Petit: Reference Manual

Foreword

Petit is a light-weight object-relational mapping framework with a goal to help developers write maintainable database queries. Unlike Hibernate, which aims to be a framework for mapping the object-oriented domain model to traditional relational database, Petit focuses on single object model-table mapping, persistence and queries.

Petit builds upon Spring Framework's JDBC abstraction framework relieving the developer from writing SQL in Java code, direct result set handling and object conversion. For persistence Petit uses Plain Old Java Objects (POJOs) that are enhanced with standard JPA annotations.

User Guide

It is always easier to show than to explain, so let's see what Petit has to offer.

As an example, let us implement a simple database that holds projects. Every project has multiple members with various roles and of course every member is a person.

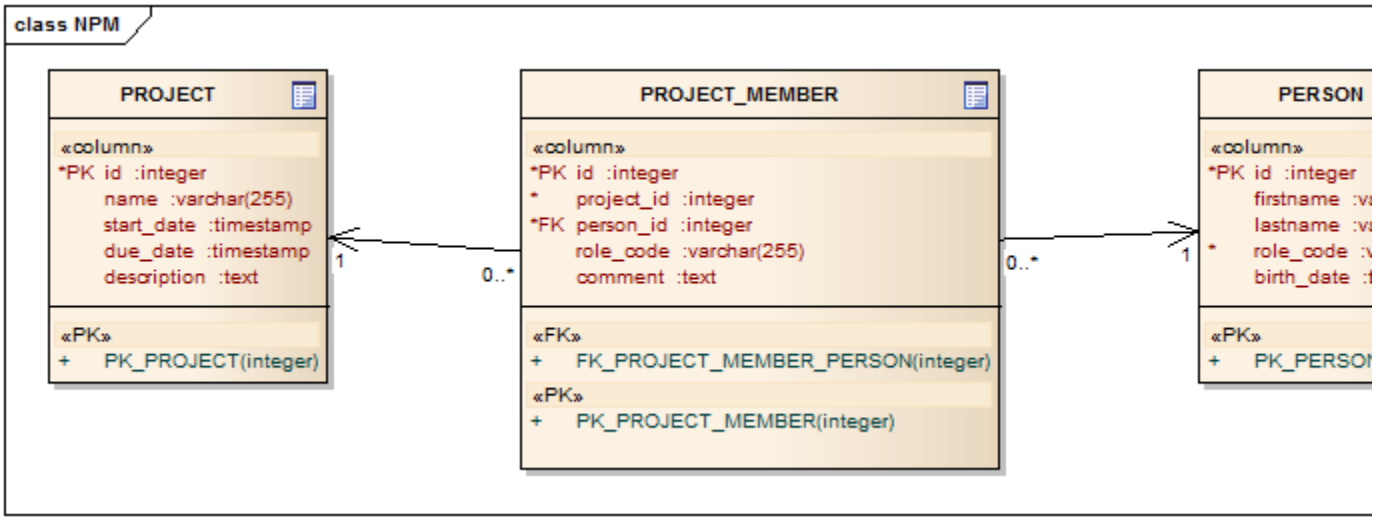

Fig. 1 Sample project database model

Our imaginary application will use this database to hold various projects, keep track of project members, and link them to the company employee data.

Model

Before we can start coding real business logic such as saving projects and adding project members we need a model representation. As mentioned earlier Petit uses POJOs for persisting data, so the following Java class represents PROJECT table:

```
@Table
public class Project {
@Id
private Long id;
private String name;
private Date startDate;
private Date dueDate;
private String description;


...getters/setters...
}
```

There are a few things to note. A class must be annotated with a `@Table` annotation and the primary key column must have an `@Id` annotation, both standard JPA annotations. Table and column names are converted between `camelCase` in Java and `under_score` in SQL, e.g., `startDate` <-> `start_date`. If you want to use different names in Java and SQL, you can provide a database column name mapping via the name attribute of standard `@Table` and `@Column` annotations. The `@Column` annotation can also be used when you want to exclude a field from insert and update queries.

If the POJO contains methods or fields that are not database-related they can me marked as ignored with a `@Transient` annotation.

Long story short, by default Petit currently employs the following JPA annotations and their attributes in a more-or-less standard JPA meaning:

- `Table(name)`
- `Id`
- `Column(name, insertable)`
- `Transient`
- `Embedded`
- `AttributeOverride`
- `AttributeOverrides`

Following the example we define Java classes for PROJECT_MEMBER and PERSON tables. Class containing project members holds also a List of projects and persons, but we come back to that when we explain loading related objects.

<<some section about explaining how to set-up BaseDAO and database dialect ?>>

InsertStatement

Now that we have the model objects defined and Petit configured it's time to start using it. First use-case scenario is to add new project to database.

For that we use the InsertStatement and the usual usage is as follows:

```
Project project = createProjectForInsert();
…
InsertStatement<Project> insert =
new InsertStatement<Project>(
dbHelper.getJdbcTemplate(),
dbHelper.getStatementBuilder(),
project);
insert.exec();
```

This code generates the following pseudocode SQL statement:

```
INSERT INTO project (.. column ..) VALUES (.. column values ..);
```

You initialize InsertStatement with jdbcTemplate and statementBuilder defined in your application, the object you want to persist and call exec method. That’s it. Petit automatically generates the necessary SQL statement, executes it through jdbcTemplate and additionally fills the primary key column with the newly generated primary key value.

<<Primary key explained>>

UpdateStatement

UpdateStatement is very similar to InsertStatement, but you can define an additional WHERE clause. By default the primary key is used in the WHERE clause of SQL statement.

```
UpdateStatement<Project> update =
new UpdateStatement<Project>(
dbHelper.getJdbcTemplate(),
dbHelper.getStatementBuilder(),
project);
update.exec();
```

This code generates the following pseudocode SQL statement:

```
UPDATE project SET [column = column_value] WHERE id = [id value];
```

Sometimes you need to update multiple rows with the same data. This can be achieved by creating a new Project object, setting the required field values, adding list of columns to be updated and a custom WHERE clause:

```
update.setBy(„name“).where(gt(„dueDate“, [date_value])).exec();
```

This code generates the following pseudocode SQL statement:

```
UPDATE project SET name = [project_name] WHERE due_date > [date_value];
```

We cover the WHERE part and its usage possibilities in Petit in more detail under LoadStatement section.

DeleteStatement

DeleteStatement also is very similary to Insert- and UpdateStatement with a few nuances.

This is the default usage of DeleteStatement:

```
DeleteStatement<Project> delete =
new DeleteStatement<Project>(
dbHelper.getJdbcTemplate(),
dbHelper.getStatementBuilder(),
project);
delete.exec();
This code genera
```

tes the following pseudocode SQL statement:

```
DELETE FROM project WHERE id = [id value];
```

If we need more specific conditions and want to delete more than one row we should use DeleteStatement in this way;

```
DeleteStatement<Project> delete =
new DeleteStatement<Project>(
dbHelper.getJdbcTemplate(),
dbHelper.getStatementBuilder(),
Project.class);
delete.where(lt("due_date", new Date())).exec();
```

This code generates the following pseudocode SQL statement:

```
DELETE FROM project WHERE due_date < [date_value];
```

LoadStatement

Finally we come to the statement you are likely to use the most SELECT or Load in Petit dialect.

LoadStatement is initialized this way:

```
LoadStatement<Project> load =
new LoadStatement<Project>(
dbHelper.getJdbcTemplate(),
dbHelper.getStatementBuilder() ,
Project.class);
```

Next we need to define the fields we query, what we query, order of items and how many.

Selecting columns

By default Petit will query all fields that are defined in the Project class. When we need to select only a sub-set of fields we can define them through select method:

```
load.select(“name”, “dueDate”);
```

Note that the names given in the select method correspond to the Java object field names.

Constructing WHERE clause

We have skipped over the construction of a WHERE clause till now. It’s time to explain it more thoroughly.

Constructing the WHERE part revolves aroung giving the statement an implementation of SqlPart interface. Interface itself is very simple:

```
public interface SqlPart {
String sql(Function<String, String> nameMapper);

List<Object> params(Function<String, Object> paramMapper);
}
```

It states basically that give me the SQL using the specified namemapper and give me the parameters used in the SQL snippet. The actual implementation used in defining WHERE clause is in SimpleWherePart and CompositeWherePart classes, that combine parts of a WHERE clause with the operation used in that operation. Thus a SimpleWherePart contains the database field name, operation and actual value used (ID = [id_value]) and CompositeWherePart contains a collection of simple or composite parts combining them with logical operation such as AND and OR.

Luckily this information is mostly irrelevant to you as a developer because of helper methods in Where class. If we want to express the following WHERE clause

```
WHERE name LIKE ‘project%’ AND
(start_date < current_time AND due_date > current_time) AND
description IS NULL
```

we would write it as:

```
.where(
like("name", "project%")
.and(lt("startDate", new Date()).and(gt("dueDate", new Date()))
.and(eq("description", null))));
```

Quite straightforward.

Constructing ORDER clause

ORDER clause is similar to WHERE clause in that sense that it is basically a different implementation of SqlPart interface. Ordering projects by name and descendingly by dueDate is simple as:

```
.order(asc("name"),desc("dueDate"));
```

Limiting query results

Insert-, Update- and DeleteStatements had only one execution method exec(). LoadStatement has three:

- `all()` – Loads all results from database
- `single()` – Loads exactly one result (throws standard Spring exceptions if no results or more than one)
- `range(start, limit)` – Adds a LIMIT part to the SQL statement

To summarize. We construct a query that selects all projects where project name starts with string “project”, are not inactive (startDate in the past and dueDate in the future), ordered by name and dueDate descendingly and limited to 10 result:

```
.where(
like("name", "project%")
.and(lt("startDate", new Date()).and(gt("dueDate", new Date()))
.and(eq("description", null))))
.order(asc("name"), desc("dueDate"))
.range(0, 10);
```

QueryStatement

QueryStatement is a specialized LoadStatement with a goal to simulate a paged table. Although doing paged queries is also possible by using LoadStatement with range limitation, QueryStatement is intended to use as a backing query for tables with filters and pagination.

QueryStatement initialization follows the familiar pattern:

```
ListQuery<Project> query = new ListQuery<Project>(Project.class);
QueryStatement<Project> qs =
 new QueryStatement<Project>(
  query,
  dbHelper.getStatementBuilder(),
  dbHelper.getJdbcTemplate());
```

Main difference between LoadStatement and QueryStatement is that QueryStatement uses a ListQuery object for input and output. ListQuery is a class that encapsulates SQL query and results. It is meant to be populated in web layer, then passed to business layer where the query is executed and finally returned with query result that can be shown in web layer. The following example constructs a ListQuery that searches projects by name and startDate.

```
ListQuery<Project> query = new ListQuery<Project>(Project.class);
query.addFilter(like("name", "Paged%"));

Calendar calStart = Calendar.getInstance();
calStart.add(Calendar.DAY_OF_YEAR, -20);
Calendar calEnd = Calendar.getInstance();
calEnd.add(Calendar.DAY_OF_YEAR, -10);

query.addFilter(range("startDate", calStart.getTime(), calEnd.getTime()));

query.setLimit(0, 5);
```

Constructed query is then passed to the business layer for execution:

```
QueryStatement<Project> qs = new QueryStatement<Project>(query,
dbTestHelper.getStatementBuilder(), dbTestHelper.getJdbcTemplate());
ListQuery<Project> result = qs.query();
return result;
```

And finally result is used in web-layer:

```
QueryResult<Project> qr = result.getQueryResult();
qr.getTotalCount()); // Total count of elements
qr.getResultList().size()); // Current resultList size, depends on limit
```

One additional possibility with QueryStatement is to cache partial results to avoid querying the same page over-and-over again. To use caching initialize QueryStatement with option cacheResult.