Nanorm Developer's Guide

Contents

Chapter 1

Introduction

The Nanorm Data Mapper framework helps you to omit a lot of book-keeping code otherwise required to access a relational database. Nanorm is nothing more than a simple mapper from Java Bean to the SQL statement parameters and SQL ResultSet to the JavaBean. The key points are simplicity combined with Java type-checking. The data access layer built using this framework is simply a number of mapper interfaces, the regular Java interfaces marked with Nanorm annotations. This approach enables easy unit-testing and allows the framework to perform all required type checks.

1.1 Concept

The Nanorm API provides most common JDBC functionality with much less code required from the developer, providing proper type safety and little performance overhead at the same time.

1.2 Nanorm principles

Nanorm is built around the concept of *mapper* , regular Java interface marked with certain Nanorm annotations. The idea is that developer writes the interface with methods formulated in terms of database queries (each method representing a single database query, probably with subqueries) and annotates them with query SQL and the results mapping. Behind the scenes, the Nanorm parses all configuration from given set of interfaces and provides the instances of these interfaces. By invoking the methods on provided instance, the application can make database queries. The typical query works like the following:

1. The application prepares the array of query parameters (which could include Java Beans, primitive types or primitive wrappers).
2. The application invokes a method on the mapper interface, passing the parameters.
3. Nanorm on its side maps method parameters to the JDBC PreparedStatement instance and executes the query.
4. The query result (either JDBC `ResultSet` or integer representing the number of rows affected) is mapped to the declared method return value (which could be primitive type, Java Bean, collection or primitive wrapper).

1.3 Installation

To install the Nanorm you need to put certain JAR files to the application classpath. This can be the classpath specefied to the java binary (java -classpath argument) or in case of web application this will be the /WEB-INF/lib/ directory of the webapp. Refer to the following resources for details:

- http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/classpath.html

- http://java.sun.com/docs/books/tutorial/java/package/managingfiles.html

The following table summarizes the dependencies of the library:

Table 1.1: nanorm dependencies

Dependency	Required?	Description
nanorm-0.1.0.jar	Yes	Core Nanorm JAR library Homepage: http://nanorm.googlecode.com/
slf4j-api-1.5.2.jar	Yes	SLF4J Logging library API, used for logging. Homepage: http://slf4j.org
asm-all-3.1.jar	No	ASM byte-code manipulation library. Used for generating bytecode for properties access. If not provided, Nanorm uses reflection for accessing the properties (which is slower). Homepage: http://asm.objectweb.org/

If you use Maven2 as a build system, you only need to add Nanorm dependency and repository definition to your pom.xml. See Maven2 pom.xml example for sample configuration snippet.

Chapter 2

Using Nanorm

2.1 Getting started

Let's start with a simple one-table "Librarian" application. We start with a model bean describing the primary entity of our application, a book. First, we describe the book entity using the Java Bean with four properties: id, name, author and published (see code snippet). Then we declare a mapper interface with CRUD methods and this is there Nanorm comes to scene. The interface implementation will be provided by the Nanorm.

```
package sample;

  import java.util.List;

  import com.google.code.nanorm.annotations.ResultMap;
  import com.google.code.nanorm.annotations.Insert;
  import com.google.code.nanorm.annotations.Update;
  import com.google.code.nanorm.annotations.Select;

  @ResultMap(id = "book", auto = "true")
  public interface BookMapper {
      @Insert("INSERT INTO books(id, name, author, published) " +
        "VALUES(${1.id}, ${1.name}, ${1.author}, ${1.published})")
      void insertBook(Book book);

      @ResultMapRef("book")
      @Select("SELECT id, name, author, published FROM books")
      List<Book> selectBooks();

      @Update("UPDATE books SET name = ${1.name}, author = ${1.author}, " +
        "published = ${1.published} WHERE id = ${1.id}")
      int updateBook(Book book);

      @Update("DELETE FROM books WHERE id = ${1}")
      int deleteBook(int id);
  }
```

The listing is almost self-describing. First, we declare a result map which we will use for the `selectBooks` method. The `auto=true` tells the Nanorm that the mapping should be generated automatically, based on the method return value and `ResultSet` metadata. In fact, we can completely omit the result mapping in that case and Nanorm will generate it automatically after the first query, based on the `ResultSet` metadata.

The `insertBook` method is annotated with `@Insert` which defines the SQL statement to execute when this method is called. The SQL can contain references to the method parameters. The reference has the syntax `${<property>`, where `<property>` is property path. First part of the property path identifies the parameter, which could be a number, like `1` or `value`. The

number identifies parameter by its position (the parameters are numbered from 1) and `value` is the first parameter. The rest part of the property path is property names or indexing operators separated by `.`(dot). See Property Paths for more details about property paths.

When `insertBook` method is invoked, the Nanorm creates the prepared statement using the provided SQL, with all references replaced by positional parameters `?`, and binds these parameters to values retrieved from the method parameters using property paths provided.

The `updateBook` and deleteBook methods work the same way as `insertMethod`, but with one small difference. Since the returnvalue of the methods are declared as int, the methods implementations will return amount of rows updated/deleted.

Note int type for the methods marked by `@Insert` annotation has the different meaning. Refer to the Generated Keys Support for more details.

Chapter 3

Reference

3.1 Property Paths

TBD

3.2 Generated Keys Support

TBD

Appendix A

Maven2 pom.xml example

```
<project>
  ...
  <repositories>
    ...
    <repository>
      <id>nanorm-releases-repo</id>
      <name>Repository on code.google.com</name>
      <url>http://nanorm.googlecode.com/svn/maven2/releases/</url>
      <snapshots><enabled>false</enabled></snapshots>
    </repository>
    ...
  </repositories>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>com.google.code.nanorm</groupId>
      <artifactId>nanorm</artifactId>
      <version>0.1.0</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Appendix B

Code Samples

B.1 Book Java Bean

```
package sample;

import java.sql.Date;

public class Book {
    private int id;

    private String name;

    private String author;

    private Date published;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setId(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```