Squill documentation

Preparation

All Squill queries begin with an instance of squill.Squill. To get an instance you should invoke the constructor or a static method Squill.squill with javax.sql.DataSource or squill.db.Database as an argument.

```
Squill squill = new Squill(dataSource);
```

If used together with Spring Framework, the most convenient approach is to inject Squill into the service.
You should also instantiate table classes you want to use in your queries/statements. You may pass a string argument, that will be used as a table alias.

```
CustomerTable c = new CustomerTable ("c");
ComplaintTable co = new ComplaintTable();
```

Note that this class isn't the same as the generated model class. Model class represents data in the table, table class represents the table itself (and it's structure).
For a more convenient usage of static methods representing SQL operations, you should include this line to your imports:

```
import static squill.api.functions.Operations.*
```

Select queries

from clause

Appears after Squill instance. Takes one base table and zero to many FromExpressions, which could be tables or joins and returns SelectBuilder. Every table class, generated by metadata generator implements the FromExpression interface.

```
SelectBuilder crossSb = squill.from(co, c);
```

Squill is capable of making implicit inner joins based on the metadata, in case there is a relation through foreign keys:

```
SelectBuilder joinSb = squill.from(co, co.customer());
```

join clause

Appears inside or after the *from* clause. In the first case it is a static method in the Operations class, in the second case - non-static method in the SelectBuilder class. Takes three arguments - a table to join and two columns, which the join is performed on. Returns a SelectBuilder, just like *from* clause.

```
SelectBuilder joinSB = squill.from(co, join(c, co.customerId, c.id));
```

Type safety is guaranteed - compile-time error will be thrown in case join is performed using columns with different data types.

where clause

Appears after *from* or *join* clause. Takes one to many WhereExpressions as arguments and returns SelWhereBuilder. In case of many arguments, they are combined using AND's. WhereExpression can be built using boolean operators. Single WhereExpression can itself combine different WhereExpressions using boolean operators.

```
WhereExpression w1 = ge(c.discount, 20);
w1 = w1.or(c.isActive.eq(1));
SelWhereBuilder swb = squill
    .from(c)
    .where(w1);
```

Conditional operators

Conditional operators are represented by a number of methods, either static ones in Operations class or non-static ones in the Column class. Most of the binary operators take two SelectExpessions, two values or a SelectExpression and a value as arguments. SelectExpression is an interface that is implemented by all tables and columns. You can use values or variables freely as Squill will set them as parameters preventing any kind of SQL injection.

Method	eq	ne	gt	ge	lt	le	isNull	notNull
Operator	=	<>	>	>=	<	<=	IS NULL	IS NOT NULL

Boolean operators

Boolean operators are represented by methods *or, and, not*, either static ones in Operations class or non-static ones in the WhereExpression interface (except for the *not* method).

orderBy clause

Appears after *from, join* or *where* clause. Takes one to many OrderByElements as arguments and returns a OrderByBuilder. OrderByElements are generated using columns and asc()/desc() methods (either static ones in Operations class or non-static ones in the Column class).
If the ascendancy/descendancy is decided at runtime, you can use the method ascOrDesc, which takes in addition a boolean argument (true -> ascendancy)

```
OrderByElement o1 = c.name.asc();
OrderByElement o2 = ascOrDesc(c.id, false);
OrderByBuilder obb = squill.from(c).orderBy(o1), asc(o2));
```

select clause

Appears in the end of the method chain. Takes 1 to 10 SelectExpressions as arguments and returns an object (in case of 1 argument) or a tuple. Tuples are types to encode rows of values with types known ahead. The tuples are available from Tuple2 to Tuple10 encoding accordingly two to ten values in a row. Tuple1 is not provided and is replaced by the value itself (i.e. Tuple1<T> -> T). Tuples have fields v*n* that hold the n^{th} value in the row. Note that since the table columns are generified the resulting tuple type is verified by the compiler and can be inferred by the IDE.

```
Tuple2<String, Date> clientTuple =
    squill
        .from(c)
        .where(eq(c.id, 1))
        .select(concat(c.firstName, " ", c.lastName),
            c.birthdate);
```

selectList clause

Almost the same as the *select* clause, but expects and returns a list of tuples or objects.

```
List<CustomerData> customerList =
    squill
      .from(c)
      .selectList(c);
```

Unchecked queries

If you need to insert a piece of raw SQL you can do that by just providing the type:

```
Integer count = squill
    .from(c)
    .select(unchecked(Integer.class, "count(*)"));
```

Subselects

subSelect method appears in the end of the method chain and works similarly to *select* method, only that it return a SelectExpression instead of a value. This method isn't initiating the query, the returned value should be used in another query.

```
    SelectExpression<Integer> subQuery =
        squill
            .from(co)
            .where(eq(co.customerId, c.id))
            .subSelect(unchecked(Integer.class, "count(*)"));

    List<Tuple2<CustomerData, Integer>> res =
        squill
            .from(c)
            .orderBy(desc(subQuery), asc(c.id))
            .selectList(c, subQuery);
```

Fetching an entire object

This is a shortcut for select query in case one row needs to be extracted using it's primary key:

```
CustomerData pet = CustomerData.get(squill, id);
```

or

```
CustomerData pet = squill.get(CustomerData.class, id);
```

CUD Statements

These are SQL insert/update/delete statements. Every statement comes in two flavors: simple, where you define the whole statement manually, and ORM-like, where Squill generates the statement itself, using the generated metadata. In case of the simple solution you work with an instance of a table class (see "Preparation"). In case of the ORM-like solution you work with an instance of a data class. It is not important how was it obtained:

```
CustomerData customer = getCustomer();
```

Simple insert statement

```
squill
    .insert(c)
    .values(
        insertElement(c.id, 7),
        insertElement(c.name, "John"));
```

This will result in a following query:

INSERT INTO customers (ID, NAME) VALUES (?, ?)
ARGS: [7, John]

Adding an alias to the table doesn't affect insert queries. Last method in the chain returns the id of the inserted object.

Inserting an object

```
squill.insertDataObject(customer);
```

or

```
customer.insert(squill);
```

The row is inserted using all mapped fields. Null is always passed as the primary key value (i.e. statement depends on autogeneration of the primary key in the database).

Simple update statement

```
squill
    .update(c)
    .where(eq(c.id, 7))
    .set(
        updateElement(c.name, "Peter"));
```

This will result in a following query:

```
UPDATE customers c SET NAME=? WHERE (c.ID = ?)
ARGS: [Peter, 7]
```

Updating an object

```
squill.updateDataObject(customer);
```

or

```
customer.update(squill);
```

The row is matched using the primary key and updated using all mapped fields.

Simple delete statement

```
squill
    .delete(c)
    .where(eq(c.id, 3));
```

This will result in a following query:

```
DELETE FROM customers c WHERE (c.ID = ?)
ARGS: [3]
```

Deleting an object

```
squill.deleteDataObject(customer);
```

or

```
customer.delete(squill);
```

The row is matched using the primary key and deleted.

Run-time query fail

Although Squill is fail-fast, some queries/statements may fail at runtime, though they were completely correct when compiled. Here is a shortlist of what you should care about, because Squill itself can't:

- Null values. Inserting/updating a row setting null into a notnull column.
- Incorrect unchecked queries.
- Dependencies. Deleting a parent element without deleting children, setting unsuitable value into a column with a constraint.
- Too many results. Simple select method expects one row at most. Use selectList in case more rows are expected.