SQLMap PHP DataMapper Application Framework – v1.0 [1]

Wei Zhuo

April 14, 2006

Contents

Legal Notice

This document is largely based on the iBATIS.NET – DataMapper Application Framework Developer Guide.

License

SQLMap for PHP is free software released under the terms of the following BSD license.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the developer nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Chapter 1

Introduction

1.1 Overview

The SQLMap DataMapper framework makes it easier to use a database with a PHP application. SQLMap DataMapper couples objects with stored procedures or SQL statements using a XML descriptor. Simplicity is the biggest advantage of the SQLMap DataMapper over object relational mapping tools. To use SQLMap DataMapper you rely on your own objects, XML, and SQL. There is little to learn that you don't already know. With SQLMap DataMapper you have the full power of both SQL and stored procedures at your fingertips.

The SQLMap for PHP is based on iBATIS.NET - DataMapper Application Framework from http://ibatis.apache.org/.The PHP version support most of the features found in iBATIS.NET exception the following:

- Dynamic SQL.
- Distributed Transactions.

1.2 What's covered here

This Guide covers the PHP implementations of SQLMap DataMapper. The Java and .NET implementation offers the same services with some changes in the API.

Since SQLMap relies on an XML descriptor to create the mappings, much of the material applies to both implementations.

For installation instructions, see the section called the SQLMap PHP Developer Guide.

A Tutorial is also available. We recommend reviewing the Tutorial for your platform before reading this Guide.

1.3 Support

Add Forum and Trac.

1.4 Disclaimer

SQLMap MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Chapter 2

The Big Picture

2.1 Introduction

SQLMap is a simple but complete framework that makes it easy for you to map your objects to your SQL statements or stored procedures. The goal of the SQLMap framework is to obtain 80% of data access functionality using only 20% of the code.

2.2 What does it do?

Developers often create maps between objects within an application. One definition of a Mapper is an "object that sets up communication between two independent objects." A Data Mapper is a "layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself." [Patterns of Enterprise Architecture, ISBN 0-321-12742-0].

You provide the database and the objects; SQLMap provides the mapping layer that goes between the two.

2.3 How does it work?

Your programming platform already provides a capable library for accessing databases, whether through SQL statements or stored procedures. But developers find several things are still hard to do well when using

"stock" PHP function including:

Separating SQL code from programming code Passing input parameters to the library classes and extracting the output Separating data access classes from business logic classes Caching often-used data until it changes Managing transactions and many more – by using XML documents to create a mapping between a plain-old object and a SQL statement or a stored procedure. The "plain-old object" can be any PHP object.

> **Tip:** The object does not need to be part of a special object hierarchy or implement a special interface. (Which is why we call them "plain-old" objects.) Whatever you are already using should work just fine.

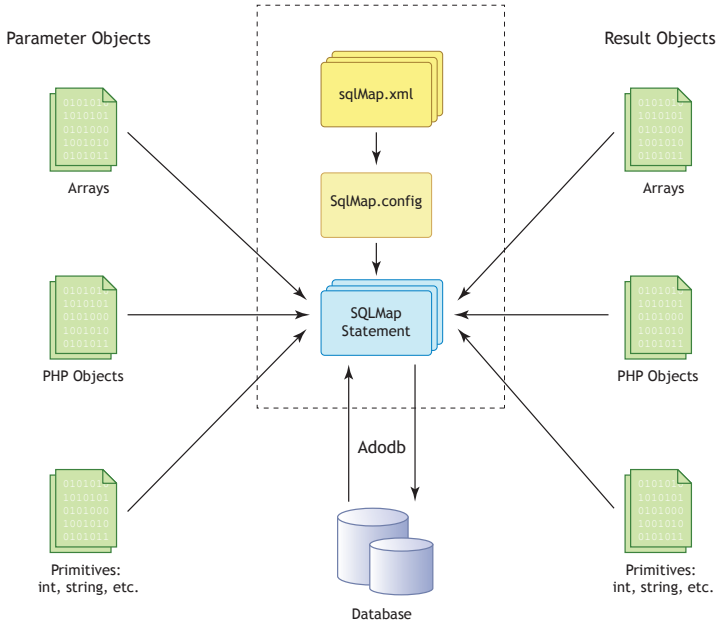

Figure 2.1: SQLMap DataMapper work flow

Here's a high level description of the work flow diagrammed by Figure 2.1: Provide a parameter, either as an object or a primitive type. The parameter can be used to set runtime values in your SQL statement or stored procedure. If a runtime value is not needed, the parameter can be omitted.

Execute the mapping by passing the parameter and the name you gave the statement or procedure in your XML descriptor. This step is where the magic happens. The framework will prepare the SQL statement or stored procedure, set any runtime values using your parameter, execute the procedure or statement, and return the result.

In the case of an update, the number of rows affected is returned. In the case of a query, a single object, or a collection of objects is returned. Like the parameter, the result object, or collection of objects, can be a plain-old object or a primitive type.

So, what does all this look like in your source code? Here's how you might code the insert of a "lineItem" object into your database.

```
TMapper::instance()->insert("InsertLineItem", $lineItem);
```

If your database is generating the primary keys, the generated key can be returned from the same method call, like this:

```
$myKey = TMapper::instance()->insert("InsertLineItem", $lineItem);
```

Example 2.3.1 shows an XML descriptor for "InsertLineItem".

Example 2.3.1 *The "InsertLineItem" descriptor*

```
<insert id="InsertLineItem" parameterClass="LineItem">
  INSERT INTO [LinesItem]
    (Order_Id, LineItem_LineNum, Item_Id, LineItem_Quantity, LineItem_UnitPrice)
  VALUES
    (#Order.Id#, #LineNumber#, #Item.Id#, #Quantity#, #Item.ListPrice#)
 <selectKey type="post" resultClass="int" property="Id" >
  select @@IDENTITY as value
 </selectKey>
</insert>
```

The `<selectKey>` stanza returns an auto-generated key from a SQL Server database (for example). If you need to select multiple rows, SQLMap can return a list of objects, each mapped to a row in the result set:

```
$productList = Mapper::instance()->queryForList("selectProduct",$categoryKey);
```

Or just one, if that's all you need:

```
$product = Mapper::instance()->queryForObject("selectProduct",$categoryKey);
```

Of course, there's more, but this is SQLMap from 10,000 meters. (For a longer, gentler introduction, see the Tutorial.) Section 3 describes the Data Map definition files – where the statement for "InsertLineItem" would be defined. The Developers Guide for PHP (Section **??**) describes the "bootstrap" configuration file that exposes SQLMap to your application.

2.4 Is SQLMap the best choice for my project?

SQLMap is a Data Mapping tool. Its role is to map the columns of a database query (including a stored procedure) to the properties of an object. If your application is based on business objects (including array or lists of objects), then SQLMap can be a good choice. SQLMap is an even better choice when your application is layered, so that that the business layer is distinct from the user-interface layer.

Under these circumstances, another good choice would be an Object/Relational Mapping tool (OR/M tool), like [...]. Other products in this category are [...] and [...] . An OR/M tool generates all or most of the SQL for you, either beforehand or at runtime. These products are called OR/M tools because they try to map an object graph to a relational schema.

SQLMap is not an OR/M tool. SQLMap helps you map objects to stored procedures or SQL statements. The underlying schema is irrelevant. An OR/M tool is great if you can map your objects to tables. But they are not so great if your objects are stored as a relational view rather than as a table. If you can write a statement or procedure that exposes the columns for your object, regardless of how they are stored, SQLMap can do the rest.

So, how do you decide whether to OR/M or to DataMap? As always, the best advice is to implement a representative part of your project using either approach, and then decide. But, in general, OR/M is a good thing when you

- Have complete control over your database implementation.
- Do not have a Database Administrator or SQL guru on the team.
- Need to model the problem domain outside the database as an object graph.

Likewise, the best time to use a Data Mapper, like SQLMap, is when:

- You do not have complete control over the database implementation, or want to continue to access a legacy database as it is being refactored.
- You have database administrators or SQL gurus on the team.
- The database is being used to model the problem domain, and the application's primary role is to help the client use the database model.

In the end, you have to decide what's best for your project. If a OR/M tool works better for you, that's great! If your next project has different needs, then we hope you give SQLMap another look. If SQLMap works for you now: Excellent!

Chapter 3

Working with Data Maps

3.1 Introduction

If you want to know how to configure and install SQLMap, see the Developer Guide section **??** . But if you want to know how SQLMap really works, continue from here.

The Data Map definition file is where the interesting stuff happens. Here, you define how your application interacts with your database. As mentioned, the Data Map definition is an XML descriptor file. By using a service routine provided by SQLMap, the XML descriptors are rendered into a client object (or Mapper). To access your Data Maps, your application calls the client object and passes in the name of the statement you need.

The real work of using SQLMap is not so much in the application code, but in the XML descriptors that SQLMap renders. Instead of monkeying with application source code, you monkey with XML descriptors instead. The benefit is that the XML descriptors are much better suited to the task of mapping your object properties to database entities. At least, that's our own experience with our own applications. Of course, your mileage may vary.

3.2 What's in a Data Map definition file, anyway?

If you read the Tutorial, you've already seen some simple Data Map examples, like the one shown in Example 2.3.1.

Example 3.2.1 *A simple Data Map (PHP)*

```
<?xml version="1.0" encoding="UTF-8" ?>
  <sqlMap namespace="LineItem">
    <insert id="InsertLineItem" parameterClass="LineItem">
      INSERT INTO [LinesItem]
        (Order_Id, LineItem_LineNum, Item_Id, LineItem_Quantity, LineItem_UnitPrice)
      VALUES
       (#Order.Id#, #LineNumber#, #Item.Id#, #Quantity#, #Item.ListPrice#)
    </insert>
</sqlMap>
```

This map takes some properties from a `LineItem` instance and merges the values into the SQL statement. The value-add is that our SQL in separated from our program code, and we can pass our `LineItem` instance directly to a library method:

```
TMapper::instance()->insert("InsertLineItem",$lineItem);
```

No fuss, no muss. Likewise, see Example3.2.2 for a simple select statement.

Info: A Quick Glance at Inline Parameters

Say we have a mapped statement element that looks like this:

```
 <statement id="InsertProduct">
  insert into Products (Product_Id, Product_Description)
  values (#Id#, #Description#);
</statement>
```

The inline parameters here are `#Id#` and `#Description#`. Let's also say that we have an object with the properties `Id` and `Description`. If we set the object properties to 5 and "dog", respectively, and passed the object to the mapped statement, we'd end up with a runtime query that looked like this:

```
insert into Products (Product_Id, Product_Description) values (5, 'dog');
```

For more about inline parameters, see Chapter 4.

But, what if you wanted some ice cream with that pie? And maybe a cherry on top? What if we wanted to cache the result of the select? Or, what if we didn't want to use SQL aliasing or named parameters. (Say, because we were using pre-existing SQL that we didn't want to touch.) Example 3.2.2 shows a Data Map that specifies a cache, and uses a `<parameterMap>` and a `<resultMap>` to keep our SQL pristine.

Example 3.2.2 *A Data Map definition file with some bells and whistles*

```
<?xml version="1.0" encoding="UTF-8" ?>
  <sqlMap namespace="Product">

    <cacheModel id="productCache" type="LRU">
      <flushInterval hours="24"/>
      <property name="CacheSize" value="1000" />
    </cacheModel>

    <resultMap id="productResult" class="Product">
      <result property="Id" column="Product_Id"/>
      <result property="Description" column="Product_Description"/>
    </resultMap>

    <select id="GetProduct" parameterMap="productParam" cacheModel="productCache">
      select * from Products where Product_Id = ?
    </select>
```

```
    <parameterMap id="productParam" class="Product">
      <parameter property="Id"/>
    </parameterMap>

</sqlMap>
```

In Example 3.2.2, `<parameterMap>` maps the SQL "?" to the product `Id` property. The `<resultMap>` maps the columns to our object properties. The `<cacheModel>` keeps the result of the last one thousand of these queries in active memory for up to 24 hours.

Example 3.2.2 is longer and more complex than Example 3.2.1, but considering what you get in return, it seems like a fair trade. (A bargain even.)

Many agile developers would start with something like Example 3.2.1 and add features like caching later. If you changed the Data Map from Example 3.2.1 to Example 3.2.2, you would not have to touch your application source code at all. You can start simple and add complexity only when it is needed.

A single Data Map definition file can contain as many Cache Models, Type Aliases, Result Maps, Parameter Maps, and Mapped Statements (including stored procedures), as you like. Everything is loaded into the same configuration, so you can define elements in one Data Map and then use them in another. Use discretion and organize the statements and maps appropriately for your application by finding some logical way to group them.

3.3 Mapped Statements

Mapped Statements can hold any SQL statement and can use Parameter Maps and Result Maps for input and output. (A stored procedure is a specialized form of a statement. See section 3.3.1 and 3.3.2 for more information.)

If the case is simple, the Mapped Statement can reference the parameter and result classes directly. Mapped Statements support caching through reference to a Cache Model element. The following example shows the syntax for a statement element.

Example 3.3.1 *Statement element syntax*

```
<statement id="statement.name"
```

```
  [parameterMap="parameterMap.name"]
  [parameterClass="class.name"]
  [resultMap="resultMap.name"]
  [resultClass="class.name"]
  [listClass="class.name"]
  [cacheModel="cache.name"]
>

  select * from Products where Product_Id = [?|#propertyName#]
  order by [$simpleDynamic$]

</statement>
```

In Example 3.3.1, the [bracketed] parts are optional, and some options are mutually exclusive. It is perfectly legal to have a Mapped Statement as simple as shown by Example 3.3.2.

Example 3.3.2 *A simplistic Mapped Statement*

```
<statement id="InsertTestProduct" >
  insert into Products (Product_Id, Product_Description) values (1, "Shih Tzu")
</statement>
```

Example 3.3.2 is obviously unlikely, unless you are running a test. But it does shows that you can use SQLMap to execute arbitrary SQL statements. More likely, you will use the object mapping features with Parameter Maps (Chapter 4) and Result Maps (Chapter 5) since that's where the magic happens.

3.3.1 Statement Types

The `<statement>` element is a general "catch all" element that can be used for any type of SQL statement. Generally it is a good idea to use one of the more specific statement-type elements. The more specific elements provided better error-checking and even more functionality. (For example, the insert statement can return a database-generated key.) Table 3.1 summarizes the statement-type elements and their supported attributes and features. The various attributes used by statement-type elements are covered in Section 3.5.

Table 3.1: The six statement-type elements

Statement Element	Attribute	Child Elements	Methods
`<statement>`	id parameterClass resultClass listClass parameterMap resultMap cacheModel	None	Insert Update Delete All query methods
`<insert>`	id parameterClass parameterMap	`<selectKey>` `<generate>`	Insert Update Delete
`<update>`	id parameterClass parameterMap extends	`<generate>`	Insert Update Delete
`<delete>`	id parameterClass parameterMap extends	`<generate>`	Insert Update Delete
`<select>`	id parameterClass resultClass listClass parameterMap resultMap cacheModel extends	`<generate>`	All query methods
`<procedure>`	id parameterMap resultClass resultMap cacheModel	None	Insert Update Delete All query methods

3.3.2 Stored Procedures

????

3.4 The SQL

If you are not using stored procedures, the most important part of a statement-type element is the SQL. You can use any SQL statement that is valid for your database system. Since SQLMap passes the SQL through to a standard libraries (Adodb for PHP), you can use any statement with SQLMap that you could use without SQLMap. You can use whatever functions your database system supports, and even send multiple statements, so long as your driver or provider supports them.

3.4.1 Escaping XML symbols

Because you are combining SQL and XML in a single document, conflicts can occur. The most common conflict is the greater-than and less-than symbols (><). SQL statements use these symbols as operators, but they are reserved symbols in XML. A simple solution is to escape the SQL statements that uses XML reserved symbols within a CDATA element. Example 3.4.1 demonstrates this.

Example 3.4.1 *Using CDATA to "escape" SQL code*

```
<statement id="SelectPersonsByAge" parameterClass="int" resultClass="Person">
  <![CDATA[
     SELECT * FROM PERSON WHERE AGE > #value#
  ]]>
</statement>
```

3.4.2 Auto-Generated Keys

Many database systems support auto-generation of primary key fields, as a vendor extension. Some vendors pre-generate keys (e.g. Oracle), some vendors post-generate keys (e.g. MS-SQL Server and MySQL). In either case, you can obtain a pre-generated key using a `<selectKey>` stanza within an `<insert>` element. Example 3.4.2 shows an `<insert>` statement for either approach.

Example 3.4.2 <insert> statements using <selectKey> stanzas

```
<!ąłOracle SEQUENCE Example using .NET 1.1 System.Data.OracleClient -->
<insert id="insertProduct-ORACLE" parameterClass="product">
  <selectKey resultClass="int" type="pre" property="Id" >
     SELECT STOCKIDSEQUENCE.NEXTVAL AS VALUE FROM DUAL
  </selectKey>
  insert into PRODUCT (PRD_ID,PRD_DESCRIPTION) values (#id#,#description#)
</insert>

<!ął Microsoft SQL Server IDENTITY Column Example -->
<insert id="insertProduct-MS-SQL" parameterClass="product">
  insert into PRODUCT (PRD_DESCRIPTION)
  values (#description#)
 <selectKey resultClass="int" type="post" property="id" >
   select @@IDENTITY as value
 </selectKey>
</insert>

<!-- MySQL Example -->
<insert id="insertProduct-MYSQL" parameterClass="product">
  insert into PRODUCT (PRD_DESCRIPTION)
  values (#description#)
 <selectKey resultClass="int" type="post" property="id" >
   select LAST_INSERT_ID() as value
 </selectKey>
</insert>
```

3.4.3 <generate> tag

You can use SQLMap to execute any SQL statement your application requires. When the requirements for a statement are simple and obvious, you may not even need to write a SQL statement at all. The <generate> tag can be used to create simple SQL statements automatically, based on a <parameterMap> element. The four CRUD statement types (insert, select, update, and delete) are supported. For a select, you can select all or select by a key (or keys). Example 3.4.3 shows an example of generating the usual array of CRUD statements.

> **Important:** The intended use of the <generate> tag is to save developers the trouble of coding mundane SQL statements (and only mundane statements). It is not meant as a object-to-relational mapping tool. There are many frameworks that provide extensive object-to-relational mapping features. The <generate> tag is not a replacement for any of those. When the <generate> tag does not suit your needs, use a conventional statement instead.

Example 3.4.3 Creating the "usual suspects" with the <generate> tag

```
<parameterMap id="insert-generate-params">
  <parameter property="Name" column="Category_Name"/>
  <parameter property="Guid" column="Category_Guid" dbType="UniqueIdentifier"/>
</parameterMap>

<parameterMap id="update-generate-params" extends="insert-generate-params">
  <parameter property="Id" column="Category_Id" />
</parameterMap>

<parameterMap id="delete-generate-params">
  <parameter property="Id" column="Category_Id" />
  <parameter property="Name" column="Category_Name"/>
</parameterMap>

<parameterMap id="select-generate-params">
  <parameter property="Id" column="Category_Id" />
  <parameter property="Name" column="Category_Name"/>
  <parameter property="Guid" column="Category_Guid" dbType="UniqueIdentifier"/>
</parameterMap>

<update id="UpdateCategoryGenerate" parameterMap="update-generate-params">
  <generate table="Categories" by="Category_Id"/>
</update>

<delete id="DeleteCategoryGenerate" parameterMap="delete-generate-params">
  <generate table="Categories" by="Category_Id, Category_Name"/>
</delete>
```

```
<select id="SelectByPKCategoryGenerate" resultClass="Category" parameterClass="Categ
        parameterMap="select-generate-params">
  <generate table="Categories" by="Category_Id"/>
</select>

<select id="SelectAllCategoryGenerate" resultClass="Category"
        parameterMap="select-generate-params">
  <generate table="Categories" />
</select>

<insert id="InsertCategoryGenerate" parameterMap="insert-generate-params">
  <selectKey property="Id" type="post" resultClass="int">
    select @@IDENTITY as value
  </selectKey>
  <generate table="Categories" />
</insert>
```

The tag generates ANSI SQL, which should work with any compliant database. Special types, such as blobs, are not supported, and vendor-specific types are also not supported. But, the generate tag does keep the simple things simple.

> **Note:** The SQL is generated when the DataMapper instance is built and can be cached afterward, so there is no performance impact at execution time.

The generate tag supports two attributes :

Table 3.2: `<generate>` attributes

Attribute	Description	Required
table	specifies the table name to use in the SQL statement.	yes
by	specifies the columns to use in a WHERE clause	no

3.5 Statement-type Element Attributes

The six statement-type elements take various attributes. See Section 3.3.1 for a table itemizing which attributes each element-type accepts. The individual attributes are described in the sections that follow.

3.5.1 `id` attribute

The required `id` attribute provides a name for this statement, which must be unique within this `<SqlMap>`.

3.5.2 `parameterMap` attribute

A Parameter Map defines an ordered list of values that match up with the "?" placeholders of a standard, parameterized query statement. Example 3.5.1 shows a `<parameterMap>` and a corresponding `<statement>`.

Example 3.5.1 *A `parameterMap` and corresponding statement*

```
<parameterMap id="insert-product-param" class="Product">
  <parameter property="id"/>
  <parameter property="description"/>
</parameterMap>

<statement id="insertProduct" parameterMap="insert-product-param">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</statement>
```

In Example 3.5.1, the Parameter Map describes two parameters that will match, in order, two placeholders in the SQL statement. The first "?" is replaced by the value of the `id` property. The second is replaced with the `description` property.

SQLMap also supports named, inline parameters, which most developers seem to prefer. However, Parameter Maps are useful when the SQL must be kept in a standard form or when extra information needs to be provided. For more about Parameter Maps see Chapter 4.

3.5.3 `parameterClass` attribute

If a `parameterMap` attribute is not specified, you may specify a `parameterClass` instead and use inline parameters (see Section 4.3). The value of the `parameterClass` attribute can be any existing PHP class name. Example 3.5.2 shows a statement using a PHP class named `Product` in `parameterClass` attribute.

Example 3.5.2 *Specify the `parameterClass` with a PHP class name.*

```
<statement id="statementName" parameterClass="Product">
  insert into PRODUCT values (#id#, #description#, #price#)
</statement>
```

3.5.4 `resultMap` attribute

A Result Map lets you control how data is extracted from the result of a query, and how the columns are mapped to object properties. Example 3.5.3 shows a `<resultMap>` element and a corresponding `<statement>` element.

Example 3.5.3 *A `<resultMap>` and corresponding `<statement>`*

```
<resultMap id="select-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

<statement id="selectProduct" resultMap="select-product-result">
  select * from PRODUCT
</statement>
```

In Example 3.5.3, the result of the SQL query will be mapped to an instance of the `Product` class using the "select-product-result" `<resultMap>`. The `<resultMap>` says to populate the `id` property from the `PRD_ID` column, and to populate the `description` property from the `PRD_DESCRIPTION` column.

> **Tip:** In Example 3.5.3, note that using " select * " is supported. If you want all the columns, you don't need to map them all individually. (Though many developers consider it a good practice to always specify the columns expected.)

For more about Result Maps, see Chapter 5.

3.5.5 `resultClass` attribute

If a `resultMap` is not specified, you may specify a `resultClass` instead. The value of the `resultClass` attribute can be the name of a PHP class or primitives like `integer`, `string`, or `array`. The class specified will be automatically mapped to the columns in the result, based on the result metadata. The following example shows a `<statement>` element with a `resultClass` attribute.

Example 3.5.4 *A `<statement>` element with `resultClass` attribute*

```
<statement id="SelectPerson" parameterClass="int" resultClass="Person">
  SELECT
  PER_ID as Id,
  PER_FIRST_NAME as FirstName,
  PER_LAST_NAME as LastName,
  PER_BIRTH_DATE as BirthDate,
  PER_WEIGHT_KG as WeightInKilograms,
  PER_HEIGHT_M as HeightInMeters
  FROM PERSON
  WHERE PER_ID = #value#
</statement>
```

In Example 3.5.4, the `Person` class has properties including: `Id`, `FirstName`, `LastName`, `BirthDate`, `WeightInKilograms`, and `HeightInMeters`. Each of these corresponds with the column aliases described by the SQL select statement using the "as" keyword ÍCa standard SQL feature. When executed, a `Person` object is instantiated and populated by matching the object property names to the column names from the query.

Using SQL aliases to map columns to properties saves defining a `<resultMap>` element, but there are limitations. There is no way to specify the types of the output columns (if needed), there is no way to automatically load related data such as complex properties.You can overcome these limitations with an explicit Result Map (Chapter 5).

3.5.6 `listClass` attribute

In addition to providing the ability to return an `TList` of objects, the DataMapper supports the use of custom collection: a class that implements `ArrayAccess`. The following is an example of a TList (it

implements ArrayAccess) class that can be used with the DataMapper.

Example 3.5.5 *An `ArrayAccess` implementation, by extending `TList`.*

```
class AccountCollection extends TList
{
    public function addRange($accounts)
    {
        foreach($accounts as $account)
            $this->add($account);
    }

    public function copyTo(TList $array)
    {
        $array->copyFrom($this);
    }
}
```

An `ArrayAccess` class can be specified for a select statement through the `listClass` attribute. The value of the `listClass` attribute is the full name of a PHP class that implements `ArrayAccess`. The statement should also indicate the `resultClass` so that the DataMapper knows how to handle the type of objects in the collection. The `resultClass` specified will be automatically mapped to the columns in the result, based on the result metadata. The following example shows a `<statement>` element with a `listClass` attribute.

Example 3.5.6 *A `<statement>` element with `listClass` attribute*

```
<statement id="GetAllAccounts"
 listClass="AccountCollection"
 resultClass="Account">
   select
   Account_ID as Id,
   Account_FirstName as FirstName,
   Account_LastName as LastName,
   Account_Email as EmailAddress
   from Accounts
```

```
   order by Account_LastName, Account_FirstName
</statement>
```

3.5.7 `cacheModel` attribute

If you want to cache the result of a query, you can specify a Cache Model as part of the `<statement>` element. Example 3.5.7 shows a `<cacheModel>` element and a corresponding `<statement>`.

Example 3.5.7 *A `<cacheModel>` element with its corresponding `<statement>`*

```
<cacheModel id="product-cache" implementation="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="size" value="1000" />
</cacheModel>

<statement id="selectProductList" parameterClass="int" cacheModel="product-cache">
  select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

In Example 3.5.7, a cache is defined for products that uses a Least Recently Used [LRU] type and flushes every 24 hours or whenever associated update statements are executed. For more about Cache Models, see Section 6.

3.5.8 `extends` attribute

When writing Sql, you often encounter duplicate fragments of SQL. SQLMap offers a simple yet powerful attribute to reuse them.

```
<select id="GetAllAccounts"
  resultMap="indexed-account-result">
select
```

```
  Account_ID,
  Account_FirstName,
  Account_LastName,
  Account_Email
from Accounts
</select>

<select id="GetAllAccountsOrderByName"
  extends="GetAllAccounts"
  resultMap="indexed-account-result">
    order by Account_FirstName
</select>
```

Chapter 4

Parameter Maps and Inline Parameters

Most SQL statements are useful because we can pass them values at runtime. Someone wants a database record with the ID 42, and we need to merge that ID number into a select statement. A list of one or more parameters are passed at runtime, and each placeholder is replaced in turn. This is simple but labor intensive, since developers spend a lot of time counting symbols to make sure everything is in sync.

> **Note:** Preceding sections briefly touched on inline parameters, which automatically map properties to named parameters. Many iBATIS developers prefer this approach. But others prefer to stick to the standard, anonymous approach to SQL parameters by using parameter maps. Sometimes people need to retain the purity of the SQL statements; other times they need the detailed specification offered by parameter maps due to database or provider-specific information that needs to be used.

4.1 Parameter Map

A Parameter Map defines an ordered list of values that match up with the placeholders of a parameterized query statement. While the attributes specified by the map still need to be in the correct order, each parameter is named. You can populate the underlying class in any order, and the Parameter Map ensures each value is passed in the correct order.

Parameter Maps can be provided as an external element and *inline*. Example 4.1.1 shows an external Parameter Map.

Example 4.1.1 *An external Parameter Map*

```
<parameterMap id="parameterMapIdentifier"
  [extends="[sqlMapNamespace.]parameterMapId"]>
  <parameter
    property ="propertyName"
    [column="columnName"]
    [dbType="databaseType"]
    [type="propertyCLRType"]
    [nullValue="nullValueReplacement"]
    [size="columnSize"]
    [precision="columnPrecision"]
    [scale="columnScale"]
    [typeHandler="class.name"]
  <parameter ... ... />
  <parameter ... ... />
</parameterMap>
```

In Example 4.1.1, the parts in [brackets] are optional. The `parameterMap` element only requires the id attribute. Example 4.1.2 shows a typical `<parameterMap>`.

Example 4.1.2 *A typical `<parameterMap>` element*

```
<parameterMap id="insert-product-param" class="Product">
  <parameter property="description" />
  <parameter property="id"/>
</parameterMap>

<statement id="insertProduct" parameterMap="insert-product-param">
  insert into PRODUCT (PRD_DESCRIPTION, PRD_ID) values (?,?);
</statement>
```

> **Note:** Parameter Map names are always local to the Data Map definition file where they are defined. You can refer to a Parameter Map in another Data Map definition file by prefixing the `id` of the Parameter Map with the namespace of the Data Map (set in the `<sqlMap>` root element). If the Parameter Map in Example 4.1.2 were in a Data Map named "Product", it could be referenced from another file using "Product.insert-product-param".

4.1.1 <parameterMap> attributes

The <parameterMap> element accepts two attributes: id (required) and extends (optional).

id attribute

The required id attribute provides a unique identifier for the <parameterMap> within this Data Map.

extends attribute

The optional extends attribute can be set to the name of another parameterMap upon which to base this parameterMap. All properties of the super parameterMap will be included as part of this parameterMap, and values from the super parameterMap are set before any values specified by this parameterMap. The effect is similar to extending a class.

4.2 <parameter> Elements

The <parameterMap> element holds one or more parameter child elements that map object properties to placeholders in a SQL statement. The sections that follow describe each of the attributes.

4.2.1 property attribute

The property attribute of <parameter> is the name of a property of the parameter object. It may also be the name of an entry in an array. The name can be used more than once depending on the number of times it is needed in the statement. (In an update, you might set a column that is also part of the where clause.)

4.2.2 direction attribute

The direction attribute may be used to indicate a stored procedure parameter's direction.

4.2.3 `column` attribute

The `column` attribute is used to define to the name of a parameter used by a stored procedure.

Table 4.1: Parameter `direction` attribute values

Value	Description
Input	input-only
Output	output-only
InputOutput	bidirectional

4.2.4 `dbType` attribute

The `dbType` attribute is used to explicitly specify the database column type of the parameter to be set by this property. This attribute is normally only required if the column is nullable. Although, another reason to use the `dbType` attribute is to explicitly specify date types. Most SQL databases have more than one `datetime` type. Usually, a database has at least three different types (DATE, DATETIME, TIMESTAMP). In order for the value to map correctly, you might need to specify the column's `dbType`.

> **Note:** Most providers only need the `dbType` specified for nullable columns. In this case, you only need to specify the type for the columns that are nullable.

4.2.5 `type` attribute

The `type` attribute is used to specify the type of the parameter's property. This attribute is useful when passing `InputOutput` and `Output` parameters into stored procedures. The framework uses the specified type to properly handle and set the parameter object's properties with the procedure's output values after execution.

4.2.6 `nullValue` attribute

The `nullValue` attribute can be set to any valid value (based on property type). The `nullValue` attribute is used to specify an outgoing null value replacement. What this means is that when the value is detected in the object property, a NULL will be written to the database (the opposite behavior of an inbound null value replacement). This allows you to use a magic null number in your application for types that do

not support null values (such as int, double, float). When these types of properties contain a matching null value (for example, say, -9999), a NULL will be written to the database instead of the value.

> **Tip:** For round-trip transparency of null values, you must also specify database columns null value replacements in your Result Map (see Chapter 5).

4.2.7 `size` attribute

The `size` attribute sets the maximum size of the data within the column.

4.2.8 `precision` attribute

The `precision` attribute is used to set the maximum number of digits used to represent the property value.

4.2.9 `scale` attribute

The `scale` attribute sets the number of decimal places used to resolve the property value.

4.2.10 `typeHandler` attribute

The `typeHandler` attribute allows the use of a Custom Type Handler (see the Custom Type Handler section). This allows you to extend the DataMapper's capabilities in handling types that are specific to your database provider, are not handled by your database provider, or just happen to be a part of your application design. You can create custom type handlers to deal with storing and retrieving booleans from your database for example.

4.3 Inline Parameter Maps

If you prefer to use inline parameters instead of parameter maps, you can add extra type information inline too. The inline parameter map syntax lets you embed the property name, the property type, the column type,

and a null value replacement into a parametrized SQL statement. The next four examples shows statements written with inline parameters.

Example 4.3.1 *A <statement> using inline parameters*

```
<statement id="insertProduct" parameterClass="Product">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
  values (#id#, #description#)
</statement>
```

The following example shows how `dbTypes` can be declared inline.

Example 4.3.2 *A <statement> using an inline parameter map with a type*

```
<statement id="insertProduct" parameterClass="Product">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
  values (#id, dbType=int#, #description, dbType=VarChar#)
</statement>
```

The next example shows how `dbTypes` and null value replacements can also be declared inline.

Example 4.3.3 *A <statement> using an inline parameter map with a null value replacement*

```
<statement id="insertProduct" parameterClass="Product">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
  values (#id, dbType=int, nullValue=-999999#, #description, dbType=VarChar#)
</statement>
```

Example 4.3.4 *A more complete example.*

```
<update id="UpdateAccountViaInlineParameters" parameterClass="Account">
 update Accounts set
 Account_FirstName = #FirstName#,
 Account_LastName = #LastName#,
 Account_Email = #EmailAddress,type=string,dbType=Varchar,nullValue=no_email@provided.
```

```
 where
 Account_ID = #Id#
</update>
```

> **Note:** Inline parameter maps are handy for small jobs, but when there are a lot of type descriptors and null value replacements in a complex statement, an industrial-strength, external `parameterMap` can be easier.

4.4 Standard Type Parameters

In practice, you will find that many statements take a single parameter, often an `integer` or a `string`. Rather than wrap a single value in another object, you can use the standard library object (string, integer, et cetera) as the parameter directly. Example 4.4.1 shows a statement using a standard type parameter.

Example 4.4.1 *A `<statement>` using standard type parameters*

```
<statement id="getProduct" parameterClass="System.Int32">
  select * from PRODUCT where PRD_ID = #value#
</statement>
```

Assuming `PRD_ID` is a numeric type, when a call is made to this Mapped Statement, a standard integer can be passed in. The `#value#` parameter will be replaced with the value of the integer. The name `value` is simply a placeholder, you can use another moniker of your choice. Result Maps support primitive types as results as well.

For your convenience, the following PHP primitive types are supported.

- `string`
- `float` or `double`
- `integer` or `int`
- `bool` or `boolean`

4.5 Array Type Parameters

You can also pass in a array as a parameter object. This would usually be a an associative array. Example 4.5.1 shows a `<statement>` using an array for a `parameterClass`.

Example 4.5.1 *A `<statement>` using an array for a `parameterClass`*

```
<statement id="getProduct" parameterClass="array">
  select * from PRODUCT
  where PRD_CAT_ID = #catId#
  and PRD_CODE = #code#
</statement>
```

In Example 4.5.1, notice that the SQL in this Mapped Statement looks like any other. There is no difference in how the inline parameters are used. If an associative array is passed, it must contain keys named `catId` and `code`. The values referenced by those keys must be of the appropriate type for the column, just as they would be if passed from a properties object.

Chapter 5

Result Maps

Chapter 4 describes Parameter Maps and Inline parameters, which map object properties to parameters in a database query. Result Maps finish the job by mapping the result of a database query (a set of columns) to object properties. Next to Mapped Statements, the Result Map is probably one of the most commonly used and most important features to understand.

A Result Map lets you control how data is extracted from the result of a query, and how the columns are mapped to object properties. A Result Map can describe the column type, a null value replacement, and complex property mappings including Collections. Example 5.0.2 shows the structure of a `<resultMap>` element.

Example 5.0.2 *The structure of a `<resultMap>` element.*

```
<resultMap id="resultMapIdentifier"
           [class="class.name"]
           [extends="[sqlMapNamespace.]resultMapId"]>

   <result property="propertyName"
           column="columnName"
           [columnIndex="columnIndex"]
           [dbType="databaseType"]
           [type="propertyCLRType"]
           [resultMapping="resultMapName"]
```

```
            [nullValue="nullValueReplacement"]
            [select="someOtherStatementName"]
            [lazyLoad="true|false"]
            [typeHandler="class.name"]
    />
    <result ... .../>
    <result ... .../>
</resultMap>
```

In Example 5.0.2, the [brackets] indicate optional attributes. The `id` attribute is required and provides a name for the statement to reference. The `class` attribute is also required, and specifies the full name of a PHP class. This is the class that will be instantiated and populated based on the result mappings it contains.

The `resultMap` can contain any number of property mappings that map object properties to the columns of a result element. The property mappings are applied, and the columns are read, in the order that they are defined. Maintaining the element order ensures consistent results between different drivers and providers.

> **Note:** As with parameter classes, the result class must be a PHP class object or array instance.

5.1 Extending `resultMaps`

The optional `extends` attribute can be set to the name of another `resultMap` upon which to base this `resultMap`. All properties of the "super" `resultMap` will be included as part of this `resultMap`, and values from the "super" `resultMap` are set before any values specified by this `resultMap`. The effect is similar to extending a class.

> **Tip:** The "super" `resultMap` must be defined in the file before the extending `resultMap`. The classes for the super and sub `resultMaps` need not be the same, and do not need to be related in any way.

5.2 `<resultMap>` attributes

The `<resultMap>` element accepts three attributes: `id` (required), `class` (optional), and `extends` (optional).

5.2.1 `id` attribute

The required `id` attribute provides a unique identifier for the `<resultMap>` within this Data Map.

5.2.2 `class` attribute

The optional `class` attribute specifies an object class to use with this `<resultMap>`. The full classname must be specified. Any class can be used.

> **Note:** As with parameter classes, the result class must be a PHP class object or array instance.

5.2.3 `extends` attribute

The optional `extends` attribute allows the result map to inherit all of the properties of the "super" `resultMap` that it extends.

5.3 `<result>` Elements

The `<resultMap>` element holds one or more `<result>` child elements that map SQL result sets to object properties.

5.3.1 `property` attribute

The `property` attribute is the name of a property of the result object that will be returned by the Mapped Statement. The name can be used more than once depending on the number of times it is needed to populate the results.

5.3.2 `column` attribute

The `column` attribute value is the name of the column in the result set from which the value will be used to populate the property.

5.3.3 `columnIndex` attribute

The `columnIndex` attribute value is the index of the column in the ResultSet from which the value will be used to populate the object property. This is not likely needed in 99% of applications and sacrifices maintainability and readability for speed. Some providers may not realize any performance benefit, while others will speed up dramatically.

5.3.4 `dbType` attribute

The `dbType` attribute is used to explicitly specify the database column type of the ResultSet column that will be used to populate the object property. Although Result Maps do not have the same difficulties with null values, specifying the type can be useful for certain mapping types such as Date properties. Because an application language has one Date value type and SQL databases may have many (usually at least 3), specifying the date may become necessary in some cases to ensure that dates (or other types) are set correctly. Similarly, String types may be populated by a `VarChar`, `Char` or `CLOB`, so specifying the type might be needed in those cases too.

5.3.5 `type` attribute

The `type` attribute is used to explicitly specify the property type of the parameter to be set. If the attribute `type` is not set and the framework cannot otherwise determine the type, the type is assumed to be `StdObject`.

5.3.6 `resultMapping` attribute

The `resultMapping` attribute can be set to the name of another `resultMap` used to fill the property. If the `resultMap` is in an other mapping file, you must specified the fully qualified name as :

```
resultMapping="[namespace.sqlMap.]resultMappingId"

resultMapping="Newspaper"
<!--resultMapping with a fully qualified name.-->
resultMapping="LineItem.LineItem"
```

5.3.7 `nullValue` attribute

The `nullValue` attribute can be set to any valid value (based on property type). The `nullValue` attribute is used to specify an outgoing null value replacement. What this means is that when the value is detected in the object property, a NULL will be written to the database (the opposite behavior of an inbound null value replacement). This allows you to use a "magic" null number in your application for types that do not support null values (such as int, double, float). When these types of properties contain a matching null value (say, -9999), a NULL will be written to the database instead of the value.

If your database has a NULLABLE column, but you want your application to represent NULL with a constant value, you can specify it in the Result Map as shown in Example 5.3.1.

Example 5.3.1 *Specifying a `nullvalue` attribute in a Result Map*

```
<resultMap id="get-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="subCode" column="PRD_SUB_CODE" nullValue="-9999"/>
</resultMap>
```

In Example 5.3.1, if PRD_SUB_CODE is read as NULL, then the `subCode` property will be set to the value of -9999. This allows you to use a primitive type to represent a NULLABLE column in the database. Remember that if you want this to work for queries as well as updates/inserts, you must also specify the `nullValue` in the Parameter Map (see, Section 4.2.6).

5.3.8 `select` attribute

The `select` attribute is used to describe a relationship between objects and to automatically load complex (i.e. user defined) property types. The value of the statement property must be the name of another mapped statement. The value of the database column (the column attribute) that is defined in the same property element as this statement attribute will be passed to the related mapped statement as the parameter. More information about supported primitive types and complex property mappings/relationships is discussed later in this document. The `lazyLoad` attribute can be specified with the `select`.

5.3.9 `lazyLoad` attribute

Use the `lazyLoad` attribute with the `select` attribute to indicate whether or not the select statement's results should be lazy loaded. This can provide a performance boost by delaying the loading of the select statement's results until they are needed/accessed.

5.3.10 `typeHandler` attribute

The `typeHandler` attribute allows the use of a Custom Type Handler (see the Custom Type Handler in the following section). This allows you to extend the DataMapper's capabilities in handling types that are specific to your database provider, are not handled by your database provider, or just happen to be a part of your application design. You can create custom type handlers to deal with storing and retrieving booleans from your database for example.

5.4 Custom Type Handlers

A custom type handler allows you to extend the DataMapper's capabilities in handling types that are specific to your database provider, not handled by your database provider, or just happen to be part of your application design. The SQLMap for PHP DataMapper provides an interface, `ITypeHandlerCallback`, for you to use in implementing your custom type handler.

Example 5.4.1 *`ITypeHandlerCallback` interface*

```
interface ITypeHandlerCallback
{
    public function getParameter($object);

    public function getResult($string);

    public function createNewInstance();
}
```

The `getParameter` method allows you to process a `<statement>` parameter's value before it is added as an parameter. This enables you to do any necessary type conversion and clean-up before the DataMapper gets to work.

The `getResult` method allows you to process a database result value right after it has been retrieved by the DataMapper and before it is used in your `resultClass`, `resultMap`, or `listClass`.

The `createNewInstance` method allows the DataMapper to create new instance of a particular type handled by this callback.

One scenario where custom type handlers are useful are the when you want to use date time values in the database. First, consider a very basic TDateTime class.

```
class TDateTime
{
    private $_datetime;

    public function __construct($datetime=null)
    {
        if(!is_null($datetime))
            $this->setDatetime($datetime);
    }

    public function getTimestamp()
    {
        return strtotime($this->getDatetime());
    }

    public function getDateTime()
    {
        return $this->_datetime;
    }

    public function setDateTime($value)
    {
        $this->_datetime = $value;
    }
}
```

We can use a custom type handler to intercept result and parameter mapping that uses the say “data” as one of its property type. The handler can be written as follows.

Example 5.4.2 *A `TDateTime` Type Handler*

```
class TDateTimeHandler implements ITypeHandlerCallback
{
    public function getResult($string)
    {
        return new TDateTime($string);
    }

    public function getParameter($parameter)
    {
        if($parameter instanceof TDateTime)
            return $parameter->getTimestamp();
        else
            return $parameter;
    }

    public function createNewInstance()
    {
        return new TDateTime;
    }
}
```

With our custom type handler we can use the handler in our SqlMaps. To do that, we specify it as a basic `<typeHandler>` for all `date` types mapped in our SqlMap files

Example 5.4.3 *`<typeHandler>` in SqlMap.config*

```
[Our SqlMap.config]

<typeHandlers>
 <typeHandler type="date" callback="TDateTimeHandler"/>
</typeHandlers>


[One of our SqlMap.xml files]
```

```
<parameterMap id="boc-params">
 <parameter property="releasedDate" type="date"/>
</parameterMap>

<resultMap id="boc-result"  class="BudgetObjectCode">
 <result property="releasedDate" column="BOC_DATE" type="date"/>
</resultMap>
```

5.5 Implicit Result Maps

If the columns returned by a SQL statement match the result object, you may not need an explicit Result Map. If you have control over the relational schema, you might be able to name the columns so they also work as property names. In Example 5.5.1, the column names and property names already match, so a result map is not needed.

Example 5.5.1 *A Mapped Statement that doesn't need a Result Map*

```
<statement id="selectProduct" resultClass="Product">
  select
    id,
    description
  from PRODUCT
  where id = #value#
</statement>
```

Another way to skip a result map is to use column aliasing to make the column names match the properties names, as shown in Example 5.5.2.

Example 5.5.2 *A Mapped Statement using column aliasing instead of a Result Map*

```
<statement id="selectProduct" resultClass="Product">
  select
  PRD_ID as id,
  PRD_DESCRIPTION as description
  from PRODUCT
```

```
  where PRD_ID = #value#
</statement>
```

Of course, these techniques will not work if you need to specify a column type, a null value, or any other property attributes.

5.6 Primitive Results (i.e. String, Integer, Boolean)

Many times, we don't need to return an object with multiple properties. We just need a string, integer, boolean, and so forth. If you don't need to populate an object, SQLMap can return one of the primitive types instead. If you just need the value, you can use a primitive type as a result class, as shown in Example 5.6.1.

Example 5.6.1 *Selecting a primitive type*

```
<select id="selectProductCount" resultClass="integer">
  select count(1)
  from PRODUCT
</select>
```

Example 5.6.2 *Loading a simple list of product descriptions*

```
<resultMap id="select-product-result" resultClass="System.String">
  <result property="value" column="PRD_DESCRIPTION"/>
</resultMap>
```

5.7 Maps with ResultMaps

Instead of a rich object, sometimes all you might need is a simple key/value list of the data, where each property is an entry on the list. If so, Result Maps can populate an array instance as easily as property objects. The syntax for using an array is identical to the rich object syntax. As shown in Example 5.7.1, only the result object changes.

Example 5.7.1 *Result Maps can use arrays.*

```
<resultMap id="select-product-result" class="array">
  <result property="id" column="PRD_ID"/>
  <result property="code" column="PRD_CODE"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="suggestedPrice" column="PRD_SUGGESTED_PRICE"/>
</resultMap>
```

In Example 5.7.1, an array instance would be created for each row in the result set and populated with the Product data. The property name attributes, like `id`, `code`, and so forth, would be the key of the entry, and the value of the mapped columns would be the value of the entry.

As shown in Example 5.7.2, you can also use an implicit Result Map with an array type.

Example 5.7.2 *Implicit Result Maps can use arrays too.*

```
<statement id="selectProductCount" resultClass="array">
  select * from PRODUCT
</statement>
```

What set of entries is returned by Example 5.7.2 depends on what columns are in the result set. If the set of column changes (because columns are added or removed), the new set of entries would automatically be returned.

> **Note:** Certain providers may return column names in upper case or lower case format. When accessing values with such a provider, you will have to pass the key name in the expected case.

5.8 Complex Properties

In a relational database, one table will often refer to another. Likewise, some of your business objects may include another object or list of objects. Types that nest other types are called “complex types”. You may not want a statement to return a simple type, but a fully-formed complex type.

In the database, a related column is usually represented via a 1:1 relationship, or a 1:M relationship where the class that holds the complex property is from the “many side” of the relationship and the property itself is from the “one side” of the relationship. The column returned from the database will not be the property we want; it is a key to be used in another query.

From the framework's perspective, the problem is not so much loading a complex type, but loading each "complex property". To solve this problem, you can specify in the Result Map a statement to run to load a given property. In Example 5.8.1, the "category" property of the "select-product-result" element is a complex property.

Example 5.8.1 *A Result Map with a Complex Property*

```
<resultMap id="select-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="category" column="PRD_CAT_ID" select="selectCategory"/>
</resultMap>

<resultMap id="select-category-result" class="category">
  <result property="id" column="CAT_ID"/>
  <result property="description" column="CAT_DESCRIPTION"/>
</resultMap>

<select id="selectProduct" parameterClass="int" resultMap="select-product-result">
 select * from PRODUCT where PRD_ID = #value#
</select>

<select id="selectCategory" parameterClass="int" resultMap="select-category-result">
 select * from CATEGORY where CAT_ID = #value#
</select>
```

In Example 5.8.1, the framework will use the "selectCategory" statement to populate the "category" property. The value of each category is passed to the "selectCategory" statement, and the object returned is set to the category property. When the process completes, each Product instance will have the the appropriate category object instance set.

5.9 Avoiding N+1 Selects (1:1)

A problem with Example 5.8.1 may be that whenever you load a Product, two statements execute: one for the Product and one for the Category. For a single Product, this issue may seem trivial. But if you load

10 products, then 11 statements execute. For 100 Products, instead of one statement product statement executing, a total of 101 statements execute. The number of statements executing for Example 5.9.1 will always be N+1: 100+1=101.

Example 5.9.1 *N+1 Selects (1:1)*

```
<resultMap id="select-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="category" column="PRD_CAT_ID" select="selectCategory"/>
</resultMap>

<resultMap id="select-category-result" class="category">
  <result property="id" column="CAT_ID"/>
  <result property="description" column="CAT_DESCRIPTION"/>
</resultMap>

<!-- This statement executes 1 time -->
<select id="selectProducts" parameterClass="int" resultMap="select-product-result">
 select * from PRODUCT
</select>

<!-- This statement executes N times (once for each product returned above) -->
<select id="selectCategory" parameterClass="int" resultMap="select-category-result">
 select * from CATEGORY where CAT_ID = #value#
</select>
```

One way to mitigate the problem is to cache the "selectCategory" statement . We might have a hundred products, but there might only be five categories. Instead of running a SQL query or stored procedure, the framework will return the category object from it cache. A 101 statements would still run, but they would not be hitting the database. (See Chapter 6 for more about caches.)

Another solution is to use a standard SQL join to return the columns you need from the another table. A join can bring all the columns we need over from the database in a single query. When you have a nested object, you can reference nested properties using a dotted notation, like "category.description".

Example 5.9.2 solves the same problem as Example 5.9.1, but uses a join instead of nested properties.

Example 5.9.2 *Resolving complex properties with a join*

```
<resultMap id="select-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="category" resultMapping="Category.CategoryResult" />
</resultMap>

<statement id="selectProduct" parameterClass="int" resultMap="select-product-result"
  select *
  from PRODUCT, CATEGORY
  where PRD_CAT_ID=CAT_ID
  and PRD_ID = #value#
</statement>
```

> **Lazy Loading vs. Joins (1:1):** It's important to note that using a join is not always better. If you are in a situation where it is rare to access the related object (e.g. the category property of the Product class) then it might actually be faster to avoid the join and the unnecessary loading of all category properties. This is especially true for database designs that involve outer joins or nullable and/or non-indexed columns. In these situations it might be better to use the sub-select solution with lazy loading enabled. The general rule of thumb is: use the join if you're more likely going to access the associated properties than not. Otherwise, only use it if lazy loading is not an option.
>
> If you're having trouble deciding which way to go, don't worry. No matter which way you go, you can always change it without impacting your application source code. Example 5.9.1 and 5.9.2 result in exactly the same object graph and are loaded using the exact same method call from the application. The only consideration is that if you were to enable caching, then the using the separate select (not the join) solution could result in a cached instance being returned. But more often than not, that won't cause a problem (your application shouldn't be dependent on instance level equality i.e. "===").

5.10 Complex Collection Properties

It is also possible to load properties that represent lists of complex objects. In the database the data would be represented by a M:M relationship, or a 1:M relationship where the class containing the list is on the "one side" of the relationship and the objects in the list are on the "many side". To load a `TList` of objects,

there is no change to the statement (see example above). The only difference required to cause the SQLMap DataMapper framework to load the property as a `TList` is that the property on the business object must be of type `TList`. For example, if a Category has a `TList` of Product instances, the mapping would look like this (assuming Category has a property called "ProductList" of `TList`.):

Example 5.10.1 *Mapping that creates a list of complex objects*

```
<resultMaps>

  <resultMap id="select-category-result" class="Category">
    <result property="Id" column="CAT_ID"/>
    <result property="Description" column="CAT_DESCRIPTION"/>
    <result property="ProductList" column="CAT_ID" select="selectProductsByCatId"/>
  </resultMap>

  <resultMap id="select-product-result" class="Product">
    <result property="Id" column="PRD_ID"/>
    <result property="Description" column="PRD_DESCRIPTION"/>
  </resultMap>
<resultMaps>

<statements>

  <statement id="selectCategory" parameterClass="int"
             resultMap="select-category-result">
    select * from CATEGORY where CAT_ID = #value#
  </statement>

  <statement id="selectProductsByCatId" parameterClass="int"
             resultMap="select-product-result">
    select * from PRODUCT where PRD_CAT_ID = #value#
  </statement>
</statements>
```

5.11 Avoiding N+1 Select Lists (1:M and M:N)

This is similar to the 1:1 situation above, but is of even greater concern due to the potentially large amount of data involved. The problem with the solution above is that whenever you load a Category, two SQL statements are actually being run (one for the Category and one for the list of associated Products). This problem seems trivial when loading a single Category, but if you were to run a query that loaded ten (10) Categories, a separate query would be run for each Category to load its associated list of Products. This results in eleven (11) queries total: one for the list of Categories and one for each Category returned to load each related list of Products (N+1 or in this case 10+1=11). To make this situation worse, we're dealing with potentially large lists of data.

Example 5.11.1 *N+1 Select Lists (1:M and M:N)*

```
<resultMaps>

  <resultMap id="select-category-result" class="Category">
    <result property="Id" column="CAT_ID"/>
    <result property="Description" column="CAT_DESCRIPTION"/>
    <result property="ProductList" column="CAT_ID" select="selectProductsByCatId"/>
  </resultMap>

  <resultMap id="select-product-result" class="Product">
    <result property="Id" column="PRD_ID"/>
    <result property="Description" column="PRD_DESCRIPTION"/>
  </resultMap>
<resultMaps>

<statements>

  <!-- This statement executes 1 time -->
  <statement id="selectCategory" parameterClass="int"
             resultMap="select-category-result">
    select * from CATEGORY where CAT_ID = #value#
  </statement>

  <!-- This statement executes N times (once for each category returned above)
```

```
      and returns a list of Products (1:M) -->
  <statement id="selectProductsByCatId" parameterClass="int"
           resultMap="select-product-result">
    select * from PRODUCT where PRD_CAT_ID = #value#
  </statement>
</statements>
```

5.11.1 1:N & M:N Solution?

Currently the feature that resolves this issue not implemented, but the development discussions are active, and we expect it to be included in a future release.

> **Lazy Loading vs. Joins (1:M and M:N):** As with the 1:1 situation described previously, it's important to note that using a join is not always better. This is even more true for collection properties than it was for individual value properties due to the greater amount of data. If you are in a situation where it is rare to access the related object (e.g. the ProductList property of the Category class) then it might actually be faster to avoid the join and the unnecessary loading of the list of products. This is especially true for database designs that involve outer joins or nullable and/or non-indexed columns. In these situations it might be better to use the sub-select solution with the lazy loading. The general rule of thumb is: use the join if you're more likely going to access the associated properties than not. Otherwise, only use it if lazy loading is not an option.
>
> As mentioned earlier, if you're having trouble deciding which way to go, don't worry. No matter which way you go, you can always change it without impacting your PHP code. The two examples above would result in exactly the same object graph and are loaded using the exact same method call. The only consideration is that if you were to enable caching, then the using the separate select (not the join) solution could result in a cached instance being returned. But more often than not, that won't cause a problem (your application should not be dependent on instance level equality i.e. "===").

5.12 Composite Keys or Multiple Complex Parameters Properties

You might have noticed that in the above examples there is only a single key being used as specified in the `resultMap` by the `column` attribute. This would suggest that only a single column can be associated to a related mapped statement. However, there is an alternate syntax that allows multiple columns to be passed to the related mapped statement. This comes in handy for situations where a composite key relationship

exists, or even if you simply want to use a parameter of some name other than `#value#`. The alternate syntax for the column attribute is simply {param1=column1, param2=column2, $\cdots$, paramN=columnN}. Consider the example below where the PAYMENT table is keyed by both Customer ID and Order ID:

Example 5.12.1 *Mapping a composite key*

```
<resultMaps>
  <resultMap id="select-order-result" class="order">
    <result property="id" column="ORD_ID"/>
    <result property="customerId" column="ORD_CST_ID"/>
    ...
    <result property="payments" column="{itemId=ORD_ID, custId=ORD_CST_ID}"
      select="selectOrderPayments"/>
  </resultMap>
<resultMaps>

<statements>

  <statement id="selectOrderPayments" resultMap="select-payment-result">
    select * from PAYMENT
    where PAY_ORD_ID = #itemId#
    and PAY_CST_ID = #custId#
  </statement>
</statements>
```

Optionally you can just specify the column names as long as they're in the same order as the parameters. For example:

```
{ORD_ID, ORD_CST_ID}
```

> **Important!** Currently the SQLMap DataMapper framework does not automatically resolve circular relationships. Be aware of this when implementing parent/child relationships (trees). An easy work around is to simply define a second result map for one of the cases that does not load the parent object (or vice versa), or use a join as described in the "N+1 avoidance" solutions.

Note: Result Map names are always local to the Data Map definition file that they are defined in. You can refer to a Result Map in another Data Map definition file by prefixing the name of the Result Map with the namespace of the SqlMap set in the `<sqlMap>` root element.

Chapter 6

Cache Models

Some values in a database are know to change slower than others. To improve performance, many developers like to cache often-used data to avoid making unnecessary trips back to the database. SQLMap provides its own caching system, that you configure through a `<cacheModel>` element.

The results from a query Mapped Statement can be cached simply by specifying the `cacheModel` parameter in the statement tag (seen above). A cache model is a configured cache that is defined within your DataMapper configuration file. Cache models are configured using the `cacheModel` element as follows:

Example 6.0.2 *Configuring a cache using the Cache Model element*

```
<cacheModel id="product-cache" implementation="LRU" >
  <flushInterval hours="24"/>
  <flushOnExecute  statement="insertProduct"/>
  <flushOnExecute  statement="updateProduct"/>
  <flushOnExecute  statement="deleteProduct"/>
  <property name="CacheSize" value="100"/>
</cacheModel>
```

The cache model above will create an instance of a cache named “product-cache” that uses a Least Recently Used (LRU) implementation. The value of the `type` attribute is either a class name, or an alias for one of the included implementations (see below). Based on the flush elements specified within the cache model, this cache will be flushed every 24 hours. There can be only one flush interval element and it can be

set using hours, minutes, seconds or milliseconds. In addition the cache will be flushed whenever the `insertProduct`, `updateProduct`, or `deleteProduct` mapped statements are executed. There can be any number of "flush on execute" elements specified for a cache. Some cache implementations may need additional properties, such as the "cache-size" property demonstrated above. In the case of the LRU cache, the size determines the number of entries to store in the cache. Once a cache model is configured, you can specify the cache model to be used by a mapped statement, for example:

Example 6.0.3 *Specifying a Cache Model from a Mapped Statement*

```
<statement id="getProductList" cacheModel="product-cache">
  select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

6.1 Cache Implementation

The cache model uses a pluggable framework for supporting different types of caches. The choice of cache is specified in the "implementation" attribute of the `cacheModel` element as discussed above. The class name specified must be an implementation of the `ISqlMapCache` interface, or one of the two aliases discussed below. Further configuration parameters can be passed to the implementation via the property elements contained within the body of the `cacheModel`. Currently there are 2 implementations included with the PHP distribution.

6.1.1 Least Recently Used [LRU] Cache

The LRU cache implementation uses an Least Recently Used algorithm to determines how objects are automatically removed from the cache. When the cache becomes over full, the object that was accessed least recently will be removed from the cache. This way, if there is a particular object that is often referred to, it will stay in the cache with the least chance of being removed. The LRU cache makes a good choice for applications that have patterns of usage where certain objects may be popular to one or more users over a longer period of time (e.g. navigating back and forth between paginated lists, popular search keys etc.).

The LRU implementation is configured as follows:

Example 6.1.1 *Configuring a LRU type cache*

```
<cacheModel id="product-cache"  implementation="LRU" >
  <flushInterval hours="24"/>
  <flushOnExecute  statement="insertProduct"/>
  <flushOnExecute  statement="updateProduct"/>
  <flushOnExecute  statement="deleteProduct"/>
   <property name="CacheSize" value="100"/>
</cacheModel>
```

Only a single property is recognized by the LRU cache implementation. This property, named `CacheSize` must be set to an integer value representing the maximum number of objects to hold in the cache at once. An important thing to remember here is that an object can be anything from a single string instance to an array of object. So take care not to store too much in your cache and risk running out of memory and disk space.

6.1.2 FIFO Cache

The FIFO cache implementation uses an First In First Out algorithm to determines how objects are automatically removed from the cache. When the cache becomes over full, the oldest object will be removed from the cache. The FIFO cache is good for usage patterns where a particular query will be referenced a few times in quick succession, but then possibly not for some time later.

The FIFO implementation is configured as follows:

Example 6.1.2 *Configuring a FIFO type cache*

```
<cacheModel id="product-cache" implementation="FIFO" >
  <flushInterval hours="24"/>
  <flushOnExecute  statement="insertProduct"/>
  <flushOnExecute  statement="updateProduct"/>
  <flushOnExecute  statement="deleteProduct"/>
  <property name="CacheSize" value="100"/>
</cacheModel>
```

Only a single property is recognized by the FIFO cache implementation. This property, named `CacheSize` must be set to an integer value representing the maximum number of objects to hold in the cache at once. An important thing to remember here is that an object can be anything from a single String instance to an

array of object. So take care not to store too much in your cache and risk running out of memory and disk space.

Chapter 7

Dynamic SQL

Not supported in this release.

Chapter 8

Installation and Setup

8.1 Introduction

This section explains how to install, configure, and use the SQLMap DataMapper with your PHP application.

8.2 Installing the DataMapper for PHP

There are two steps to using SQLMap DataMapper with your application for the first time.

- Setup the distribution
- Add XML documents

8.2.1 Setup the Distribution

The official site for SQLMap DataMapper for PHP is http://... . The DataMapper is availabe as a source distribution in the form of a ZIP archive. To download the distributions, follow the link to the Downloads area on the web site, and select the the source distribution for the SQLMap PHP DataMapper release. You can extract the distribution using a utility like WinZip or the extractor built into newer versions of Windows.

Under the distribution's source folder are eight folders that make up the SQLMap PHP DataMapper distribution, as shown in Table 4.1.

Table 8.1: Folders found in the SQLMap PHP DataMapper source distribution

Folder name	Description

8.2.2 Add XML file items

After unpacking the source distribution, you will need to add two types of XML files to your Web application, or library project (and Test project if you have one). These files are:

SqlMap.xml – A Data Map file that contains your SQL queries. Your project will contain one or more of these files with names such as Account.xml or Product.xml.

SqlMap.config – The DataMapper configuration file that is used to specify the locations of your SqlMap.xml files. It is also used to define other DataMapper configuration options such as caching. You will need to include one SqlMap.config file for each data source that your project has.

As expected, the `SqlMap.config` file must be placed where the DataMapper can find them at runtime.

8.3 Configuring the DataMapper for PHP

The SQLMap PHP DataMapper is configured using a central XML descriptor file, usually named `SqlMap.config`, which provides the details for your data source, data maps, and other features like caching, and transactions. At runtime, your application code will call a class method provided by the SQLMap library to read and parse your `SqlMap.config` file. After parsing the configuration file, a DataMapper client will be returned by SQLMap for your application to use.

8.3.1 DataMapper clients

Currently, the SQLMap PHP DataMapper framework revolves around the `TSqlMapper` class, which acts as a facade to the DataMapper framework API. You can create a DataMapper client by instantiating an object of the `TSqlMapper` class. An instance of the `TSqlMapper` class (your DataMapper client) is created by reading a single configuration file. Each configuration file can specify one database or data source.

You can of couse use multiple DataMapper clients in your application. Just create another configuration file and pass the name of that file when the DataMapper client is created. The configuration files might use a different account with the same database, or reference different databases on different servers. You can read from one client and write to another, if that's what you need to do. See Section **??** for more details on building a `TSqlMapper` instance, but first, let's take a look at the DataMapper configuration file.

8.4 DataMapper Configuration File (SqlMap.config)

A sample configuration file for a PHP web application is shown in Example 8.4.1. Not all configuration elements are required. The following sections describe the elements of this `SqlMap.config` file in more detail.

Example 8.4.1 *Sample SqlMap.Config for a PHP Web Application.*

```
<?xml version="1.0" encoding="utf-8"?>
<sqlMapConfig>

    <provider class="TAdodbProvider" >
        <datasource ConnectionString="mysql://user:pass@localhost/test1" />
    </provider>

    <sqlMaps>
        <sqlMap name="Account" resource="maps/Account.xml"/>
        <sqlMap name="Order" resource="maps/Order.xml"/>
        <sqlMap name="Category" resource="maps/Category.xml"/>
        <sqlMap name="LineItem" resource="maps/LineItem.xml"/>
    </sqlMaps>

</sqlMapConfig>
```

8.5 DataMapper Configuration Elements

Sometimes the values we use in an XML configuration file occur in more than one element. Often, there are values that change when we move the application from one server to another. To help you manage

configuration values, you can specify a standard properties file (with name=value entries) as part of a DataMapper configuration. Each named value in the properties file becomes a shell variable that can be used in the DataMapper configuration file and your Data Map definition files (see Chapter 3).

8.5.1 <properties> attributes

The <properties> element can accept one resource attribute to specify the location of the properties file.

Table 8.2: Attributes of the <properties> element

Attribute	Description
resource	Specify the properties file to be loaded from the directory relative to the current file. resource="properties.config"

For example, if the "properties.config" file contains

```
<?xml version="1.0" encoding="utf-8" ?>
<settings>
  <add key="username" value="albert" />
</settings>
```

then all elements in the DataMapper configuration can use the variable $username to insert the value "albert". For example:

```
<provider ConnectionString="mysql://${username}:..."
```

Properties are handy during building, testing, and deployment by making it easy to reconfigure your application for multiple environments.

8.5.2 <property> element and attributes

You can also specify more than one properties file or add property keys and values directly into your SqlMap.config file by using <property> elements. For example:

```
<properties>
 <property resource="myProperties.config"/>
 <property resource="anotherProperties.config"/>
 <property key="host" value="ibatis.com" />
</properties>
```

Table 8.3: Attributes of the <property> element

Attribute	Description
resource	Specify the properties file to be loaded from the directory relative to the current file. resource="properties.config"
key	Defines a property key (variable) name key="username"
value	Defines a value that will be used by the DataMapper in place of the the specified property key/variable value="mydbuser"

8.5.3 The <typeHandler> Element

The <typeHandler> element allows for the configuration and use of a Custom Type Handler (see the Custom Type Handler section). This extends the DataMapper's capabilities in handling types that are specific to your database provider, are not handled by your database provider, or just happen to be a part of your application design.

```
<typeHandler type="date" callback="TDateTimeHandler"/>
```

The <typeHandler> element has three attributes:

8.5.4 The <provider> element and attribute

The <provider> element encloses a <datasource> that configure the database system for use by the framework.

Table 8.4: Attributes of the <typeHandler> element

Attribute	Description
type	Refers to the name of the type to handle type="date"
dbType	Indicates the provider dbType to handle dbType="Varchar2"
callback	The custom type handler class name callback="TDateTimeHandler"

Table 8.5: Attributes of the <provider> element

Attribute	Description
class	The database provider class that extends TDatabaseProvider. class="TAdodbProvider"

8.5.5 The <datasource> element and attributes

The <datasource> element specifies the connection string. Example 8.5.1 shows sample element MySql.

Example 8.5.1 *Sample <provider> and <datasource> elements.*

```
<!-- The ${properties} are defined in an external file, -->
<!-- but the values could also be coded inline. -->

<!-- Connecting to a MySQL database -->
<provider class="TAdodbProvider" >
  <datasource
    ConnectionString="mysql://${username}:${password}@${host}/${database}" />
</provider>
```

> **Tip:** Use Data Source Name (DSN) connection string or specify the necessary individual connection parameters.

Table 8.6: Attributes of the <datasource> element

Attribute	Description
connectionString	Data Source Name (DSN) connection string. connectionString="mysql://root:pwd@localhost/mydb"
driver	Database driver name (mysql, sqlite, etc.) driver="mysql"
host	DB host name/IP (and port number) in the format host[:port]. connectionString="mysql://root:pwd@localhost/mydb"
username	Database connection username.
password	Database connection password.
database	Database name to use in the connection.

8.5.6 The <sqlMap> Element

On a daily basis, most of your work will be with the Data Maps, which are covered by Chapter 3. The Data Maps define the actual SQL statements or stored procedures used by your application. The parameter and result objects are also defined as part of the Data Map. As your application grows, you may have several varieties of Data Map. To help you keep your Data Maps organized, you can create any number of Data Map definition files and incorporate them by reference in the DataMapper configuration. All of the definition files used by a DataMapper instance must be listed in the configuration file.

Example 8.5.2 shows <sqlMap> elements for loading a set of Data Map definitions. For more about Data Map definition files, see Chapter 3.

Example 8.5.2 *Specifying sqlMap locations*

```
<!-- Relative path from the directory of the
     current file using a property variable -->
<sqlMap resource="${root}/Maps/Account.xml"/>
<sqlMap resource="${root}/Maps/Category.xml"/>
<sqlMap resource="${root}/Maps/Product.xml"/>

<!-- Full file path with a property variable -->
<sqlMap resource="/${projectdir}/MyApp/Maps/Account.xml"/>
```

```
<sqlMap resource="/${projectdir}/MyApp/Maps/Category.xml"/>
<sqlMap resource="/${projectdir}/MyApp/Maps/Product.xml"/>
```

> **Tip:** Since the application root directory location differs by project type (Windows, Web, or library), it is best to use a properties variable to indicate the relative path when using the `<sqlMap> resource` attribute. Having a variable defined in a properties file makes it easy to change the path to all your Data Mapper configuration resources in one location (note the `${projectdir}` and `${root}` variables in the example above).

Chapter 9

Using SQLMap PHP DataMapper

The SQLMap DataMapper API provides four core functions:

- build a `TSqlMapper` instance from a configuration file or cache
- execute an update query (including insert and delete).
- execute a select query for a single object
- execute a select query for a list of objects

The API also provides support for retrieving paginated lists and managing transactions.

9.1 Building a `TSqlMapper` instance

An XML document is a wonderful tool for describing a database configuration (Chapter 8) or defining a set of data mappings (Chapter 3), but you can't execute XML. In order to use the SQLMap configuration and definitions in your PHP application, you need a class you can call.

The framework provides service methods that you can call which read the configuration file (and any of its definition files) and builds a `TSqlMapper` object. The `TSqlMapper` object provides access to the rest of the framework. Example 9.1.1 shows a singleton Mapper that is similar to the one bundled with the framework.

Example 9.1.1 *A Mapper singleton you can call from your own applications*

```
require_once('/path/to/SQLMap/TSqlMapper.php');

class TMapper
{
    private static $_mapper;

    public static function configure($configFile)
    {
        if(is_null(self::$_mapper))
        {
            $builder = new TDomSqlMapBuilder();
            self::$_mapper = $builder->configure($configFile);
        }
        return self::$_mapper;
    }

    public static function instance()
    {
        return self::$_mapper;
    }
}
```

To obtain the `TSqlMapper` instance, first configure the mapper once,

```
TMapper::configure('path/to/sqlmap.config');
```

The `TDomSqlMapBuilder` object will go throught the the `sqlmap.config` file and build a `TSqlMapper` instance. To use `TSqlMapper` in your application, specify one of the `TSqlMapper` methods (see Section ???). Here's an example:

```
$list = TMapper::instance()->queryForList("PermitNoForYearList", $values);
```

9.1.1 Multiple Databases

If you need access to more than one database from the same application, create a DataMapper configuration file for that database and another Mapper class to go with it.

9.1.2 `TDomSqlMapBuilder` Configuration Options

If you find that you already have loaded your DataMapper configuration information as a `SimpleXMLElement` instance within your application, the `TDomSqlMapBuilder` provides `Configure` overloads for those types as well.

9.2 Exploring the SQLMap PHP DataMapper API through the `TSqlMapper`

The `TSqlMapper` instance acts as a facade to provide access the rest of the DataMapper framework. The DataMapper API methods are shown in Example 4.11.

Example 9.2.1 *The SQLMap DataMapper API for PHP.*

```
/* Query API */
public function queryForObject($statementName, $parameter=null, $result=null);
public function queryForList($statementName, $parameter=null, $result=null,
                                    $skip=-1, $max=-1);
public function queryForPagedList($statementName, $parameter=null, $pageSize=10);
public function queryForMap($statementName, $parameter=null,
                                    $keyProperty=null, $valueProperty=null);

public function insert($statementName, $parameter=null)
public function update($statementName, $parameter=null)
public function delete($statementName, $parameter=null)

/* Connection API */
public function openConnection()
public function closeConnection()
```

```
/* Transaction API */
public function beginTransaction()
public function commitTransaction()
public function rollBackTransaction()
```

Note that each of the API methods accept the name of the Mapped Statement as the first parameter. The `statementName` parameter corresponds to the `id` of the Mapped Statement in the Data Map definition (see Section **??**). In each case, a `parameterObject` also may be passed. The following sections describe how the API methods work.

9.2.1 Insert, Update, Delete

```
public function insert($statementName, $parameter=null)
public function update($statementName, $parameter=null)
public function delete($statementName, $parameter=null)
```

If a Mapped Statement uses one of the `<insert>`, `<update>`, or `<delete>` statement-types, then it should use the corresponding API method. The `<insert>` element supports a nested `<selectKey>` element for generating primary keys (see Section 3.4). If the `<selectKey>` stanza is used, then `insert` returns the generated key; otherwise a null object is returned. Both the `update` and `delete` methods return the number of rows affected by the statement.

9.2.2 QueryForObject

```
public function queryForObject($statementName, $parameter=null, $result=null);
```

If a Mapped Statement is expected to select a single row, then call it using `queryForObject`. Since the Mapped Statement definition specifies the result class expected, the framework can both create and populate the result class for you. Alternatively, if you need to manage the result object yourself, say because it is being populated by more than one statement, you can use the alternate form and pass your `$resultObject` as the third parameter.

9.2.3 QueryForList

```
public function queryForList($statementName, $parameter=null, $result=null,
```

```
                                        $skip=-1, $max=-1);
```

If a Mapped Statement is expected to select multiple rows, then call it using `queryForList`. Each entry in the list will be an result object populated from the corresponding row of the query result. If you need to manage the `$resultObject` yourself, then it can be passed as the third parameter. If you need to obtain a partial result, the fourth parameter `$skip` and fifth parameter `$max` allow you to skip a number of records (the starting point) and the maximum number to return.

9.2.4 QueryForPagedList

```
public function queryForPagedList($statementName, $parameter=null, $pageSize=10);
```

We live in an age of information overflow. A database query often returns more hits than users want to see at once, and our requirements may say that we need to offer a long list of results a "page" at a time. If the query returns 1000 hits, we might need to present the hits to the user in sets of fifty, and let them move back and forth between the sets. Since this is such a common requirement, the framework provides a convenience method.

The `TSqlMapPagedList` interface includes methods for navigating through pages (`nextPage()`, `previousPage()`, `gotoPage($pageIndex)`) and also checking the status of the page (`getIsFirstPage()`, `getIsMiddlePage()`, `getIsLastPage()`, `getIsNextPageAvailable()`, `getIsPreviousPageAvailable()`, `getCurrentPageInd` `getPageSize()`). The total number of records available is not accessible from the `TSqlMapPagedList` interface, unless a virtual count is defined using `setVirtualCount($value)`, this should be easily accomplished by simply executing a second statement that counts the expected results.

> **Tip:** The `queryForPagedList` method is convenient, but note that a larger set (up to 3 times the page size) will first be returned by the database provider and the smaller set extracted by the framework. The higher the page size, the larger set that will be returned and thrown away. For very large sets, you may want to use a stored procedure or your own query that uses `$skip` and `$max` as parameters in `queryForList`.

9.2.5 QueryForMap

```
public function queryForMap($statementName, $parameter=null,
                                        $keyProperty=null, $valueProperty=null);
```

The `queryForList` methods return the result objects within a `TList` or array instance. Alternatively, the `queryForMap` returns a TMap or associative array instance. The value of each entry is one of the result objects. The key to each entry is indicated by the `$keyProperty` parameter. This is the name of the one of the properties of the result object, the value of which is used as the key for each entry. For example, If you needed a set of `Employee` objects, you might want them returned as a `TMap` keyed by each object's `EmployeeNumber` property.

If you don't need the entire result object in your result, you can add the `$valueProperty` parameter to indicate which result object property should be the value of an entry. For example, you might just want the `EmployeeName` keyed by `EmployeeNumber`.

9.2.6 Transaction

The DataMapper API includes methods to demarcate transactional boundaries. A transaction can be started, committed and/or rolled back. You can call the transaction methods from the `TSqlMapper` instance.

```
// Begin a transactional session using Adodb transaction API
public function beginTransaction()

// Commit a transaction, uses Adodb transaction API
public function commitTransaction()

// RollBack a transaction, uses Adodb transaction API
public void RollBackTransaction()
```

Example 9.2.2 *Using transactions*

```
try
{
    $sqlMap->beginTransaction();
    $item = $sqlMap->queryForObject("getItem", $itemId);
    $item->setDescription($newDescription);
    $sqlMap->update("updateItem", $item);
    $sqlMap->commitTransaction();
}
catch
```

```
{
    $sqlMap->rollBackTransaction();
}
```

9.3 Coding Examples

Example 9.3.1 *Executing Update (insert, update, delete)*

```
$product = new Product();
$product->setId(1);
$product->setDescription('Shih Tzui');

$key = $sqlMap->insert('insertProduct', $product);
```

Example 9.3.2 *Executing Query for Object (select)*

```
$key = 1;
$product = $sqlMap->queryForObject ('getProduct', $key);
```

Example 9.3.3 *Executing Query for Object (select) With Preallocated Result Object*

```
$customer = new Customer();

$sqlMap->beginTransaction();

$sqlMap->queryForObject('getCust', $parameter, $customer);
$sqlMap->queryForObject('getAddr', $parameter, $customer);
$sqlMap->commitTransaction();
```

Example 9.3.4 *Executing Query for List (select)*

```
$list = $sqlMap->queryForList ('getProductList');
```

Example 9.3.5 *Executing Query for List (select) With Result Boundaries*

```
$list = $sqlMap->queryForList ('getProductList', $key, null, 0, 40);
```

Example 9.3.6 *Executing Query for Paginated List (select)*

```
$list = $sqlMap->queryForPagedList ('getProductList', null, 10);
$list->nextPage();
$list->previousPage();
```

Example 9.3.7 *Executing Query for Map*

```
 $map = $sqlMap->QueryForMap('getProductList', null, 'productCode');
 $product = $map['EST-93'];
```