Getting Started With jOOQ

jOOQ provides you with a Java DSL that enables you to build SQL queries in a comfortable and type-safe way. It abstracts the technical difficulties of plain JDBC and handles the subtle differences of the various SQL dialects.

Required Dependencies

Before you can use jOOQ in your project, you need to add a few dependencies to it. The following code snippet shows the maven dependencies of the jOOQ community edition, which you can use with open-source databases. If you use other databases, like Oracle or SQL Server, you need to get a commercial license of jOOQ.

```
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq</artifactId>
  <version>${version.jooq}</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>${version.jooq}</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>${version.jooq}</version>
</dependency>
```

Code Generation

The code generation step is optional but I highly recommend it. If you want to improve developer productivity and write your queries in a type-safe way, you should use jOOQ's code generator. It creates Java classes that map your tables, sequences, stored procedures and more. You can use these classes to define your queries and to process the selected results.

jOOQ provides you with a set of code generators that you can use on the command line, within Eclipse and as a Maven plugin. They can generate jOOQ's metamodel classes based on an existing database, an SQL script or your entity mappings.

Here is an example of a Maven build configuration that calls the code generator within Maven's generate goal. The generator connects to the public schema of the jooq database on a PostgreSQL server on localhost. It writes the generated classes to the package *org.thoughts.on.java.db* in the folder target*/generated-sources/jooq*.

```
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <version>${version.jooq}</version>

    <executions>
        <execution>
          <goals>
            <goal>generate</goal>
          </goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
          <groupId>org.postgresql</groupId>
          <artifactId>postgresql</artifactId>
          <version>9.4.1208</version>
        </dependency>
    </dependencies>

    <configuration>
        <jdbc>
          <driver>org.postgresql.Driver</driver>
          <url>jdbc:postgresql:jOOQ</url>
          <user>postgres</user>
          <password>postgres</password>
        </jdbc>
        <generator>
          <database>
            <name>org.jooq.util.postgres.PostgresDatabase</name>
            <includes>.*</includes>
            <excludes></excludes>
            <inputSchema>public</inputSchema>
          </database>
          <target>
            <packageName>org.thoughts.on.java.db</packageName>
            <directory>target/generated-sources/jooq</directory>
          </target>
        </generator>
    </configuration>
</plugin>
```

Implementing Queries with jOOQ

The great thing about jOOQ is that the DSL is very similar to SQL's syntax. So, if you're familiar with SQL, you will have no problems to write your queries with jOOQ.

It all starts with the creation of a *DSLContext* which you need to initialize with a JDBC *Connection* and the *SQLDialect* you want to use. In this example, I'm using a PostgreSQL 9.4 database on localhost.

```
String user = "postgres";
String pass = "postgres";
String url = "jdbc:postgresql:jOOQ";

// Create a JDBC Connection
try (Connection conn = DriverManager.getConnection(url, user, pass))
{
    // Create a context for your database
    DSLContext ctx = DSL.using(conn, SQLDialect.POSTGRES_9_4);

    // Do something useful ...

} catch (Exception e) {
    e.printStackTrace();
}
```

A Simple Example

Let's select the first name, last name and the number of books written by all authors whose last name starts with "Jan" and ends with "en".

As you can see, the Java code looks extremely similar to the SQL statement you want to create.

```
Result<Record3<String, String, Integer>> result =
        ctx.select(
                AUTHOR.FIRSTNAME,
                AUTHOR.LASTNAME,
                DSL.count(BOOK_AUTHOR.BOOKID).as("bookCount"))
            .from(AUTHOR)
                .leftJoin(BOOK_AUTHOR)
                    .on(AUTHOR.ID.eq(BOOK_AUTHOR.AUTHORID))
            .where(AUTHOR.LASTNAME.like("Jan%en"))
            .groupBy(AUTHOR.FIRSTNAME, AUTHOR.LASTNAME)
            .fetch();
for (Record r : result) {
    String firstName = r.get(AUTHOR.FIRSTNAME);
    String lastName = r.get(AUTHOR.LASTNAME);
    Integer bookCount = (Integer) r.get("bookCount");
    System.out.println(firstName + " " + lastName + " wrote "
         + bookCount + " book(s).");
}
```

The select method defines the projection. I use the generated *Author* class to reference the *firstname* and *lastname* columns of the *author* table. jOOQ's *DSL* class provides lots of methods that enable you to call SQL functions. I use it here to call SQL's *count* function and define the alias *bookCount* for that field.

Then I define the FROM clause of the query. The *from* method returns a *SelectJoinStep* interface which enables you to define different types of join clauses or to combine the results of multiple queries with set operators, like UNION or INTERSECT.

Let's continue by specifying the WHERE clause. You can do that by calling the *where* method with a *String*, an SQL query part, or one or more *Conditions*. I prefer to define my query in a type-safe way, so I use the generated *Book* class to reference the *lastname* column and to define the *like* expression. As you can see in the code snippet, I don't define a bind parameter and just set the *String* "Jan%en" as its value. jOOQ automatically adds a bind parameter to the query and sets the provided value as the bind parameter value.

OK, almost done. We just need to add a GROUP BY clause for the columns *firstname* and *lastname*. Similar to the definition of the pervious clauses, you can do that by calling the *groupBy* method with references to the 2 database columns.

That's all you need to do to define the query. The call of the *fetch* method executes the query and returns a *Result* interface which contains a collection of strongly typed *Record* interfaces. As in the previous example, you can then use that interface and jOOQ's generated classes to process the query result.