uses procedures for all operations. Someone has experience with this?

Posted by [Rasiel Aponcio Borges](#) at [Feb 02, 2009 15:36](#)

Site running on a free **Atlassian Confluence Open Source Project License** granted to OSS. [Evaluate Confluence today.](#)

Powered by [Atlassian Confluence](#), the [Enterprise Wiki](#). (Version: 2.5.5 Build:#811 Jul 25, 2007) - [Bug/feature request](#) - [Contact Administrators](#)