S Y S E

Tornado Query

v. 1.0.3

Tutorial

Table of contents

1 Introduction

1.1 Introduction

Tornado Query is a persistence framework heavily influenced by mybatis, without the XML. It sports automatic CRUD based on Mappers that translate SQL result sets to domain objects, and lets you use your domain objects as parameters to queries.

- Automatic creation of Mappers based on database metadata
- CRUD (Create, Read, Update, Delete)
- Insert statements knows how to increment sequences
- Autogenerated SQL with joins

NOTE: Even though Tornado Query can save you a lot of time for example by creating the join part of your queries, the main focus is still to let YOU write your SQL, your way, without interference. If you know SQL, there is little extra you need to know to use Tornado Query efficiently.

Add Tornado Query to your Maven project:

```
<dependency>
    <groupId>no.tornado</groupId>
    <artifactId>query</artifactId>
    <version>1.0.3</version>
</dependency>
```

Check out the source:

```
svn checkout https://opensource.subversion.no/query/trunk query
```

2 Getting Started

2.1 Getting Started

In this tutorial we'll introduce an example database and some corresponding domain objects.

Start by examining the example schema, and move on to how to get a database connection. From there, we'll start exploring how Tornado Query can save you time and make you write less code, but accomplish more, in less time than you are used to!

NOTE: The tutorial is also available in PDF format.

3 Example database

3.1 Example database

This tutorial will use an example database with three tables, and three corresponding domain objects.

We use PostgreSQL syntax in the examples. Here is the DDL for our database:

```
CREATE SEQUENCE country_id;
CREATE SEQUENCE address_id;
CREATE SEQUENCE customer_id;

CREATE TABLE country (
    id INTEGER NOT NULL PRIMARY KEY DEFAULT nextval('country_id'),
    name TEXT
);

CREATE TABLE address (
    id INTEGER NOT NULL DEFAULT nextval('address_id'),
    street TEXT,
    zip TEXT,
    city TEXT,
    country INTEGER NOT NULL REFERENCES country(id)
);

CREATE TABLE customer (
    id INTEGER NOT NULL PRIMARY KEY DEFAULT nextval('customer_id'),
    name TEXT,
    email TEXT,
    delivery_address INTEGER NOT NULL REFERENCES address(id),
    billing_address INTEGER NOT NULL REFERENCES address(id)
);
```

And these are our corresponding domain objects, getters and setters omitted for brevity:

```
public class Country {
    private Integer id;
    private String name;
}

public class Address {
    private Integer id;
    private String street;
    private String zip;
    private String city;
    private Country country;
}

public class Customer {
    private Integer id;
    private String name;
    private String email;
    private Address deliveryAddress;
    private Address billingAddress;
}
```

4 Getting a connection

4.1 Getting a connection

Tornado Query expects that you give it a `java.sql.Connection`, and doesn't really help you in obtaining one. Here we'll introduce some ways to do that, and show you how you would typically use Tornado Query in your applications. Please feel free to skip this chapter for now, and revisit it once you start actually using Tornado Query. Read about mappers and usage to wet your appetite, then come back for the details!

4.1.1 Explicit

The `Query` class has a static `ThreadLocal` where you can set a connection before performing queries. Tornado Query expects this connection to be ready to use, and will not commit, rollback or close this connection in any way. In it's simplest form, it means that you can use Tornado Query like this:

```
// Create a connection and set it
Query.connection.set(DriverManager.getConnection("jdbc:postgresql:/host/database"));

// Perform your queries here
...

// Close the connection
Query.connection.get().close();
```

Instead of using the static `ThreadLocal`, you can also supply each Query object with a specific connection. Remember that you are still responsible for closing, committing and rolling it back:

```
// Create a connection and set it
Connection conn = DriverManager.getConnection("jdbc:postgresql:/host/database");

// Set the connection on each query
Query.create(...).connection(conn);

// Close the connection
conn.close();
```

4.1.2 WebApp

In a web application it might make more sense to let a `Filter` make sure you have a connection, and even take care of the transaction demarcation for you. If you want to make sure that you always have a connection handy in every request, and that it gets commited if everything is OK, and rolled back if something goes wrong, you can register a `Filter` in your web application. We will also create a Commons DBCP connection pool to give each thread a connection efficiently.

```
@WebFilter("/*")
public class QueryFilter implements Filter {
    private BasicDataSource ds;

    public void init(FilterConfig filterConfig) throws ServletException {
        ds = new BasicDataSource();
        ds.setDriverClassName("org.postgresql.Driver");
        ds.setUrl("jdbc:postgresql:/host/dbname");
        ds.setDefaultAutoCommit(false);
        ds.setUsername("myuser");
        ds.setPassword("mypassword");
    }

    public void doFilter(ServletRequest request, ServletResponse response,
                    FilterChain chain) throws IOException, ServletException {
        try (Connection c = ds.getConnection()) {
            Query.connection.set(c);

            chain.doFilter(request, response);

            c.commit();
        } catch (Exception ex) {
            if (Query.connection.get() != null)
                try { Query.connection.get().rollback(); }
                    catch (SQLException ignored) { }
            throw new ServletException(ex);
        } finally {
            Query.connection.remove();
        }
    }

    public void destroy() {
        try {
            ds.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

In the `init` method, we create a DataSource which is basically a pool of opened connections. We make sure that the connections returned from the pool does not auto commit each transaction. This way, we wrap the entire HTTP request in one SQL Transaction, and commit only if everything is OK. We roll back if an Exception occurs. Finally, the `destroy` method will close the datasource and release all the connections.

You might want to exclude certain patterns, like files etc from the filter, or use a more sofisticated algoritm to determine if a connection should be provided.

All examples in this tutorial will assume that you have already given the current thread a connection using a strategy like the ones described here.

5 Mappers

5.1 Mappers

The main responsibility of a Mapper is to make sure that your SQL ResulSet is converted into your domain objects. However, since you tell them about the columns in your database, they can also be used to automatically create the SQL to perform CRUD operations.

For now, we'll write Mappers for our example domain objects manually.

```
public class Mappers {
    public static final Mapper<Country> countryMapper = new Mapper(Country.class)
        .tablename("country")
        .id("id", "id", "country_id", INTEGER)
        .property("name", "name", VARCHAR);

    public static final Mapper<Address> addressMapper = new Mapper(Address.class)
        .tablename("address")
        .id("id", "id", "address_id", INTEGER)
        .property("street", "street", VARCHAR)
        .property("zip", "zip", VARCHAR)
        .property("city", "city", VARCHAR)
        .join("country", countryMapper, "country");

    public static final Mapper<Customer> customerMapper = new Mapper(Customer.class)
        .tablename("customer")
        .id("id", "id", "customer_id", INTEGER)
        .property("name", "name", VARCHAR)
        .property("email", "email", VARCHAR)
        .join("deliveryAddress", addressMapper, "delivery_address")
        .join("billingAddress", addressMapper, "billing_address");
}
```

Tornado Query can auto-create Mapper objects by trying to match column names to domain object properties. It will even figure out how domain objects fit together based on your database foreign keys. After you have auto-created a Mapper, you are free to alter it manually, or you can ofcourse write your mapper objects manually.

The `Mapper#generate()` method is used to auto-create Mappers. The generator creates one Java class for each domain object, and adds a static final field for the actual Mapper. So, the Mapper for a Customer object would be accessed via `CustomerMapper.FULL`. You are free to store the Mappers any way to like, infact in our example we'll create a single class called Mappers, and put our Mappers in there.

5.1.1 Country mapper

Let's examine the `countryMapper` first. The constructor takes the domain object class as it's argument. Next, we supply the name of the table (country). The `id` method maps the Country domain object's `id` to the `id` column in the country database table. The third argument (country_id) tells the Mapper that the id should be auto-generated by incrementing the country_id sequence. The last argument is the `java.sql.Types` type of the database column. You actually don't have to supply

this argument, Tornado Query can figure it out, and will cache the result. If you supply it, the first query will be a couple of milliseconds faster.

As you probably guessed by now, the `property` method maps the `name` field to the `name` column.

5.1.2 Address mapper

The `addressMapper` starts out in much the same way as the countryMapper, except it has more fields, mapped with `property` mappings. The interesting bit however, is the `join` method. This single line will merge the `countryMapper` into the `country` property of our Address domain object, so that all fields joined in from country will be populated into our Address object.

Instead of referencing the `countryMapper`, we could also have mentioned the fields from Country directly in the `addressMapper`. The following example gives the exact same mapping possibilities, but with more/explicit code:

```
public static final TableJoin countryJoin = new TableJoin("country", "country")
    .on("address.country = country.id");

public static final Mapper<Address> addressMapper = new Mapper(Address.class)
    .tablename("address")
    .id("id", "id", "address_id", INTEGER)
    .property("street", "street", VARCHAR)
    .property("zip", "zip", VARCHAR)
    .property("city", "city", VARCHAR)
    .property("country.id", "country", INTEGER);
    .join("country.name", countryJoin, "name");
```

Here we created a `TableJoin` manually. The contructor takes the table name and an alias to use in the SQL as arguments. Then we give it information about how to perform the join in the `on` chaining method.

The `countryJoin` TableJoin is then used as the second argument to the `join` method, telling the Mapper that the `country.name` nested property of our `Customer` object should be filled with the name column from the joined in country table.

For more information about `TableJoin` and join types, please see the JavaDoc.

As you can see, a lot was given to us for free by referencing the `countryMapper`, but sometimes you want more control of how you configure your mappings. Under the hood, Tornado Query will create a similar join table on-the-fly when you use the simple approach.

5.1.3 Customer mapper

The `customerMapper` references the `addressMapper` two times, for the `deliveryAddress` and `billingAddress` properties. Since the `addressMapper` references the `countryMapper`, our Customer will automatically know how to both map and join all the way to `deliveryAddress.country.name`.

NOTE: It is important to understand that you can still write your query by hand, and still use the Mapper to convert your ResultSet into a domain object. You don't need to use the automatically generated join, but it will save you a lot of boiler plate SQL code.

For completeness, this is how the `customerMapper` would be written if you didn't reference the `addressMapper`:

```
TableJoin deliveryAddress = new TableJoin("address", "delivery_address")
    .on("customer.delivery_address = delivery_address.id");

TableJoin deliveryAddressCountry = new TableJoin("country", "delivery_address_country")
    .on("delivery_address.country = delivery_address_country.id");

TableJoin billingAddress = new TableJoin("address", "billing_address")
    .on("customer.billing_address = billing_address.id");

TableJoin billingAddressCountry = new TableJoin("country", "billing_address_country")
    .on("billing_address.country = billing_address_country.id");

public static final Mapper<Customer> customerMapper = new Mapper(Customer.class)
    .tablename("customer")
    .id("id", "id", "customer_id", INTEGER)
    .property("name", "name", VARCHAR)
    .property("email", "email", VARCHAR)
    .property("deliveryAddress.id", "delivery_address", INTEGER)
    .join("deliveryAddress.street", deliveryAddress, "street", VARCHAR)
    .join("deliveryAddress.zip", deliveryAddress, "zip", VARCHAR)
    .join("deliveryAddress.city", deliveryAddress, "city", VARCHAR)
    .join("deliveryAddress.country.id", deliveryAddress, "country", VARCHAR)
    .join("deliveryAddress.country.name", deliveryAddressCountry, "name", VARCHAR)
    .property("billingAddress.id", "billing_address", INTEGER)
    .join("billingAddress.street", billingAddress, "street", VARCHAR)
    .join("billingAddress.zip", billingAddress, "zip", VARCHAR)
    .join("billingAddress.city", billingAddress, "city", VARCHAR)
    .join("billingAddress.country.id", billingAddress, "country", VARCHAR)
    .join("billingAddress.country.name", billingAddressCountry, "name", VARCHAR);
```

Executing:

```
Query.select(customerMapper).rows();
```

Gives you this SQL for free, and maps your result to a `List<Customer>`:

```
SELECT  customer.id,
        customer.name,
        customer.email,
        customer.delivery_address,
        delivery_address.street AS delivery_address_street,
        delivery_address.zip AS delivery_address_zip,
        delivery_address.city AS delivery_address_city,
        delivery_address.country AS delivery_address_country,
        delivery_address_country.name AS delivery_address_country_name,
        customer.billing_address,
        billing_address.street AS billing_address_street,
        billing_address.zip AS billing_address_zip,
        billing_address.city AS billing_address_city,
        billing_address.country AS billing_address_country,
        billing_address_country.name AS billing_address_country_name
FROM    customer
        JOIN address AS delivery_address
            ON customer.delivery_address = delivery_address.id
        JOIN country AS delivery_address_country
            ON delivery_address.country = delivery_address_country.id
        JOIN address AS billing_address
            ON customer.billing_address = billing_address.id
        JOIN country AS billing_address_country
            ON billing_address.country = billing_address_country.id
```

I'm pretty sure you realize the potential savings by using the full power of Mappers by now :)

5.2 RowConverters - native ResultSet to domain objects mapping

The Mapper uses reflection to convert a ResultSet into domain objects. It takes care of instantiation objects in deep object graphs to avoid null pointer exceptions, and it is very fast. However, nothing can beat the speed of explicitly converting a ResultMap to a domain object, so you can choose to add a `RowConverter` to your Mapper to do exactly this.

NOTE: Using a `RowConverter` is very seldom needed, and the speed of the default reflection based mapping should be adequate for most uses cases. However, it might be worth the extra code in cases where you need to convert ResultMap entries into exotic domain members, or to map to a domain object that can only be created correctly through calling a specific constructor method for example.

Let's add a `RowConverter` to our CountryMapper:

```
countryMapper.rowConverter(new RowConverter<Country>() {
    public Country convert(ResultSet rs) throws SQLException {
        Country country = new Country();
        country.setId(rs.getInt("id"));
        country.setName(rs.getString("name"));
        return country;
    }
});
```

5.3 Auto increment / sequences

The `id` method is just a shortcut for the following:

```
String property = "id";
String column = "id";
int sqlType = java.sql.Types.INTEGER;
Mapping mapping = new Mapping(property, column, MapType.PRIMARY_KEY)
                    .sqlType(sqlType).sequence("id");
// Add to mapper
.property(column, mapping);
```

By setting the `MapType` to `PRIMARY_KEY` we tell the Mapper about what column to join on, and what fields to use in `byId` queries. The `sequence` property is again a shortcut to automatically retrieving a sequence value for the `id` property of the domain object. This can also be done manually in an `INSERT` statement, see INSERT usage for more information.

6 Usage

6.1 Usage

In the following we'll look at how to perform actual queries against our database. Tornado Query was created with one main goal: Let you write SQL explicitly, and map the results to/from your domain objects. There was nevery any intention for Tornado Query to do magical stuff, like Hibernate, JPA and the likes.

As it turns out, describing the mappings between table columns, table relations and domain objects, you have already told Tornado Query enough to do automatic CRUD with joins, so it would be foolish to not support this. Even so, you can still do everything manually, and just use the Mappers for the actual mapping of the ResultSet.

Because of this fact, the tutorial shows you how to perform queries the manual way first, and then you'll see how Mappers can make life easier.

NOTE: It is important to realize that you can create multiple Mappers for the same domain objects. The default name of `CustomerMapper.FULL` indicates that the autocreated Mapper perform full joins. You can add a `CustomerMapper.SIMPLE` to be used in SELECTs where you don't need joins. You can also use the `FULL` Mapper with a manual SELECT that does not perform any joins, even if it mentions fields not in the ResultSet.

6.Tips and tricks

Some tricks are used across all query operations, and we'll briefly talk about some of them here.

6.1.2 Appending SQL to a query

When you first create a Query object, you can choose to give it some SQL right away, or you can add SQL using the add command. Both commands support varargs as well, so these are all equivalent:

```
Query.create("SELECT * FROM country WHERE id = 1");

Query.create("SELECT *",
         "FROM country",
         "WHERE id = 1");

Query.create()
.add("SELECT *")
.add("FROM country")
.add("WHERE id = 1");
```

Tornado Query will automatically introduce whitespace where needed. There are lots of mutators like `addIf`, `addUnless`, `repeat` etc that are further described in the SELECT part of this tutorial.

7 INSERT

7.1 INSERT

7.1.1 Manual INSERT

Let's start with manually inserting a `Country` in our example database. For this, we don't need a Mapper, and the code is straight forward:

```
Query.create("INSERT INTO country (name) VALUES ('Norway')").insert();
```

Let's supply the country name as parameter instead, to avoid SQL injection if the country came from an untrusted source:

```
Query.create("INSERT INTO country (name) VALUES (:name)")
    .param("name", "Norway").insert();
```

We might want to insert the data from a domain object:

```
Country country = new Country("Norway");
Query.create("INSERT INTO country (name) VALUES (:name)").param(country).insert();
```

The country object was set as the root parameter, so that we can look up values using the object's properties directly. We can also supply the country as a named parameter:

```
Query.create("INSERT INTO country (name) VALUES (:country.name)")
    .param("country", country).insert();
```

Since our database schema for the country table dictated a default value for the id field from the `country_id` SEQUENCE, we have automatically gotten an id, but we had no way of retrieving it. We can manually increment the sequence and put the result in the id property of our `Country` object:

```
Query.create("INSERT INTO country (name) VALUES (:name)")
    .param(country)
    .key("id", "SELECT nextval('country_id')")
    .insert();
```

After the insert is performed, `country.id` will contain the id from the sequence. There is a shortcut for the `key` method called `sequence`:

```
Query.create("INSERT INTO country (name) VALUES (:name)")
    .param(country)
    .sequence("id", "country_id")
    .insert();
```

7.1.2 MySQL LAST_INSERT_ID

After performing an INSERT against MySQL you can use the convenience call `Query.lastInsertId()`.

7.1.3 INSERT with a Mapper

Take a good look at the `countryMapper` we created earlier. You can see that it already mentions the table name and both id and name fields, and it even knows about the sequence for the id field. We can take advantage of that to have it automatically write the `INSERT` statement for us:

```
Query.insert(countryMapper, country);
```

Wow, that was easy! And really, there is no magic here - writing an INSERT is mostly about enumerating columns and giving them the right values, so you wouldn't want to do that manually unless there is something very spesific and special about your usecase. Keep it DRY!

NOTE: If you use a mapper that has join columns, Tornado Query is smart enough to understand what columns are native to the table you are inserting into, and skips those that describe columns in other tables. Tornado Query only saves your main object, there is no magical traversal down to child objects that don't have ids etc. Tornado Query is here to help you, but it refuses to perform magic tricks :)

8 UPDATE

8.1 UPDATE

8.1.1 Manual UPDATE

Again we start with manually updating a `Country` in our example database.

```
Query.create("UPDATE country SET name = 'Norway' WHERE id = 1").update();
```

Again, to avoid SQL injection we'll send in the values as parameters:

```
Query.create("UPDATE country SET name = :name WHERE id = :id")
    .param("name", "Norway").param("id", 1).update();
```

Update an existing domain object:

```
Query.create("UPDATE country SET name = :name WHERE id = :id")
    .param(country).update();
```

We can also supply the country as a named parameter:

```
Query.create("UPDATE country SET name = :country.name WHERE id = :country.id")
    .param("country", country).update();
```

8.1.2 UPDATE with a Mapper

As with `INSERT`, Tornado Query can automatically write the `UPDATE` statement for us:

```

Query.update(countryMapper, country);
```

Since the countryMapper was created with an `id` method, it knows how to construct the `WHERE` clause for the update. This is true even if there are multiple id fields in the Mapper.

NOTE: The `Query.update()` method will return the number of rows that was altered.

9 DELETE

9.1 DELETE

9.1.1 Manual DELETE

We'll delete a `Country` in our example database. First inline:

```
Query.create("DELETE FROM country WHERE id = 1").delete();
```

Again, to avoid SQL injection we'll send in the id as a parameter:

```
Query.create("DELETE FROM country WHERE id = :id").param("id", 1).delete();
```

Delete using an existing domain object:

```
Query.create("DELETE FROM country WHERE id = :id")
    .param(country).delete();
```

We can also supply the country as a named parameter:

```
Query.create("DELETE FROM country WHERE id = :country.id")
    .param("country", country).delete();
```

9.1.2 DELETE with a Mapper

As with `INSERT` and `UPDATE`, Tornado Query can automatically write the `DELETE` statement for us:

```

Query.delete(countryMapper, country);
```

Since the countryMapper was created with an `id` method, it knows how to construct the `WHERE` clause for the delete. This is true even if there are multiple id fields in the Mapper.

NOTE: The `Query.delete()` method will return the number of rows that was deleted.

10 SELECT

10.1 SELECT

10.1.1 SELECT without a Mapper and no domain object

Even when you don't use a Mapper, one is automatically created behind the scenes. The Mapper is then primarily used for caching information so that subsequent operations can be performed faster. That being said, it is always recommended that you use a predefined, `static final` Mapper for best performance.

If you don't supply a Mapper, and maybe you don't even have a domain object, you can get the results from the database as either a `HashMap` or a `List<HashMap>`. This can be OK for small operations:

```
Map country = Query.create(HashMap.class, "SELECT * FROM country WHERE id = 1").first();
```

The `first()` method will retrieve the first row that matches your query and turn the ResultSet into a HashMap. You can also obtain and use a new Query element this way:

```
Map country = new Query<HashMap>().select("SELECT * FROM country WHERE id = 1").first()
```

It is recommended to use the static `create` approach, as it has more convenience methods. Let's send in the `id` value as a parameter, and return a list of Countries using the `rows()` method:

```
Map country = Query.create(HashMap.class)
    .add("SELECT * FROM country WHERE id < 100")
    .param("id", 100)
    .rows();
```

10.1.2 SELECT without a Mapper, but with domain object

Even without a Mapper, Tornado Query can still convert the ResultSet into your domain object. You will only get columns mapped to fields where the name is the same, or where underscores in the columns can be swapped out for camelCase syntax in the domain object. So you can write:

```
Country country = Query.create(Country.class,
    "SELECT * FROM country WHERE id = 1").first();
```

Though this works, it is highly recommended to create a Mapper for your domain objects.

10.1.3 SELECT with a Mapper

This is where Tornado Query shines! The basic select is then turned into:

```
Country country = Query.select(countryMapper).where("id = :id").param("id", 1).first();
```

There is also a shortcut to select by id:

```
Country country = Query.byId(countryMapper, 1).first();
```

10.1.4 Joins

Let's query for a Customer object instead. First we'll insert a country and an address, along with a customer. Then we'll perform some queries against this customer.

```
Country usa = new Country(1, 'USA');
Query.insert(countryMapper, usa);

Address address = new Address("Sesame Street", "10001", "New York", usa);
Query.insert(addressMapper, address);

Customer customer = new Customer("Edvin Syse", "my@email.addr");
customer.setDeliveryAddress(address);
customer.setBillingAddress(address);
Query.insert(customerMapper, customer);
```

First we insert the country. No surprises there. Next up is the address, which references the country. Lastly, we insert a customer, which references the address two times, for the `deliveryAddress` and `billingAddress` fields.

Now let's select a customer. Since our `customerMapper` references the `addressMapper` , and the `addressMapper` references our `countryMapper`, we'll get automatic joins if we select using the `customerMapper`.

```
Customer customer = Query.byId(customerMapper, 1).first();
```

The above select query will turn into the same SQL sentence you saw in the Mappers introduction.

10.1.5 Dynamic mutators

Sometimes, part of the mutators are only supposed to be applied depending on certain conditions. Concider the following query:

```
Customer customer = Query.select(customerMapper)
    .where("id = :id")
    .and("name = :name")
    .param(...)
    .first();
```

Firstly, the `and("name = :name")` is just a shortcut for writing `add("AND name = :name")`. Let's suppose that this query was part of a DAO query, where both id and name was optional parameters. Then it would make sense to only add the mutators that was actually sent to the method:

```
public Customer getCustomer(Integer id, String name) {
    return Query.select(customerMapper)
        .where()
        .addIf(id != null, "id = :id")
        .addIf(name != null, "name = :name")
        .param("id", id)
        .param("name", name)
        .first();
}
```

The `where()` method will make sure that a `WHERE` expression is added to the statement if any additional mutators are added. Second, the `addIf()` will only add the following sentence to the query if the expression is true. There is an `addUnless` method as well, to make your code more readable.

NOTE: If you dislike this approach, you can create your query any way you like, using any programmatic approach you feel comfortable with, as long as the named parameters you mentioned in your SQL sentence is added as `param()`'s to your query.

This makes sure that even the corner cases can be handled elegantly. The Query class accepts normal SQL, there is no magic, so you should never need to drop out to a plain old Connection. If you need it though, it's still available via `Query.connection.get()`.

10.1.6 Repeaters

Especially when performing "search" queries, we often repeat part of the SQL sentence multiple times. Tornado Query can express this elegantly. Concider the following SQL query:

```
SELECT  *
FROM    users
WHERE   UPPER(username) LIKE UPPER('%min')
OR      UPPER(username) LIKE UPPER('%mon%')
OR      UPPER(username) LIKE UPPER('man')
```

This is how you would write it using Tornado Query:

```
List<String> usernames = Arrays.asList("%min", "%mon%", "man");

List<User> users = Query.create(User.class)
    .select().from("users").where()
    .repeat("OR", "UPPER(username) LIKE UPPER(:usernames[])")
    .param("usernames", usernames)
    .rows();
```

Here we didn't use a resultMap, and selected straight from a fictive table called `users`. The `repeat()` method made sure that each entry in the `usernames` list would result in another `LIKE` query for that particular username, glued together with an `OR`.

10.1.7 Want more?

We have just scratched the surface of what you can do with Tornado Query. Consult the JavaDoc for more information or email me at es@syse.no. If you have any questions or would like to see additional stuff covered in this tutorial.

11 Logging

11.1 Logging

Tornado Query uses SLF4J for logging. You should provide a `log4j.properties` in your application to make sure you receive the wanted logging information.

A good starting point is this configuration:

```
log4j.rootLogger=ERROR,A1

log4j.logger.org.apache.commons.beanutils.converters=ERROR

log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} [%t] %-5p %c %x - %m%n
```

If you want to output the SQL statement that Tornado Query executes, you can set the log level to `DEBUG`. You can also retrieve the SQL to be executed by printing out the statement created by the `prepare` method:

```
String sql = Query.byId(CustomerMapper.FULL, 1).prepare().toString()
```