

# **Nanorm Developer's Guide**

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Concept . . . . .	1
1.2	Nanorm principles . . . . .	1
1.3	Installation . . . . .	1
<b>2</b>	<b>Using Nanorm</b>	<b>3</b>
2.1	Getting started . . . . .	3
2.2	Using the Mapper . . . . .	4
2.3	Autogenerated keys . . . . .	5
2.4	Mapping the collections . . . . .	6
<b>3</b>	<b>Reference</b>	<b>8</b>
3.1	Property Paths . . . . .	8
<b>A</b>	<b>Maven2 pom.xml example</b>	<b>9</b>
<b>B</b>	<b>Code Samples</b>	<b>10</b>
B.1	Book Java Bean . . . . .	10

# 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 specified 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: <a href="http://nanorm.googlecode.com/">http://nanorm.googlecode.com/</a>
slf4j-api-1.5.2.jar	Yes	SLF4J Logging library API, used for logging. Homepage: <a href="http://slf4j.org">http://slf4j.org</a>
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: <a href="http://asm.objectweb.org/">http://asm.objectweb.org/</a>

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 where Nanorm comes to the scene. The interface implementation will be provided by the Nanorm.

```
package sample;

import java.util.List;

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

@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 WHERE id = ${1}")
    Book selectBook(int id);

    @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 `selectBook` 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 is a number. This number identifies parameter

by its position (the parameters are numbered from 1). 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 difference related to the return value. Since the return value 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 [Autogenerated keys](#) for more details.

---

Finally, the query preparation and execution for the `selectBooks` works the same way as update methods described before. The difference is that this method maps the `JDBC ResultSet` after executing the query to the type identified by method return type, `Book`. The mapping is driven by the result map, referenced by `@ResultMapRef` annotation. Since our mapping is with `auto=true`, the Nanorm will map each column from the `ResultSet` to the property with same name.

## 2.2 Using the Mapper

To perform basic database operations, defined by the methods above, the implementation should be retrieved from the Nanorm factory. First, we create Nanorm configuration and factory:

```
NanormFactory factory = new NanormConfiguration().buildFactory();
```

Then we need to get the mapper implementation:

```
BookMapper mapper = factory.createMapper(BookMapper.class);
```

Finally, we need to provide to Nanorm our database settings. For simplicity, we will setup Nanorm for external transaction management and for this we need to provide it a `JDBC` connection. External transaction management means that Nanorm will take no actions on `JDBC` connection besides the queries themselves and the application should commit/rollback the transaction itself. In the example below the connection is made to the [H2](#) anonymous database:

```
Class.forName("org.h2.Driver");
Connection conn = DriverManager.getConnection("jdbc:h2:mem:", "sa", "");
Session session = factory.openSession(conn);
```

Before running the actual queries, database structure should be set up. The following snippet in SQL describes the sample database structure:

```
CREATE TABLE BOOKS(id INTEGER PRIMARY KEY, name VARCHAR(100),
    author VARCHAR(100), published DATE)
```

Now we can invoke mapper methods to perform database operations:

```
Book book = new Book();
book.setId(1);
book.setAuthor("Brain");
book.setName("World Domination.");
book.setPublished(new java.sql.Date(788922000)); // January 1, 1995

mapper.insertBook(book); // Insert into the database

book.setName("World Domination. Second Edition.");
mapper.updateBook(book); // Update the book

Book book2 = mapper.selectBook(1); // Select the book into other bean by id
mapper.deleteBook(book2.getId()); // Delete the book record from the database
```

Before moving to more advanced examples, remember the following notes regarding the thread-safety:

**Note** rules about the instances thread-safety.

- Configuration instances are not thread-safe. You should not invoke methods on it from different threads.
- Factory instances are thread-safe, you can invoke any methods from any thread. However, the session opened by `openSession` is bound to the current thread only.
- Session instances are not thread-safe. The session is opened for the current thread only.
- Mapper instances are thread-safe, however, their methods are executed in the context of session for the current thread. Before invoking any method on mapper instance, you should open the session for current thread using the factory the mapper is bound to. Mapper instance is intrinsically bound to the factory that created it.

## 2.3 Autogenerated keys

Next thing we want to make the primary key auto-generated. Nanorm supports two kinds of auto-generated keys: before-generated keys and after-generated keys. Before-generated keys are keys that are generated explicitly before invoking the actual query and passed to the query itself via the parameter property. After-generated keys are keys that are either selected using the specified query after invoking the primary query or retrieved using the JDBC support for generated keys. In both cases, the generated key could be returned from the method or could be returned via setting the property of the method parameter, or both.

For our sample application we will use the second approach, after-generated keys. For that to work, we need a source for generated keys first. In case of H2 this will be the sequence:

```
CREATE SEQUENCE SEQ START WITH 100 INCREMENT BY 1
```

Then we need to change our insert method to return generated keys:

```
@Insert("INSERT INTO books(id, name, author, published) " +
"VALUES(NEXT VALUE FOR SEQ, ${1.name}, ${1.author}, ${1.published})")
@SelectKey
int insertBook(Book book);
```

Note the changes: first, we added a `@SelectKey` to our method to indicate that it should return the generated key (note that the after-generated key is default), secondly, the SQL statement was changed, so value for id now is `NEXT VALUE FOR SEQ` and finally the return type of insert method was changed to `int`.

The usage of this method remains the same, but now you simply don't have to specify the id:

```
Book book = new Book();
book.setAuthor("Brain");
book.setName("World Domination.");
book.setPublished(new java.sql.Date(788922000)); // January 1, 1995

mapper.insertBook(book); // Insert into the database
```

If your database JDBC driver does not support generated keys, you can explicitly specify the statement to select the key in the annotation, like `@SelectKey("SELECT CURRVAL('SEQ')")`.

Instead of returning the framework can return the generated key via the property of one of the parameters bean (or both, return the value and set the property). To make framework to set the generated value to the property you just need to specify the property path in the `@SelectKey` annotation. The property path is specified via the `property` parameter of the annotation. In our sample scenario, the changed annotation will look like `@SelectKey(property = "1.id")`. Note that first element of the property path identifies the argument by position (starting with 1). See [Property Paths](#) for more details about property paths.

Since in before-generated key case the query parameter is the only way to pass the generated value to the insertion query itself, the property path is required for before-generated keys. It is assumed that you will use this property to provide value for the

primary key column (however, this is not enforced, so you can generate key before the query and don't use it, forget it, although this does not make any sense).

In our application we could use the before-generated keys the following way:

```
@Insert("INSERT INTO books(id, name, author, published) " +
"VALUES(${1.id}, ${1.name}, ${1.author}, ${1.published})")
@SelectKey(value = "SELECT NEXTVAL('SEQ')", property = "1.id", type = SelectKeyType.BEFORE)
void insertBook(Book book);
```

Note that although we could use `void` return type, the property is mandatory (because otherwise it would be not clear how to pass the generated value to the insertion query itself).

## 2.4 Mapping the collections

After inserting new rows into the database we probably want to select some of them. We already know how to select single entity, let's try select several of them.

First of all let's implement a method that lists all books in the table. To do this, we just remove the `WHERE` clause from the SQL and change the return type to the array of beans, leaving the same result map. Declaring the `listBooks` method:

```
@ResultMapRef("book")
@Select("SELECT id, name, author, published FROM books")
Book[] listBooks();
```

Using the method does not differ from using any other mapper method:

```
Book[] books = mapper.listBooks();
for(Book book : books) {
    System.out.println(book.getName());
}
```

Alternatively, you can make the method return the generic collection. Three collection interfaces are supported: `java.util.Collection`, `java.util.List` and `java.util.ArrayList`. In all three cases the actual collection will be `java.util.ArrayList`. Here is the example of declaring the method:

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

Using it:

```
List<Book> books = mapper.listBooks();
for(Book book : books) {
    System.out.println(book.getName());
}
```

As you can see, there is almost no difference in usage.

What to do if you want to put the result into other collection type or process it as it arrives? Of course, you can select the data into the list or array and to iterate through it later, but this is not very optimal approach. To deal with this situation, framework allows you to pass callback which will receive the data as it is mapped. The callback is implementation of the `com.google.code.nanorm.DataSink` generic interface and it could be passed as any parameter of the method (the parameter index does not matter). If you pass a callback, you should change your method return type to `void`. Here is the example of declaring the method:

```
@ResultMapRef("book")
@Select("SELECT id, name, author, published FROM books")
void listBooks(DataSink<Book> sink);
```



Note that type parameter of the interface specifies the bean class. Here is the example using the method to put the mapped data into the hash map:

```
final Map<String, Book> map = new HashMap<String, Book>();
mapper.listBooks(new DataSink<Book>() {
    public void commitData() {
        // Nothing...
    }

    public void pushData(Book book) {
        map.put(book.getName(), book);
    }
});
```

## Chapter 3

# Reference

### 3.1 Property Paths

TBD

DRAFT

## 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>
...Id
<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 setName(String name) {
        this.name = name;
    }

    public String getAuthor() {
        return author;
    }

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

    public Date getPublished() {
        return published;
    }
}
```

```
public void setPublished(Date published) {  
    this.published = published;  
}  
}
```

DRAFT

---