SideNail

Martín Schonaker, Santiago Martínez

March 24, 2010

Chapter 1

Quickstart

1.1 Hello, World!

Listing 1.1 shows the simplest mapping example one can build with SideNail. The example shows how to query the complete "student" table. The contents of the table are "dynamically mapped" as the `students` list.

```
// Obtain a database connection.
// java.sql.Connection connection = ...

// Start the SideNail session.
SideNail.start();

// Set the current connection (can switch at any point).
SideNail.setConnection(connection);

// Query.
List<?> students = SideNail.map("select * from student");

// Do something reflective with students...

// Finish the SideNail Session.
SideNail.end();
```

Listing 1.1: Simple mapping example.

In the listing, the first (commented) part states that you need a database connection (`java.sql.Connection`) to start. It doesn't matter (from the SideNail point of view) if you obtain it through driver manager `java.sql.DriverManager` or through a datasource `javax.sql.DataSource`.

Once you've obtained a connection you're almost ready to perform the mapping. But before you need to tell SideNail what will be the scope of the

dynamic objects you are creating. To do so, you need to invoke methods `SideNail#start()` and `SideNail#end()`.

With the scope already declared, you can now map database queries (not only tables). It is quite probable that you always perform like in the above example: query the database (`SideNail#map(String)`) and do something with the gather data. Since the classes SideNail creates are not known until runtime, you'll have to perform something with the Java reflection API.

Here's the explanation of what's really happening. SideNail will create a prepared statement with the text you introduced for the query. Will execute the query in the given connection. Once the result has arrived, SideNail will create, compile, and load a class for it. A unique class only known at runtime, since you're query is only known at runtime. For each row in the result, SideNail will create an object and fill a collection with them. Benefits? lesser boilerplate code. Drawbacks? you can only access the objects through reflection, since their class is not known at compile-time[1].

1.2 Accessing Results

The classes created for the queries results are standard Javabeans. This means that you will be able to use several reflection-based APIs used for them, such as JSP with EL, Velocity, Flexjson, XMLEncoder, etc, etc, etc. just to mention a few.

However for the simpler tasks, SideNail provides two helpful methods: `SideNail#set(Object, String, Object)` and `SideNail#get(Object, String)`.

1.3 Querying

The listing 1.3 show how to use arguments to prevent SQL-injection. Arguments are "varargs" so for every parameter in the query (indicated with '?') you should put an argument with the value. Objects to use depend on the types de database driver can handle. They are just like in `java.sql.PreparedStatement`.

```
// Returns the students whose name contains "A".
List<?> students = SideNail.map("select * from student
    where name like ?", "%A%");
```

Listing 1.2: Querying with arguments.

There are a few more handy methods when querying. `SideNail#scalar(String)` (and its parameterized counterpart `SideNail#scalar(String, Object...)`) allows to query a scalar value from the database, just like in listing 1.3.

[1] But keep reading there's also "typed" mapping in SideNail.

```
// Returns the students whose name contains "A".
Long count = (Long) SideNail.scalar("select count(*) from
    student where name like ?", "%A%");
```

Listing 1.3: Querying scalars.

`SideNailexecute(String)` (and its parameterized counterpart `SideNail#execute(String, Object...)`) allows you to execute update queries, like DDL queries, updates, insertions, deletions, etc. This is when you don't expect any result, the execution is just enough.

1.4 Typed Mapping

When you need strong-type checking, just tell SideNail what class you need to use (check listing 1.4). Then you can access the beans properties without having to exclusively use a reflective API.

```
// Obtain a database connection.
// java.sql.Connection connection = ...

// Start the SideNail session.
SideNail.start();

// Set the current connection (can switch at any point).
SideNail.setConnection(connection);

List<Country> countries = SideNail.map(Country.class, "
    select limit 0 3 * from country order by code");

// Do something not necessarily reflective with countries
    ...

// Finish the SideNail Session.
SideNail.end();
```

Listing 1.4: Simple typed mapping example.

1.5 CRUD

Since version 0.3, SideNail is able to perform simple CRUD operations. Here's an example. Not much to comment.

```
Object country = SideNail.create("COUNTRY");
```

```
SideNail.set(country, "name", "SideLand");
SideNail.set(country, "code", "SL");
SideNail.set(country, "area", Integer.valueOf(20));
SideNail.set(country, "province", "SideLand");
SideNail.set(country, "population", Integer.valueOf(2));
SideNail.set(country, "capital", "Orm");

SideNail.insert(country);

Long count = (Long) SideNail.scalar(
  "select count(*) from COUNTRY where CODE = ? and NAME =
      ?",
  "SL", "SideLand");

assert 1L ==  count.longValue();

SideNail.set(country, "name", "SideNailLand");

SideNail.update(country);

count = (Long) SideNail.scalar(
  "select count(*) from COUNTRY where CODE = ? and NAME =
      ?",
  "SL", "SideNailLand");

assert 1L == count.longValue();

count = (Long) SideNail.scalar(
  "select count(*) from COUNTRY where CODE = ? and NAME =
      ?",
  "SL", "SideLand");

assert 0L == count.longValue();

SideNail.delete(country);

count = (Long) SideNail.scalar(
  "select count(*) from COUNTRY where CODE = ? and NAME =
      ?",
  "SL", "SideNailLand");

assert 0L == count.longValue();
```

Listing 1.5: CRUD operations.

Chapter 2

Advanced Features

2.1 Collection Reuse

//TODO

2.2 Structured Mapping

//TODO

2.2.1 Simple Structured Mapping

//TODO

2.2.2 Typed Structured Mapping

//TODO

2.2.3 "Hybrid" Structured Mapping

//TODO

2.3 Class-Definition Manipulation

//TODO

2.3.1 Adding, Removing and Modifying Properties

//TODO

2.4 Relationship Mapping

//TODO

2.4.1 One-To-One Mapping

//TODO

2.4.2 One-To-Many Mapping

//TODO

2.4.3 Inheritance Mapping

//TODO

Chapter 3

Experimental Features

//TODO

3.1 Join Mapping

//TODO

3.2 One-shot Paging

//TODO