Querydsl

Reference Documentation

Timo Westkämper
Samppa Saarela
Vesa Marttila
Lassi Immonen

Querydsl: Reference Documentation

by Timo Westkämper, Samppa Saarela, Vesa Marttila, and Lassi Immonen

2.3.0

Legal Notice

Table of Contents

Preface

Querydsl is a framework which enables the construction of statically typed SQL-like queries. Instead of writing queries as inline strings or externalizing them into XML files they can be constructed via a fluent API like Querydsl.

The benefits of using a fluent API in comparison to simple strings are for example

1. code completion in IDE
2. almost none syntactically invalid queries allowed
3. domain types and properties can be referenced safely
4. adopts better to refactoring changes in domain types

1. Introduction

1.1. Background

Querydsl was born out of the need to maintain HQL queries in a typesafe way. Incremental construction of HQL queries requires String concatenation and results in hard to read code. Unsafe references to domain types and properties via plain Strings were another issue with String based HQL construction.

With a changing domain model type-safety brings huge benefits in software development. Domain changes are directly reflected in queries and autocomplete in query construction makes query construction faster and safer.

HQL for Hibernate was the first target language for Querydsl, but nowadays it supports JPA, JDO, JDBC, Lucene, Hibernate Search, MongoDB, Collections and RDFBean as backends.

1.2. Principles

Type safety is the core principle of Querydsl. Queries are constructed based on generated query types that reflect the properties of your domain types. Also function/method invocations are constructed in a fully type-safe manner.

Consistency is another important principle. The query paths and operations are the same in all implementations and also the Query interfaces have a common base interface.

All query instances can be reused multiple times. After the projection the paging data (limit and offset) and the definition of the projection are removed.

To get an impression of the expressivity of the Querydsl query and expression types go to the javadocs and explore com.mysema.query.Query, com.mysema.query.Projectable and com.mysema.query.types.Expression.

2. Tutorials

Instead of a general Getting started guide we provide integration guides for the main backends of Querydsl.

2.1. Querying JPA

Querydsl defines a general statically typed syntax for querying on top of persisted domain model data. JDO and JPA are the primary integration technologies for Querydsl. This guide describes how to use Querydsl in combination with JPA/Hibernate.

Querydsl for JPA/Hibernate is an alternative to both JPQL and Criteria queries. It combines the dynamic nature of Criteria queries with the expressiveness of JPQL and all that in a fully typesafe manner.

Maven integration

Add the following dependencies to your Maven project and make sure that the Maven 2 repo of Mysema Source (http://source.mysema.com/maven2/releases) is accessible from your POM :

```
<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-apt</artifactId>
  <version>${querydsl.version}</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-jpa</artifactId>
  <version>${querydsl.version}</version>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.6.1</version>
</dependency>
```

And now, configure the Maven APT plugin :

```
<project>
  <build>
  <plugins>
    ...
    <plugin>
      <groupId>com.mysema.maven</groupId>
      <artifactId>maven-apt-plugin</artifactId>
      <version>1.0</version>
      <executions>
        <execution>
          <goals>
```

```
            <goal>process</goal>
          </goals>
          <configuration>
            <outputDirectory>target/generated-sources/java</outputDirectory>
            <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>
          </configuration>
        </execution>
      </executions>
    </plugin>
    ...
  </plugins>
  </build>
</project>
```

The JPAAnnotationProcessor finds domain types annotated with the javax.persistence.Entity annotation and generates query types for them.

If you use Hibernate annotations in your domain types you should use the APT processor com.mysema.query.apt.hibernate.HibernateAnnotationProcessor instead.

Run clean install and you will get your Query types generated into target/generated-sources/java.

If you use Eclipse, run mvn eclipse:eclipse to update your Eclipse project to include target/generated-sources/java as a source folder.

Now you are able to construct JPQL query instances and instances of the query domain model.

Ant integration

Place the jar files from the full-deps bundle on your classpath and use the following tasks for Querydsl code generation :

```
    <!-- APT based code generation -->
    <javac srcdir="${src}" classpathref="cp">
      <compilerarg value="-proc:only"/>
      <compilerarg value="-processor"/>
      <compilerarg value="com.mysema.query.apt.jpa.JPAAnnotationProcessor"/>
      <compilerarg value="-s"/>
      <compilerarg value="${generated}"/>
    </javac>

    <!-- compilation -->
    <javac classpathref="cp" destdir="${build}">
      <src path="${src}"/>
      <src path="${generated}"/>
    </javac>
```

Replace *src* with your main source folder, *generated* with your folder for generated sources and *build* with your target folder.

Generating the model from hbm.xml files

If you are using Hibernate with an XML based configuration, you can use the XML metadata to create your Querydsl model.

com.mysema.query.jpa.hibernate.HibernateDomainExporer provides the functionality for this :

```
HibernateDomainExporter exporter = new HibernateDomainExporter(
  "Q",                    // name prefix
  new File("target/gen3"), // target folder
  configuration);          // instance of org.hibernate.cfg.Configuration

exporter.export();
```

The HibernateDomainExporter needs to be executed within a classpath where the domain types are visible, since the property types are resolved via reflection.

All JPA annotations are ignored, but Querydsl annotations such as @QueryInit and @QueryType are taken into account.

Using query types

To create queries with Querydsl you need to instantiate variables and Query implementations. We will start with the variables.

Let's assume that your project has the following domain type :

```
@Entity
public class Customer {
    private String firstName;
    private String lastName;

    public String getFirstName(){
        return firstName;
    }

    public String getLastName(){
        return lastName;
    }

    public void setFirstName(String fn){
        firstName = fn;
    }

    public void setLastName(String ln)[
        lastName = ln;
    }
}
```

Querydsl will generate a query type with the simple name QCustomer into the same package as Customer. QCustomer can be used as a statically typed variable in Querydsl queries as a representative for the Customer type.

QCustomer has a default instance variable which can be accessed as a static field :

```
QCustomer customer = QCustomer.customer;
```

Alternatively you can define your own Customer variables like this :

```
QCustomer customer = new QCustomer("myCustomer");
```

Querying

The Querydsl JPA module supports both the JPA and the Hibernate API.

To use the Hibernate API you use HibernateQuery instances for your queries like this :

```
// where session is a Hibernate session
JPQLQuery query = new HibernateQuery (session);
```

If you are using the JPA API instead, you can instantiate a JPAQuery like this :

```
// where entityManager is a JPA EntityManager
JPQLQuery query = new JPAQuery (entityManager);
```

Both HibernateQuery and JPAQuery implement the JPQLQuery interface.

The default configuration of JPAQuery is best suited for use with Hibernate. Whe using EclipseLink as the JPA provider, the JPAQuery should be constructed like this :

```
JPQLQuery query = new JPAQuery (entityManager, EclipseLinkTemplates.DEFAULT);
```

If you want to use the standard JPA 2 serialization then create the query like this :

```
JPQLQuery query = new JPAQuery (entityManager, JPQLTemplates.DEFAULT);
```

To retrieve the customer with the first name Bob you would construct a query like this :

```
QCustomer customer = QCustomer.customer;
JPQLQuery query = new HibernateQuery (session);
Customer bob = query.from(customer)
  .where(customer.firstName.eq("Bob"))
```

```
  .uniqueResult(customer);
```

The from call defines the query source, the where part defines the filter and uniqueResult defines the projection and tells Querydsl to return a single element. Easy, right?

To create a query with multiple sources you use the query like this :

```
query.from(customer, company);
```

And to use multiple filters use it like this

```
query.from(customer)
    .where(customer.firstName.eq("Bob"), customer.lastName.eq("Wilson"));
```

Or like this

```
query.form(customer)
    .where(customer.firstName.eq("Bob").and(customer.lastName.eq("Wilson")));
```

In native JPQL form the query would be written like this :

```
from Customer as customer
    where customer.firstName = "Bob" and customer.lastName = "Wilson"
```

If you want to combine the filters via "or" then use the following pattern

```
query.form(customer)
    .where(customer.firstName.eq("Bob").or(customer.lastName.eq("Wilson")));
```

Using joins

Querydsl supports the following join variants in JPQL : inner join, join, left join and full join. Join usage is typesafe, and follows the following pattern :

```
query.from(cat)
    .innerJoin(cat.mate, mate)
    .leftJoin(cat.kittens, kitten)
    .list(cat);
```

The native JPQL version of the query would be

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

Another example

```
query.from(cat)
    .leftJoin(cat.kittens, kitten)
    .with(kitten.bodyWeight.gt(10.0))
    .list(cat);
```

With the following JPQL version

```
from Cat as cat
    left join cat.kittens as kitten
    with kitten.bodyWeight > 10.0
```

General usage

Use the the cascading methods of the JPQLQuery interface like this

from : Define the query sources here.

innerJoin, join, leftJoin, fullJoin, with : Define join elements using these constructs. For the join methods the first argument is the join source and the second the target (alias).

where : Define the query filters, either in varargs form separated via commas or cascaded via the and-operator.

groupBy : Define the group by arguments in varargs form.

having : Define the having filter of the "group by" grouping as an varags array of Predicate expressions.

orderBy : Define the ordering of the result as an varargs array of order expressions. Use asc() and desc() on numeric, string and other comparable expression to access the OrderSpecifier instances.

limit, offset, restrict : Define the paging of the result. Limit for max results, offset for skipping rows and restrict for defining both in one call.

Ordering

The syntax for declaring ordering is

```
query.from(customer)
    .orderBy(customer.lastName.asc(), customer.firstName.desc())
    .list(customer);
```

which is equivalent to the following native JPQL

```
from Customer as customer
```

```
    order by customer.lastName asc, customer.firstName desc
```

Grouping

Grouping can be done in the following form

```
query.from(customer)
    .groupBy(customer.lastName)
    .list(customer.lastName);
```

which is equivalent to the following native JPQL

```
select customer.lastName
    from Customer as customer
    group by customer.lastName
```

Delete clauses

Delete clauses in Querydsl JPA follow a simple delete-where-execute form. Here are some examples :

```
QCustomer customer = QCustomer.customer;
// delete all customers
new HibernateDeleteClause(session, customer).execute();
// delete all customers with a level less than 3
new HibernateDeleteClause(session, customer).where(customer.level.lt(3)).execute();
```

The second parameter of the HibernateDeleteClause constructor is the entity to be deleted. The where call is optional and the execute call performs the deletion and returns the amount of deleted entities.

For JPA based Delete usage, use the JPADeleteClause instead.

Update clauses

Update clauses in Querydsl JPA follow a simple update-set/where-execute form. Here are some examples :

```
QCustomer customer = QCustomer.customer;
// rename customers named Bob to Bobby
new HibernateUpdateClause(session, customer).where(customer.name.eq("Bob"))
    .set(customer.name, "Bobby")
    .execute();
```

The second parameter of the HibernateUpdateClause constructor is the entity to be updated. The set invocations define the property updates in SQL-Update-style and the execute call performs the Update and returns the amount of updated entities.

For JPA based Update usage, use the JPAUpdateClause instead.

Subqueries

To create a subquery you create a HibernateSubQuery instance, define the query parameters via from, where etc and use unique or list to create a subquery, which is just a type-safe Querydsl expression for the query. unique is used for a unique (single) result and list for a list result.

```
query.from(department)
    .where(department.employees.size().eq(
        new HibernateSubQuery().from(d).unique(d.employees.size().max())
     )).list(department);
```

Another example

```
query.from(employee)
    .where(employee.weeklyhours.gt(
        new HibernateSubQuery().from(employee.department.employees, e)
        .where(e.manager.eq(employee.manager))
        .unique(e.weeklyhours.avg())
    )).list(employee);
```

For JPA based sub query usage, use the JPASubQuery instead.

Exposing the original query

If you need to do tune the original Query before the execution of the query you can expose it like this :

```
HibernateQuery query = new HibernateQuery(session);
org.hibernate.Query hibQuery = query.from(employee).createQuery(employee);
hibQuery.setResultTransformer(someTransformer);
List results = hibQuery.list();
```

Using Native SQL in Hibernate queries

Querydsl supports Native SQL in Hibernate via the HibernateSQLQuery class.

To use it, you must generate Querydsl query types for your SQL schema. This can be done for example with the following Maven configuration :

```
<plugin>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-maven-plugin</artifactId>
  <version>${project.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>export</goal>
      </goals>
    </execution>
```

```
  </executions>
  <configuration>
    <jdbcDriver>org.apache.derby.jdbc.EmbeddedDriver</jdbcDriver>
    <jdbcUrl>jdbc:derby:target/demoDB;create=true</jdbcUrl>
    <packageName>com.mycompany.mydomain</packageName>
    <targetFolder>${project.basedir}/target/generated-sources/java</targetFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>${derby.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

When the query types have successfully been generated into the location of your choice, you can use them in your queries.

Single column query :

```
// serialization templates
SQLTemplates templates = new DerbyTemplates();
// query types (S* for SQL, Q* for domain types)
SAnimal cat = new SAnimal("cat");
SAnimal mate = new SAnimal("mate");
QCat catEntity = QCat.cat;

HibernateSQLQuery query = new HibernateSQLQuery(session, templates);
List<String> names = query.from(cat).list(cat.name);
```

Query multiple columns :

```
query = new HibernateSQLQuery(session, templates);
List<Object[]> rows = query.from(cat).list(cat.id, cat.name);
```

Query all columns :

```
List<Object[]> rows = query.from(cat).list(cat.all());
```

Query in SQL, but project as entity :

```
query = new HibernateSQLQuery(session, templates);
List<Cat> cats = query.from(cat).orderBy(cat.name.asc()).list(catEntity);
```

Query with joins :

```
query = new HibernateSQLQuery(session, templates);
```

```
cats = query.from(cat)
    .innerJoin(mate).on(cat.mateId.eq(mate.id))
    .where(cat.dtype.eq("Cat"), mate.dtype.eq("Cat"))
    .list(catEntity);
```

Query and project into DTO :

```
query = new HibernateSQLQuery(session, templates);
List<CatDTO> catDTOs = query.from(cat)
    .orderBy(cat.name.asc())
    .list(ConstructorExpression.create(CatDTO.class, cat.id, cat.name));
```

If you are using the JPA API instead of the Hibernate API, then use JPASQLQuery instead of HibernateSQLQuery

2.2. Querying JDO

Querydsl defines a general statically typed syntax for querying on top of persisted domain model data. JDO and JPA are the primary integration technologies for Querydsl. This guide describes how to use Querydsl in combination with JDO.

Maven integration

Add the following dependencies to your Maven project and make sure that the Maven 2 repo of Mysema Source (http://source.mysema.com/maven2/releases) is accessible from your POM if the version cannot yet be found in other public Maven repos :

```
<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-apt</artifactId>
  <version>${querydsl.version}</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-jdo</artifactId>
  <version>${querydsl.version}</version>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.6.1</version>
</dependency>
```

And now, configure the Maven APT plugin which generates the query types used by Querydsl :

```
<project>
```

```
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>com.mysema.maven</groupId>
        <artifactId>maven-apt-plugin</artifactId>
        <version>1.0</version>
        <executions>
          <execution>
            <goals>
              <goal>process</goal>
            </goals>
            <configuration>
              <outputDirectory>target/generated-sources/java</outputDirectory>
              <processor>com.mysema.query.apt.jdo.JDOAnnotationProcessor</processor>
            </configuration>
          </execution>
        </executions>
      </plugin>
    ...
    </plugins>
  </build>
</project>
```

The JDOAnnotationProcessor finds domain types annotated with the javax.jdo.annotations.PersistenceCapable annotation and generates Querydsl query types for them.

Run clean install and you will get your Query types generated into target/generated-sources/java.

If you use Eclipse, run mvn eclipse:eclipse to update your Eclipse project to include target/generated-sources/java as a source folder.

Now you are able to construct JDOQL query instances and instances of the query domain model.

Ant integration

Place the jar files from the full-deps bundle on your classpath and use the following tasks for Querydsl code generation :

```
    <!-- APT based code generation -->
    <javac srcdir="${src}" classpathref="cp">
      <compilerarg value="-proc:only"/>
      <compilerarg value="-processor"/>
      <compilerarg value="com.mysema.query.apt.jdo.JDOAnnotationProcessor"/>
      <compilerarg value="-s"/>
      <compilerarg value="${generated}"/>
    </javac>

    <!-- compilation -->
    <javac classpathref="cp" destdir="${build}">
      <src path="${src}"/>
      <src path="${generated}"/>
    </javac>
```

Replace *src* with your main source folder, *generated* with your folder for generated sources and *build* with your target folder.

Using query types

To create queries with Querydsl you need to instantiate variables and Query implementations. We will start with the variables.

Let's assume that your project has the following domain type :

```
@PersistenceCapable
public class Customer {
  private String firstName;
  private String lastName;

  public String getFirstName(){
    return firstName;
  }

  public String getLastName(){
    return lastName;
  }

  public void setFirstName(String fn){
    firstName = fn;
  }

  public void setLastName(String ln)[
    lastName = ln;
  }
}
```

Querydsl will generate a query type with the simple name QCustomer into the same package as Customer. QCustomer can be used as a statically typed variable in Querydsl as a representative for the Customer type.

QCustomer has a default instance variable which can be accessed as a static field :

```
QCustomer customer = QCustomer.customer;
```

Alternatively you can define your own Customer variables like this :

```
QCustomer customer = new QCustomer("myCustomer");
```

QCustomer reflects all the properties of the original type Customer as public fields. The firstName field can be accessed like this

```
customer.firstName;
```

Querying with JDOQL

For the JDOQL-module JDOQLQueryImpl is the main Query implementation. It is instantiated like this :

```
PersistenceManager pm = ...;
JDOQLQuery query = new JDOQLQueryImpl (pm);
```

To retrieve the customer with the first name Bob you would construct a query like this :

```
QCustomer customer = QCustomer.customer;
JDOQLQuery query = new JDOQLQueryImpl (pm);
Customer bob = query.from(customer)
  .where(customer.firstName.eq("Bob"))
  .uniqueResult(customer);
query.close();
```

The from call defines the query source, the where part defines the filter and uniqueResult defines the projection and tells Querydsl to return a single element. Easy, right?

To create a query with multiple sources you just use the JDOQLQuery interface like this :

```
query.from(customer, company);
```

And to use multiple filters use it like this

```
query.from(customer)
    .where(customer.firstName.eq("Bob"), customer.lastName.eq("Wilson"));
```

Or like this

```
query.form(customer)
    .where(customer.firstName.eq("Bob").and(customer.lastName.eq("Wilson")));
```

If you want to combine the filters via "or" then use the following pattern

```
query.form(customer)
    .where(customer.firstName.eq("Bob").or(customer.lastName.eq("Wilson")));
```

General usage

Use the the cascading methods of the JDOQLQuery interface like this

from : Define the query sources here, the first argument becomes the main source and the others are treated as variables.

where : Define the query filters, either in varargs form separated via commas or cascaded via the and-operator.

groupBy : Define the group by arguments in varargs form.

having : Define the having filter of the "group by" grouping as an varargs array of Predicate expressions.

orderBy : Define the ordering of the result as an varargs array of order expressions. Use asc() and desc() on numeric, string and other comparable expression to access the OrderSpecifier instances.

limit, offset, restrict : Define the paging of the result. Limit for max results, offset for skipping rows and restrict for defining both in one call.

Ordering

The syntax for declaring ordering is

```
query.from(customer)
    .orderBy(customer.lastName.asc(), customer.firstName.desc())
    .list(customer);
```

Grouping

Grouping can be done in the following form

```
query.from(customer)
    .groupBy(customer.lastName)
    .list(customer.lastName);
```

Delete clauses

Delete clauses in Querydsl JDOQL follow a simple delete-where-execute form. Here are some examples :

```
QCustomer customer = QCustomer.customer;
// delete all customers
new JDOQLDeleteClause(pm, customer).execute();
// delete all customers with a level less than 3
new JDOQLDeleteClause(pm, customer).where(customer.level.lt(3)).execute();
```

The second parameter of the JDOQLDeleteClause constructor is the entity to be deleted. The where call is optional and the execute call performs the deletion and returns the amount of deleted entities.

Subqueries

To create a subquery you create a JDOQLSubQuery instance, define the query parameters via from, where etc and use unique or list to create a subquery, which is just a type-safe Querydsl expression for the query. unique is used for a unique result and list for a list result.

```
query.from(department)
    .where(department.employees.size().eq(
        new JDOQLSubQuery().from(d).unique(AggregationFunctions.max(d.employees.size()))
     )).list(department);
```

represents the following native JDOQL query

```
SELECT this FROM com.mysema.query.jdoql.models.company.Department
WHERE this.employees.size() ==
(SELECT max(d.employees.size()) FROM com.mysema.query.jdoql.models.company.Department d)
```

Another example

```
query.from(employee)
    .where(employee.weeklyhours.gt(
        new JDOQLSubQuery().from(employee.department.employees, e)
        .where(e.manager.eq(employee.manager))
        .unique(AggregationFunctions.avg(e.weeklyhours))
    )).list(employee);
```

which represents the following native JDOQL query

```
SELECT this FROM com.mysema.query.jdoql.models.company.Employee
WHERE this.weeklyhours >
(SELECT avg(e.weeklyhours) FROM this.department.employees e WHERE e.manager == this.manager)
```

Using Native SQL

Querydsl supports Native SQL in JDO via the JDOSQLQuery class.

To use it, you must generate Querydsl query types for your SQL schema. This can be done for example with the following Maven configuration :

```
<plugin>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-maven-plugin</artifactId>
  <version>${project.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>export</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <jdbcDriver>org.apache.derby.jdbc.EmbeddedDriver</jdbcDriver>
    <jdbcUrl>jdbc:derby:target/demoDB;create=true</jdbcUrl>
    <packageName>com.mycompany.mydomain</packageName>
```

```
    <targetFolder>${project.basedir}/target/generated-sources/java</targetFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>${derby.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

When the query types have successfully been generated into the location of your choice, you can use them in your queries.

Single column query :

```
// serialization templates
SQLTemplates templates = new DerbyTemplates();
// query types (S* for SQL, Q* for domain types)
SAnimal cat = new SAnimal("cat");
SAnimal mate = new SAnimal("mate");

JDOSQLQuery query = new JDOSQLQuery(pm, templates);
List<String> names = query.from(cat).list(cat.name);
```

Query multiple columns :

```
query = new JDOSQLQuery(pm, templates);
List<Object[]> rows = query.from(cat).list(cat.id, cat.name);
```

Query all columns :

```
List<Object[]> rows = query.from(cat).list(cat.all());
```

Query with joins :

```
query = new JDOSQLQuery(pm, templates);
cats = query.from(cat)
    .innerJoin(mate).on(cat.mateId.eq(mate.id))
    .where(cat.dtype.eq("Cat"), mate.dtype.eq("Cat"))
    .list(catEntity);
```

Query and project into DTO :

```
query = new JDOSQLQuery(pm, templates);
List<CatDTO> catDTOs = query.from(cat)
    .orderBy(cat.name.asc())
```

```
    .list(ConstructorExpression.create(CatDTO.class, cat.id, cat.name));
```

2.3. Querying SQL

This chapter describes the query type generation and querying functionality of the SQL module.

Creating the query types

To get started export your schema into Querydsl query types like this :

```
java.sql.Connection conn = ...;
MetaDataExporter exporter = new MetaDataExporter();
exporter.setPackageName("com.myproject.mydomain");
exporter.setTargetFolder(new File("src/main/java"));
exporter.export(conn.getMetaData());
```

This declares that the database schema is to be mirrored into the com.myproject.domain package in the src/main/java folder.

The generated types have the table name transformed to mixed case as the class name and a similar mixed case transformation applied to the columns which are available as property paths in the query type.

In addition to this primary key and foreign key constraints are provided as fields which can be used for compact join declarations.

Maven integration

This functionality is also available as a Maven plugin. The presented example can be declared like this in the POM :

```
<plugin>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-maven-plugin</artifactId>
  <version>${querydsl.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>export</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <jdbcDriver>org.apache.derby.jdbc.EmbeddedDriver</jdbcDriver>
    <jdbcUrl>jdbc:derby:target/demoDB;create=true</jdbcUrl>
    <packageName>com.myproject.domain</packageName>
    <targetFolder>${project.basedir}/target/generated-sources/java</targetFolder>
 <!--
      optional elements :
      * jdbcUser = connection user
```

```
      * jdbcPassword = connection password
      * namePrefix = name prefix for generated query classes (default: Q)
      * schemaPattern = ant style pattern to restrict code generation to certain schemas (default: null)
      * tableNamePattern = ant style pattern to restrict code generation to certain tables (default: null)
      * exportBeans = set to true to generate beans as well, see section 2.14.13 (default: false)
      * innerClassesForKeys = set to true to generate inner classes for keys (default: false)
      * validationAnnotations = set to false to disable serialization of validation annotations (default: true)
    -->
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>${derby.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

Use the goal *test-export* to add the targetFolder as a test compile source root instead of a compile source root.

ANT integration

The ANT task com.mysema.query.sql.ant.AntMetaDataExporter of the querydsl-sql module provides the same functionality as an ANT task. The configuration parameters of the task are jdbcDriverClass, dbUrl, dbUserName, dbPassword, namePrefix, targetPackage, targetSourceFolder, schemaPattern, tableNamePattern, exportBeans, innerClassesForKeys and validationAnnotations.

Querying

Querying with Querydsl SQL is as simple as this :

```
QCustomer customer = new QCustomer("c");

SQLTemplates dialect = new HSQLDBTemplates(); // SQL-dialect
SQLQuery query = new SQLQueryImpl(connection, dialect);
List<String> lastNames = query.from(customer)
    .where(customer.firstName.eq("Bob"))
    .list(customer.lastName);
```

which is transformed into the following sql query, assuming that the related table name is *customer* and the columns *first_name* and *last_name* :

```
SELECT c.last_name
FROM customer c
WHERE c.first_name = 'Bob'
```

Querydsl uses SQL dialects to customize the SQL serialization needed for different relational databases. The available dialects are :

- DerbyTemplates

  - tested with version 10.5.3
- HSQLDBTemplates
  - tested with version 1.8.0.7
- H2Templates
  - tested with H2 1.2.133
- MySQLTemplates
  - tested with MySQL CE 5.1
- OracleTemplates
  - tested with Oracle 10g XE
- PostgresTemplates
  - tested with Postgres 8.4
- SQLServerTemplates
  - tested with SQL Server 2008

General usage

Use the the cascading methods of the SQLQuery interface like this

from : Define the query sources here.

innerJoin, join, leftJoin, fullJoin, on : Define join elements using these constructs. For the join methods the first argument is the join source and the second the target (alias).

where : Define the query filters, either in varargs form separated via commas or cascaded via the and-operator.

groupBy : Define the group by arguments in varargs form.

having : Define the having filter of the "group by" grouping as an varags array of Predicate expressions.

orderBy : Define the ordering of the result as an varargs array of order expressions. Use asc() and desc() on numeric, string and other comparable expression to access the OrderSpecifier instances.

limit, offset, restrict : Define the paging of the result. Limit for max results, offset for skipping rows and restrict for defining both in one call.

Joins

Joins are constructed using the following syntax :

```
query.from(customer)
    .innerJoin(customer.company, company)
    .list(customer.firstName, customer.lastName, company.name);
```

and for a left join :

```
query.from(customer)
    .leftJoin(customer.company, company)
    .list(customer.firstName, customer.lastName, company.name);
```

Alternatively the join condition can also be written out :

```
query.from(customer)
    .leftJoin(company).on(customer.company.eq(company.id))
    .list(customer.firstName, customer.lastName, company.name);
```

Ordering

The syntax for declaring ordering is

```
query.from(customer)
    .orderBy(customer.lastName.asc(), customer.firstName.asc())
    .list(customer.firstName, customer.lastName);
```

which is equivalent to the following native SQL

```
SELECT c.first_name, c.last_name
FROM customer c
ORDER BY c.last_name ASC, c.first_name ASC
```

Grouping

Grouping can be done in the following form

```
query.from(customer)
    .groupBy(customer.lastName)
    .list(customer.lastName);
```

which is equivalent to the following native SQL

```
```

```
SELECT c.last_name
FROM customer c
GROUP BY c.last_name
```

Using Subqueries

To create a subquery you create a SQLSubQuery instance, define the query parameters via from, where etc and use unique or list to create a subquery, which is just a type-safe Querydsl expression for the query. unique is used for a unique (single) result and list for a list result.

```
query.from(customer).where(
  customer.status.eq(new SQLSubQuery().from(customer2).unique(customer2.status.max()))
  .list(customer.all())
```

Another example

```
query.from(customer).where(
  customer.status.in(new SQLSubQuery().from(status).where(status.level.lt(3)).list(status.id))
  .list(customer.all())
```

Query extension support

Custom query extensions to support engine specific syntax can be created by subclassing AbstractSQLQuery and adding flagging methods like in the given MySQLQuery example :

```
public class MySQLQuery extends AbstractSQLQuery<MySQLQuery>{

    public MySQLQuery(Connection conn) {
        this(conn, new MySQLTemplates(), new DefaultQueryMetadata());
    }

    public MySQLQuery(Connection conn, SQLTemplates templates) {
        this(conn, templates, new DefaultQueryMetadata());
    }

    protected MySQLQuery(Connection conn, SQLTemplates templates, QueryMetadata metadata) {
        super(conn, new Configuration(templates), metadata);
    }

    public MySQLQuery bigResult(){
        return addFlag(Position.AFTER_SELECT, "SQL_BIG_RESULT ");
    }

    public MySQLQuery bufferResult(){
        return addFlag(Position.AFTER_SELECT, "SQL_BUFFER_RESULT ");
    }


    // ...
}
```

The flags are custom SQL snippets that can be inserted at specific points in the serialization. The supported positions are the enums of the com.mysema.query.QueryFlag.Position enum class.

Using DDL commands

CREATE TABLE commands can be used in fluent form via the CreateTableClause. Here are some examples :

```
createTable("language")
 .column("id", Integer.class).notNull()
 .column("text", String.class).size(256).notNull()
 .primaryKey("PK_LANGUAGE","id")
 .execute();

createTable("symbol")
 .column("id", Long.class).notNull()
 .column("lexical", String.class).size(1024).notNull()
 .column("datatype", Long.class)
 .column("lang", Integer.class)
 .column("intval",Long.class)
 .column("floatval",Double.class)
 .column("datetimeval",Timestamp.class)
 .primaryKey("PK_SYMBOL","id")
 .foreignKey("FK_LANG","lang").references("language","id")
 .execute();

createTable("statement")
 .column("model", Long.class)
 .column("subject", Long.class).notNull()
 .column("predicate", Long.class).notNull()
 .column("object", Long.class).notNull()
 .foreignKey("FK_MODEL","model").references("symbol","id")
 .foreignKey("FK_SUBJECT","subject").references("symbol","id")
 .foreignKey("FK_PREDICATE","predicate").references("symbol","id")
 .foreignKey("FK_OBJECT","object").references("symbol","id")
 .execute();
```

The factory method for CreateTableClause construction is :

```
    private CreateTableClause createTable(String tableName) {
        return new CreateTableClause(conn, templates, tableName);
    }
```

The constructor of CreateTableClause takes the connection, the templates and the table name. The rest is declared via column, primaryKey and foreignKey invocations.

Here are the corresponding CREATE TABLE clauses as they are executed.

```
CREATE TABLE language (
 id INTEGER NOT NULL,
 text VARCHAR(256) NOT NULL,
```

```
  CONSTRAINT PK_LANGUAGE PRIMARY KEY(id)
)

 CREATE TABLE symbol (
  id BIGINT NOT NULL,
  lexical VARCHAR(1024) NOT NULL,
  datatype BIGINT,
  lang INTEGER,
  intval BIGINT,
  floatval DOUBLE,
  datetimeval TIMESTAMP,
  CONSTRAINT PK_SYMBOL PRIMARY KEY(id),
  CONSTRAINT FK_LANG FOREIGN KEY(lang) REFERENCES language(id)
)

 CREATE TABLE statement (
  model BIGINT,
  subject BIGINT NOT NULL,
  predicate BIGINT NOT NULL,
  object BIGINT NOT NULL,
  CONSTRAINT FK_MODEL FOREIGN KEY(model) REFERENCES symbol(id),
  CONSTRAINT FK_SUBJECT FOREIGN KEY(subject) REFERENCES symbol(id),
  CONSTRAINT FK_PREDICATE FOREIGN KEY(predicate) REFERENCES symbol(id),
  CONSTRAINT FK_OBJECT FOREIGN KEY(object) REFERENCES symbol(id)
)
```

Using Data manipulation commands

All the DMLClause implementation in the Querydsl SQL module take three parameters, the Connection, the SQLTemplates instance used in the queries and the main entity the DMLClause is bound to.

Insert examples :

```
QSurvey survey = QSurvey.survey;

// with columns
new SQLInsertClause(conn, dialect, survey)
    .columns(survey.id, survey.name)
    .values(3, "Hello").execute();

// without columns
new SQLInsertClause(conn, dialect, survey)
    .values(4, "Hello").execute();

// with subquery
new SQLInsertClause(conn, dialect, survey)
    .columns(survey.id, survey.name)
    .select(new SQLSubQuery().from(survey2).list(survey2.id.add(1), survey2.name))
    .execute();

// with subquery, without columns
new SQLInsertClause(conn, dialect, survey)
    .select(new SQLSubQuery().from(survey2).list(survey2.id.add(10), survey2.name))
    .execute();
```

Update examples :

```
QSurvey survey = QSurvey.survey;

// update with where
new SQLUpdateClause(conn, dialect, survey)
    .where(survey.name.eq("XXX"))
    .set(survey.name, "S")
    .execute();

// update without where
new SQLUpdateClause(conn, dialect, survey)
    .set(survey.name, "S")
    .execute()
```

Delete examples :

```
QSurvey survey = QSurvey.survey;

// delete with where
new SQLDelecteClause(conn, dialect, survey)
    .where(survey.name.eq("XXX"))
    .execute();

// delete without where
new SQLDelecteClause(conn, dialect, survey)
    .execute()
```

Batch support in DML clauses

Querydsl SQL supports usage of JDBC batch updates through the DML APIs. If you have consecutive DML calls with a similar structure, you can bundle the the calls via addBatch() usage into one DMLClause. See the examples how it works for UPDATE, DELETE and INSERT.

```
    private static final QSurvey survey = QSurvey.survey;

    @Test
    public void updateExample() throws SQLException{
        insert(survey).values(2, "A").execute();
        insert(survey).values(3, "B").execute();

        SQLUpdateClause update = update(survey);
        update.set(survey.name, "AA").where(survey.name.eq("A")).addBatch();
        update.set(survey.name, "BB").where(survey.name.eq("B")).addBatch();
        assertEquals(2, update.execute());
    }

    @Test
    public void deleteExample() throws SQLException{
        insert(survey).values(2, "A").execute();
        insert(survey).values(3, "B").execute();
```

```
        SQLDeleteClause delete = delete(survey);
        delete.where(survey.name.eq("A")).addBatch();
        delete.where(survey.name.eq("B")).addBatch();
        assertEquals(2, delete.execute());
    }

    @Test
    public void insertExample(){
        SQLInsertClause insert = insert(survey);
        insert.set(survey.id, 5).set(survey.name, "5").addBatch();
        insert.set(survey.id, 6).set(survey.name, "6").addBatch();
        assertEquals(2, insert.execute());
    }
```

Bean class generation

To create JavaBean DTO types for the tables of your schema use the MetaDataExporter like this :

```
java.sql.Connection conn = ...;
MetaDataExporter exporter = new MetaDataExporter();
exporter.setPackageName("com.myproject.mydomain");
exporter.setTargetFolder(new File("src/main/java")) ;
exporter.setBeanSerializer(new BeanSerializer());
exporter.export(conn.getMetaData());
```

Now you can use the bean types as arguments to the populate method in DML clauses and you can project directly to bean types in queries. Here is a simple example in JUnit form :

```
    @Test
    public void Insert_Update_Query_and_Delete(){
        QEmployee e = new QEmployee("e");

        // Insert
        Employee employee = new Employee();
        employee.setFirstname("John");
        Integer id = insert(e).populate(employee).executeWithKey(e.id);
        employee.setId(id);

        // Update
        employee.setLastname("Smith");
        assertEquals(1l, update(e).populate(employee).where(e.id.eq(employee.getId())).execute());

        // Query
        Employee smith = query().from(e).where(e.lastname.eq("Smith")).uniqueResult(e);
        assertEquals("John", smith.getFirstname());

        // Delete
        assertEquals(1l, delete(e).where(e.id.eq(employee.getId())).execute());
    }
```

The factory methods used in the previous example are here :

```
```

```
    protected SQLUpdateClause update(RelationalPath<?> e){
        return new SQLUpdateClause(Connections.getConnection(), templates, e);
    }

    protected SQLInsertClause insert(RelationalPath<?> e){
        return new SQLInsertClause(Connections.getConnection(), templates, e);
    }

    protected SQLDeleteClause delete(RelationalPath<?> e){
        return new SQLDeleteClause(Connections.getConnection(), templates, e);
    }

    protected SQLMergeClause merge(RelationalPath<?> e){
        return new SQLMergeClause(Connections.getConnection(), templates, e);
    }

    protected SQLQuery query() {
        return new SQLQueryImpl(Connections.getConnection(), templates);
   }
```

Custom syntax expressions

If you need to specify SQL function calls in Querydsl you can use TemplateExpressions to express them. For general expressions you can use the SimpleTemplate class and for typed expressions BooleanTemplate, ComparableTemplate, DateTemplate, DateTimeTemplate, EnumTemplate, NumberTemplate, StringTemplate and TimeTemplate.

Here is an example for SimpleTemplate usage :

```
Expression<?> arg1 = ...;
Expression<?> arg2 = ...;
Expression<String> expression = SimpleTemplate.create(String.class, "myfunction({0},{1})", arg1, arg2);
```

And here is an example for a Number typed template expression :

```
Expression<?> arg1 = ...;
Expression<?> arg2 = ...;
NumberExpression<Integer> expression = NumberTemplate.create(Integer.class, "myfunction({0},{1})", arg1, arg2);
```

Custom types

Querydsl SQL provides the possibility to declare custom type mappings for ResultSet/Statement interaction. The custom type mappings can be declared in com.mysema.query.sql.Configuration instances, which are supplied as constructor arguments to the actual queries :

```
    @Test
    public void ForSQLType(){
        Configuration configuration = new Configuration(new H2Templates());
```

```
        // overrides the mapping for Types.DATE
        configuration.register(new UtilDateType());
    }

    @Test
    public void ForTableColumn(){
        Configuration configuration = new Configuration(new H2Templates());
        // declares a maping for the gender column in the person table
        configuration.register("person", "gender",  new EnumByNameType<Gender>(Gender.class));
        assertEquals(Gender.class, configuration.getJavaType(java.sql.Types.VARCHAR, "person", "gender"));
    }
```

2.4. Querying Lucene

This chapter describes the querying functionality of the Lucene module.

Creating the query types

With fields year and title a manually created query type could look something like this :

```
public class QDocument extends EntityPathBase<Document>{
    private static final long serialVersionUID = -4872833626508344081L;

    public QDocument(String var) {
        super(Document.class, PathMetadataFactory.forVariable(var));
    }

    public final StringPath year = createString("year");

    public final StringPath title = createString("title");
}
```

QDocument represents a Lucene document with the fields year and title.

Querying

Querying with Querydsl Lucene is as simple as this :

```
QDocument doc = new QDocument("doc");

IndexSearcher searcher = new IndexSearher(index);
LuceneQuery query = new LuceneQuery(true, searcher);
List<Document> documents = query
    .where(doc.year.between("1800", "2000").and(doc.title.startsWith("Huckle"))
    .list();
```

which is transformed into the following Lucene query :

```
+year:[1800 TO 2000] +title:huckle*
```

General usage

Use the the cascading methods of the LuceneQuery class like this

where : Define the query filters, either in varargs form separated via commas or cascaded via the and-operator. Supported operations are operations performed on PStrings except *matches*, *indexOf*, *charAt*. Currently *in* is not supported, but will be in the future.

orderBy : Define the ordering of the result as an varargs array of order expressions. Use asc() and desc() on numeric, string and other comparable expression to access the OrderSpecifier instances.

limit, offset, restrict : Define the paging of the result. Limit for max results, offset for skipping rows and restrict for defining both in one call.

Ordering

The syntax for declaring ordering is

```
query
    .where(doc.title.like("*"))
    .orderBy(doc.title.asc(), doc.year.desc())
    .list();
```

which is equivalent to the following Lucene query

```
title:*
```

The results are sorted ascending based on title and year.

Limit

The syntax for declaring a limit is

```
query
    .where(doc.title.like("*"))
    .limit(10)
    .list();
```

Offset

The syntax for declaring an offset is

```
query
    .where(doc.title.like("*"))
    .offset(3)
```

```
    .list();
```

Fuzzy searches

Fuzzy searches can be expressed via fuzzyLike methods in the com.mysema.query.lucene.LuceneUtils class :

```
query
    .where(LuceneUtils.fuzzyLike(doc.title, "Hello"))
    .list();
```

Applying Lucene filters to queries

It is possible to apply a single Lucene filter to the query like this :

```
query
    .where(doc.title.like("*"))
    .filter(filter)
    .list();
```

A shortcut for distinct filtering is provided via the distinct(Path) method :

```
query
    .where(doc.title.like("*"))
    .distinct(doc.title)
    .list();
```

2.5. Querying Hibernate Search

This chapter describes the querying functionality of the Hibernate Search module.

Creating the Querydsl query types

See Querying JPA/Hibernate sources for instructions on how to create query types.

Querying

Querying with Querydsl Hibernate Search is as simple as this :

```
QUser user = new QUser("user");

SearchQuery<User> query = new SearchQuery<User>(session, user);

List<User> list = query
 .where(user.firstName.eq("Bob"))
 .list();
```

General usage

For general usage instructions see Querying Lucene sources.

In the query serialization the only difference to the Querydsl Lucene module is that paths are treated differently. For *org.hibernate.search.annotations.Field* annotated properties the name attribute is used with the property name as fallback for the field name.

2.6. Querying Mongodb

This chapter describes the querying functionality of the Mongodb module.

Maven integration

Add the following dependencies to your Maven project and make sure that the Maven 2 repo of Mysema Source (http://source.mysema.com/maven2/releases) is accessible from your POM if the version cannot yet be found in other public Maven repos :

```
<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-mongodb</artifactId>
  <version>${querydsl.version}</version>
</dependency>

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.6.1</version>
</dependency>
```

And now, configure the Maven APT plugin which generates the query types used by Querydsl :

```
<project>
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>com.mysema.maven</groupId>
        <artifactId>maven-apt-plugin</artifactId>
        <version>1.0</version>
        <executions>
          <execution>
            <goals>
              <goal>process</goal>
            </goals>
            <configuration>
              <outputDirectory>target/generated-sources/java</outputDirectory>
              <processor>com.mysema.query.mongodb.morphia.MorphiaAnnotationProcessor</processor>
            </configuration>
          </execution>
```

```
        </executions>
      </plugin>
    ...
    </plugins>
  </build>
</project>
```

The MorphiaAnnotationProcessor finds domain types annotated with the com.google.code.morphia.annotations.Entity annotation and generates Querydsl query types for them.

Run clean install and you will get your Query types generated into target/generated-sources/java.

If you use Eclipse, run mvn eclipse:eclipse to update your Eclipse project to include target/generated-sources/java as a source folder.

Now you are able to construct Mongodb queries and instances of the query domain model.

Querying

Querying with Querydsl Mongodb with Morphia is as simple as this :

```
Morphia morphia;
Datastore datastore;
// ...
QUser user = new QUser("user");
MorphiaQuery<User> query = new MorphiaQuery<User>(morphia, datastore, user);
List<User> list = query
 .where(user.firstName.eq("Bob"))
 .list();
```

General usage

Use the the cascading methods of the MongodbQuery class like this

where : Define the query filters, either in varargs form separated via commas or cascaded via the and-operator. Supported operations are operations performed on PStrings except *matches*, *indexOf*, *charAt*. Currently *in* is not supported, but will be in the future.

orderBy : Define the ordering of the result as an varargs array of order expressions. Use asc() and desc() on numeric, string and other comparable expression to access the OrderSpecifier instances.

limit, *offset*, *restrict :* Define the paging of the result. Limit for max results, offset for skipping rows and restrict for defining both in one call.

Ordering

The syntax for declaring ordering is

```
```

```
query
    .where(doc.title.like("*"))
    .orderBy(doc.title.asc(), doc.year.desc())
    .list();
```

The results are sorted ascending based on title and year.

Limit

The syntax for declaring a limit is

```
query
    .where(doc.title.like("*"))
    .limit(10)
    .list();
```

Offset

The syntax for declaring an offset is

```
query
    .where(doc.title.like("*"))
    .offset(3)
    .list();
```

Geospatial queries

Support for geospatial queries is available for Double typed arrays (Double[]) via the near-method :

```
query
    .where(geoEntity.location.near(50.0, 50.0))
    .list();
```

2.7. Querying Collections

The querydsl-collections module can be used with generated query types and without. The first section describes the usage without generated query types :

Usage without generated query types

To use querydsl-collections without generated query types you need to use the Querydsl alias feature. Here are some examples.

To get started, add the following static imports :

```
import static com.mysema.query.alias.Alias.*;
```

And now create an alias instance for the Cat class. Alias instances can only be created for classes with an empty constructor. Make sure your class has one.

The alias instance of type Cat and it's getter invocations are transformed into Querydsl paths by wrapping them into dollar method invocations. The call *c.getKittens()* for example is internally transformed into the property path c.kittens inside the dollar method.

```
Cat c = alias(Cat.class, "cat");
for (String name : from($(c),cats)
  .where($(c.getKittens()).size().gt(0))
  .list($(c.getName()))){
    System.out.println(name);
}
```

The following example is a variation of the previous, where the access to the list size happens inside the dollar-method invocation.

```
Cat c = alias(Cat.class, "cat");
for (String name : from($(c),cats)
  .where($(c.getKittens().size()).gt(0))
  .list($(c.getName()))){
    System.out.println(name);
}
```

All non-primitive and non-String typed properties of aliases are aliases themselves. So you may cascade method calls until you hit a primitive or String type in the dollar-method scope.

e.g.

```
$(c.getMate().getName())
```

is transformed into *c.mate.name* internally, but

```
$(c.getMate().getName().toLowerCase())
```

is not transformed properly, since the toLowerCase() invocation is not tracked.

Note also that you may only invoke getters, size(), contains(Object) and get(int) on alias types. All other invocations throw exceptions.

Usage with generated query types

The example above can be expressed like this with generated query types

```
QCat cat = new QCat("cat");
```

```
for (String name : from(cat,cats)
  .where(cat.kittens.size().gt(0))
  .list(cat.name)){
    System.out.println(name);
}
```

When you use generated query types, you instantiate query types instead of alias instances and use the property paths directly without any dollar-method wrapping.

Maven integration

If you are not using JPA or JDO you can generate Querydsl query types for your domain types by annotating them with the com.mysema.query.annotations.QueryEntity annotation and adding the following plugin configuration into your Maven configuration (pom.xml) :

```
<project>
  <build>
  <plugins>
    ...
    <plugin>
      <groupId>com.mysema.maven</groupId>
      <artifactId>maven-apt-plugin</artifactId>
      <version>1.0</version>
      <executions>
        <execution>
          <goals>
            <goal>process</goal>
          </goals>
          <configuration>
            <outputDirectory>target/generated-sources/java</outputDirectory>
            <processor>com.mysema.query.apt.QuerydslAnnotationProcessor</processor>
          </configuration>
        </execution>
      </executions>
    </plugin>
    ...
  </plugins>
  </build>
</project>
```

Ant integration

Place the jar files from the full-deps bundle on your classpath and use the following tasks for Querydsl code generation :

```
    <!-- APT based code generation -->
    <javac srcdir="${src}" classpathref="cp">
      <compilerarg value="-proc:only"/>
      <compilerarg value="-processor"/>
      <compilerarg value="com.mysema.query.apt.QuerydslAnnotationProcessor"/>
      <compilerarg value="-s"/>
```

```
      <compilerarg value="${generated}"/>
    </javac>

    <!-- compilation -->
    <javac classpathref="cp" destdir="${build}">
      <src path="${src}"/>
      <src path="${generated}"/>
    </javac>
```

Replace *src* with your main source folder, *generated* with your folder for generated sources and *build* with your target folder.

2.8. Querying in Scala

Generic support for Querydsl usage in Scala is available via querydsl-scala module. To add it to your Maven build, use the following snippet :

```
<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-scala</artifactId>
  <version>${querydsl.version}</version>
</dependency>
```

DSL expressions for Scala

Querydsl for Scala provides an alternative DSL for expression construction. The Scala DSL utilizes language features such as operator overloading, function pointers and implicit imports for enhanced readability and conciseness.

Here is an overview of the main alternatives :

```
//Standard              Alternative

expr isNotNull          expr is not(null)
expr isNull             expr is null
expr eq "Ben"           expr === "Ben"
expr ne "Ben"           expr !== "Ben"
expr append "X"         expr + "X"
expr isEmpty            expr is empty
expr isNotEmpoty        expr not empty

// boolean
left and right          left && right
left or right           left || right
expr not                !expr

// comparison
expr lt 5               expr < 5
expr loe 5              expr <= 5
expr gt 5               expr > 5
```

```
expr goe 5              expr >= 5
expr notBetween(2,6)    expr not between (2,6)
expr negate             -expr

// numeric
expr add 3              expr + 3
expr subtract 3       expr - 3
expr divide 3           expr / 3
expr multiply 3         expr * 3
expr mod 5              expr % 5

// collection
list.get(0)             list(0)
map.get("X")            map("X")
```

Improved projections

The Querydsl Scala module offers a few implicit conversion to make Querydsl query projections more Scala compatible.

The RichProjectable and RichSimpleProjectable wrappers should be used to enable Scala projections for Querydsl queries. By importing the contents of com.mysema.query.scala.Helpers the needed implicit conversions become available.

For example the following query with the standard API would return a java.util.List of type Object[].

```
query.from(person).list(person.firstName, person.lastName, person.age)
```

With the added conversions you can use select instead of list for Scala list typed results, unique instead of uniqueResult for Option typed results and single instead of singleResult for Option typed results.

The previous query could be expressed like this with the implicit conversions:

```
import com.mysema.query.scala.Helpers._

query.from(person).select(person.firstName, person.lastName, person.age)
```

In this case the result type would be List[(String,String,Integer)] or in other words List of Tuple3[String,String,Integer].

Querying with SQL

Like with Querydsl SQL for Java you need to generate Query types to be able to construct your queries. The following code examples show how this is done :

Generation without Bean types :

```
```

```
val directory = new java.io.File("target/jdbcgen1")
val namingStrategy = new DefaultNamingStrategy()
val exporter = new MetaDataExporter()
exporter.setNamePrefix("Q")
exporter.setPackageName("com.mysema")
exporter.setSchemaPattern("PUBLIC")
exporter.setTargetFolder(directory)
exporter.setSerializerClass(classOf[ScalaMetaDataSerializer])
exporter.setCreateScalaSources(true)
exporter.setTypeMappings(ScalaTypeMappings.create)
exporter.export(connection.getMetaData)
```

Generation with Bean types :

```

val directory = new java.io.File("target/jdbcgen2")
val namingStrategy = new DefaultNamingStrategy()
val exporter = new MetaDataExporter()
exporter.setNamePrefix("Q")
exporter.setPackageName("com.mysema")
exporter.setSchemaPattern("PUBLIC")
exporter.setTargetFolder(directory)
exporter.setSerializerClass(classOf[ScalaMetaDataSerializer])
exporter.setBeanSerializerClass(classOf[ScalaBeanSerializer])
exporter.setCreateScalaSources(true)
exporter.setTypeMappings(ScalaTypeMappings.create)
exporter.export(connection.getMetaData)
```

Compact queries

Querydsl Scala provides a compact query syntax for Querydsl SQL. The syntax is inspired by domain oriented query syntaxes like that from the Rogue framework.

The domain oriented queries are implemented as implicit conversions from RelationalPath instances into queries. This functionality can be made available by implementing the com.mysema.query.scala.sql.SQLHelpers trait in your service or DAO classes.

Using this compact syntax you can use your meta model classes as a starting point for queries.

Instead of the following normal syntax

```

query().from(employee).select(employee.firstName, employee.lastName)
```

you could use the companion object of Employee or QEmployee and write it like this

```

Employee.select(_.firstName, _.lastName)
```

Instead of giving expressions to orderBy, where, select, single and unique you can give functions which take the root expression of the query and return another expression. The expanded form of the previous example would be

```
Employee.select({ e => e.firstName }, { e => e.lastName })
```

See the signature of the com.mysema.query.scala.sql.RichSimpleQuery class for details.

Code generation

Scala sources for SQL metatypes and projections can be generated with querydsl-maven-plugin. Here is an example configuration

```
<plugin>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-maven-plugin</artifactId>
  <version>${querydsl.version}</version>
  <configuration>
    <jdbcDriver>com.mysql.jdbc.Driver</jdbcDriver>
    <jdbcUrl>jdbc:mysql://localhost:3306/test</jdbcUrl>
    <jdbcUser>matko</jdbcUser>
    <jdbcPassword>matko</jdbcPassword>
    <packageName>com.example.schema</packageName>
    <targetFolder>${project.basedir}/src/main/scala</targetFolder>
    <exportBeans>true</exportBeans>
    <createScalaSources>true</createScalaSources>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
      <version>5.1.16</version>
 </dependency>
   <dependency>
     <groupId>com.mysema.querydsl</groupId>
     <artifactId>querydsl-scala</artifactId>
     <version>${querydsl.version}</version>
     </dependency>
     <dependency>
     <groupId>org.scala-lang</groupId>
     <artifactId>scala-library</artifactId>
     <version>${scala.version}</version>
     </dependency>
  </dependencies>
</plugin>
```

The maven goal to execute is querydsl:export.

Querying with other backends

When querying with other backends the Expression model has to be created manually or alternatively the alias functionality can be used.

Here is a minimal example with JPA/Hibernate :

```
@Entity
class User {
  @BeanProperty
  @Id
  var id: Integer = _;
  @BeanProperty
  var userName: String = _;
  @BeanProperty
  @ManyToOne
  var department: Department = _;
}

@Entity
class Department {
  @BeanProperty
  @Id
  var id: Integer = _;
  @BeanProperty
  var name: String = _;
}
```

And here are some query examples

```
import com.mysema.query.scala.Conversions._
import com.mysema.query.jpa.impl.JPAQuery

import com.mysema.query.types.path._
import org.junit.Test

class JPAQueryTest {

  val person = alias(classOf[Person])

  @Test
  def Various() {
    // list
    query from person where (person.firstName $like "Rob%")
      .list person

    // unique result
    query from person where (person.firstName $like "Rob%")
      .unique person

    // long where
    query from person
      .where (person.firstName $like "Rob%", person.lastName $like "An%")
      .list person

    // order
    query from person orderBy (person.firstName asc) list person

    // not null
    query from person
      .where (person.firstName $isEmpty, person.lastName $isNotNull)
      .list person
```

```
  }

  def query() = new JPAQuery(entityManager)

}
```

The main import for Querydsl Scala integration is the following

```
import com.mysema.query.scala.Conversions._
```

The factory method for query creation is

```
def query() = new JPAQuery(entityManager)
```

In addition to queries you need variables which can be created like this

```
var person = alias(classOf[Person])
```

The person variable is a proxy instance of the the Person class which can be used in queries. Now you can construct your queries, populate them via from-where-...-orderBy calls and get the projection out via list/uniqueResult/listResults calls.

Querydsl expressions are constructed via method calls starting with the "$" sign.

With the Querydsl Java API a simple like expression would be constructed like this :

```
person.firstName.like("Rob%")
```

Using the Scala API it is

```
person.firstName $like "Rob%"
```

3. General usage

3.1. Expressions

Custom expressions

The com.mysema.query.support.Expressions class is a static factory class for Querydsl expression construction.

The following expression

```
QPerson person = QPerson.person;
person.firstName.startsWith("P");
```

could be constructed like this if Q-types wouldn't be available

```
Path<Person> person = Expressions.path(Person.class, "person");
Path<String> personFirstName = Expressions.path(String.class, person, "firstName");
Constant<String> constant = Expressions.constant("P");
Expressions.predicate(Ops.STARTS_WITH, personFirstName, constant);
```

Path instances represent variables and properties, Constants are constants, Operations are operations and TemplateExpression instances can be used to express expressions as String templates.

Custom projections

For custom projections the com.mysema.query.types.Projections class offers a set of factory methods for Bean, constructor, Tuple and other projections.

Inheritance in Querydsl types

To avoid a generic signature in Querydsl query types the type hierarchies are flattened. The result is that all generated query types are direct subclasses of com.mysema.query.types.path.EntityPathBase or com.mysema.query.types.path.BeanPath and cannot be directly cast to their Querydsl supertypes.

Instead of a direct Java cast, the supertype reference is accessible via the _super field. A _super-field is available in all generated query types with a single supertype :

```
// from Account
QAccount extends EntityPathBase<Account>{
    // ...
}

// from BankAccount extends Account
QBankAccount extends EntityPathBase<BankAccount>{
```

```
    public final QAccount _super = new QAccount(this);

    // ...
}
```

To cast from a supertype to a subtype you can use the as-method of the EntityPathBase class :

```
QAccount account = new QAccount("account");
QBankAccount bankAccount = account.as(QBankAccount.class);
```

Constructor projections

Querydsl provides the possibility to use constructor invocations in projections. To use a constructor in a query projection, you need to annotate it with the QueryProjection annotation :

```
class CustomerDTO {

  @QueryProjection
  public CustomerDTO(long id, String name){
     ...
  }

}
```

And then you can use it like this in the query

```
QCustomer customer = QCustomer.customer;
JPQLQuery query = new HibernateQuery(session);
List<CustomerDTO> dtos = qry.from(customer).list(new QCustomerDTO(customer.id, customer.name));
```

While the example is Hibernate specific, this feature is available in all modules.

If the type with the QueryProjection annotation is not an annotated entity type, you can use the constructor projection like in the example, but if the annotated type would be an entity type, then the constructor projection would need to be created via a call to the static create method of the query type :

```
@Entity
class Customer {

  @QueryProjection
  public Customer(long id, String name){
     ...
  }

}
```

```
QCustomer customer = QCustomer.customer;
JPQLQuery query = new HibernateQuery(session);
List<Customer> dtos = qry.from(customer).list(QCustomer.create(customer.id, customer.name));
```

Alternatively, if code generation is not an option, you can create a constructor projection like this :

```
List<Customer> dtos = qry.from(customer)
    .list(ConstructorExpression.create(Customer.class, customer.id, customer.name));
```

Complex boolean expressions

To construct complex boolean expressions, use the BooleanBuilder class. It implements Predicate and can be used in cascaded form :

```
public List<Customer> getCustomer(String... names){
    QCustomer customer = QCustomer.customer;
    HibernateQuery qry = new HibernateQuery(session).from(customer);
    BooleanBuilder builder = new BoolenBuilder();
    for (String name : names){
        builder.or(customer.name.eq(name));
    }
    qry.where(builder); // customer.name eq name1 OR customer.name eq name2 OR ...
    return qry.list(customer);
}
```

Case expressions

To construct case-when-then-else expressions use the CaseBuilder class like this :

```
QCustomer customer = QCustomer.customer;
Expression<String> cases = new CaseBuilder()
    .when(customer.annualSpending.gt(10000)).then("Premier")
    .when(customer.annualSpending.gt(5000)).then("Gold")
    .when(customer.annualSpending.gt(2000)).then("Silver")
    .otherwise("Bronze");
// The cases expression can now be used in a projection or condition
```

For case expressions with equals-operations use the following simpler form instead :

```
QCustomer customer = QCustomer.customer;
Expression<String> cases = customer.annualSpending
    .when(10000).then("Premier")
    .when(5000).then("Gold")
    .when(2000).then("Silver")
    .otherwise("Bronze");
// The cases expression can now be used in a projection or condition
```

Case expressions are not yet supported in JDOQL.

Dynamic path usage

For dynamic path generation the PathBuilder class can be used. It extends EntityPathBase and can be used as an alternative to class generation and alias-usage for path generation.

String property :

```
PathBuilder<User> entityPath = new PathBuilder<User>(User.class, "entity");
// fully generic access
entityPath.get("userName");
// .. or with supplied type
entityPath.get("userName", String.class);
// .. and correct signature
entityPath.getString("userName").lower();
```

List property with component type :

```
entityPath.getList("list", String.class).get(0);
```

Using a component expression type :

```
entityPath.getList("list", String.class, StringPath.class).get(0).lower();
```

Map property with key and value type :

```
entityPath.getMap("map", String.class, String.class).get("key");
```

Using a component expression type :

```
entityPath.getMap("map", String.class, String.class, StringPath.class).get("key").lower();
```

3.2. Configuration

Path initialization

By default Querydsl initializes only direct reference properties. In cases where longer initialization paths are required, these have to be annotated in the domain types via com.mysema.query.annotations.QueryInit usage. QueryInit is used on properties where deep initializations are needed. The following example demonstrates the usage.

```
@Entity
class Event {
    @QueryInit("customer")
    Account account;
```

```
}

@Entity
class Account{
    Customer customer;
}

@Entity
class Customer{
    String name;
    // ...
}
```

This example enforces the initialization of the account.customer path, when an Event path is initialized as a root path / variable. The path initialization format supports wildcards as well, e.g. "customer.*" or just "*".

The declarative path initialization replaces the manual one, which required the entity fields to be non-final. The declarative format has the benefit to be applied to all top level instances of a Query type and to enable the usage of final entity fields.

Declarative path initialization is the preferred initialization strategy, but manual initialization can be activated via the Config annotation, which is described below.

Customization of serialization

The serialization of Querydsl can be customized via Config annotations on packages and types. They customize the serialization of the annotated package or type.

The serialization options are *entityAccessors* to generate accessor methods for entity paths instead of public final fields (default : false), *listAccessors* to generate listProperty(int index) style methods (default : false), *mapAccessors* to generate mapProperty(Key key) style accessor methods (default : false) and *createDefaultVariable* to generate the default variable (default : true).

Below are some examples.

Customization of Entity type serialization :

```

@Config(entityAccessors=true)
@Entity
public class User {
 //...
}
```

Customization of package content :

```

@Config(listAccessors=true)
package com.mysema.query.domain.rel;

import com.mysema.query.annotations.Config;
```

If you want to customize the serializer configuration globally, you can do this via the APT options *querydsl.entityAccessors* to enable reference field accessors, *querydsl.listAccessors* to enable accessors for direct indexed list access, *querydsl.mapAccessors* to enable accessors for direct key based map access, *querydsl.prefix* to override the prefix for query types(default: Q), *querydsl.suffix* to set a suffix for query types, *querydsl.packageSuffix* to set a suffix for query type packages, *querydsl.createDefaultVariable* to set whether default variables are created.

Using the Maven APT plugin this works for example like this :

```
<project>
  <build>
  <plugins>
    ...
    <plugin>
      <groupId>com.mysema.maven</groupId>
      <artifactId>maven-apt-plugin</artifactId>
      <version>1.0</version>
      <executions>
        <execution>
          <goals>
            <goal>process</goal>
          </goals>
          <configuration>
            <outputDirectory>target/generated-sources/java</outputDirectory>
            <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>
            <options>
              <querydsl.entityAccessors>true</querydsl.entityAccessors>
            </options>
          </configuration>
        </execution>
      </executions>
    </plugin>
    ...
  </plugins>
  </build>
</project>
```

Custom type mappings

Custom type mappings can be used on properties to override the derived Path type. This can be useful for example in cases where comparison and String operations should be blocked on certain String paths or Date / Time support for custom types needs to be added. Support for Date / Time types of the Joda time API and JDK (java.util.Date, Calendar and subtypes) is built in, but other APIs might need to be supported using this feature.

The following example demonstrates the usage :

```
@Entity
public class MyEntity{
    @QueryType(PropertyType.SIMPLE)
```

```
    public String stringAsSimple;

    @QueryType(PropertyType.COMPARABLE)
    public String stringAsComparable;

    @QueryType(PropertyType.NONE)
    public String stringNotInQuerydsl;
}
```

The value PropertyType.NONE can be used to skip a property in the Querydsl query type generation. This case is different from @Transient or @QueryTransient annotated properties, where properties are not persisted. PropertyType.NONE just omits the property from the Querydsl query type.

Delegate methods

To declare a static method as a delegate method add the QueryDelegate annotation with the corresponding domain type as a value and provide a method signature that takes the corresponding Querydsl query type as the first argument.

Here is a simple example from a unit test:

```

    @QueryEntity
    public static class User{

        String name;

        User manager;

    }

    @QueryDelegate(User.class)
    public static BooleanPath isManagedBy(QUser user, User other){
        return user.manager.eq(other);
    }
```

And the generated methods in the QUser query type :

```

    public BooleanPath isManagedBy(QUser other) {
        return com.mysema.query.domain.DelegateTest.isManagedBy(this, other);
    }
```

Delegate methods can also be used to extend built-in types. Here are some examples

```

public class QueryExtensions {

    @QueryDelegate(Date.class)
    public static BooleanExpression inPeriod(DatePath<Date> date, Pair<Date,Date> period){
```

```
        return date.goe(period.getFirst()).and(date.loe(period.getSecond()));
    }

    @QueryDelegate(Timestamp.class)
    public static BooleanExpression inDatePeriod(DateTimePath<Timestamp> timestamp, Pair<Date,Date> period){
        Timestamp first = new Timestamp(DateUtils.truncate(period.getFirst(), Calendar.DAY_OF_MONTH).getTime())
        Calendar second = Calendar.getInstance();
        second.setTime(DateUtils.truncate(period.getSecond(), Calendar.DAY_OF_MONTH));
        second.add(1, Calendar.DAY_OF_MONTH);
        return timestamp.goe(first).and(timestamp.lt(new Timestamp(second.getTimeInMillis())));
    }

}
```

When delegate methods are declared for builtin types then subclasses with the proper delegate method usages are created :

```
public class QDate extends DatePath<java.sql.Date> {

    public QDate(BeanPath<? extends java.sql.Date> entity) {
        super(entity.getType(), entity.getMetadata());
    }

    public QDate(PathMetadata<?> metadata) {
        super(java.sql.Date.class, metadata);
    }

    public BooleanExpression inPeriod(com.mysema.commons.lang.Pair<java.sql.Date, java.sql.Date> period) {
        return QueryExtensions.inPeriod(this, period);
    }

}

public class QTimestamp extends DateTimePath<java.sql.Timestamp> {

    public QTimestamp(BeanPath<? extends java.sql.Timestamp> entity) {
        super(entity.getType(), entity.getMetadata());
    }

    public QTimestamp(PathMetadata<?> metadata) {
        super(java.sql.Timestamp.class, metadata);
    }

    public BooleanExpression inDatePeriod(com.mysema.commons.lang.Pair<java.sql.Date, java.sql.Date> period) {
        return QueryExtensions.inDatePeriod(this, period);
    }

}
```

Query type generation for not annotated types

It is possible to create Querydsl query types for not annotated types by creating @QueryEntities annotations. Just place a QueryEntities annotation into a package of your choice and the classes to mirrored in the value attribute.

To actually create the types use the com.mysema.query.apt.QuerydslAnnotationProcessor. In Maven you do it like this :

```
<project>
  <build>
  <plugins>
    ...
    <plugin>
      <groupId>com.mysema.maven</groupId>
      <artifactId>maven-apt-plugin</artifactId>
      <version>1.0</version>
      <executions>
        <execution>
          <goals>
            <goal>process</goal>
          </goals>
          <configuration>
            <outputDirectory>target/generated-sources/java</outputDirectory>
            <processor>com.mysema.query.apt.QuerydslAnnotationProcessor</processor>
          </configuration>
        </execution>
      </executions>
    </plugin>
    ...
  </plugins>
  </build>
</project>
```

3.3. Best practices

Use default variable of the Query types

Use the default variables of the query types as much as possible. The default variables are available as static final fields in the query types. The name is always the decapitalized version of the simple type name. For the type Account this would be account :

```
public class QAccount extends EntityPathBase<Account>{

    public static final QAccount account = new QAccount("account");

}
```

Querydsl query types are safe to re-use, and by using Querydsl default variables you save initialization time and memory.

Interface based usage

Whenever possible, use interface based query references : e.g. JDOQLQuery for JDO and HQLQuery for HQL

Custom query extensions

TODO

DAO integration

A practice which we have found to be very easy to use is to provide factory methods for Query instances in DAO implementations in the following form.

For JPA usage :

```
protected JPAQuery from(EntityPath<?>... o) {
    return new JPAQuery(entityManager).from(o);
}
```

For JDO usage :

```
protected JDOQLQuery from(EntityPath<?>... o) {
    return new JDOQLQueryImpl(persistenceManager).from(o);
}
```

3.4. Alias usage

In cases where code generation is not an option, alias objects can be used as path references for expression construction.

The following examples demonstrate how alias objects can be used as replacements for expressions based on generated types.

At first an example query with APT generated domain types :

```
QCat cat = new QCat("cat");
for (String name : from(cat,cats)
  .where(cat.kittens.size().gt(0))
  .iterate(cat.name)){
    System.out.println(name);
}
```

And now with an alias instance for the Cat class. The call "c.getKittens()" inside the dollar-method is internally transformed into the property path c.kittens.

```
Cat c = alias(Cat.class, "cat");
for (String name : from($(c),cats)
  .where($(c.getKittens()).size().gt(0))
  .iterate($(c.getName()))){
```

```
    System.out.println(name);
}
```

To use the alias functionality in your code, add the following two imports

```
import static com.mysema.query.alias.Alias.$;
import static com.mysema.query.alias.Alias.alias;
```

The following example is a variation of the previous, where the access to the list size happens inside the dollar-method invocation.

```
Cat c = alias(Cat.class, "cat");
for (String name : from($(c),cats)
  .where($(c.getKittens().size()).gt(0))
  .iterate($(c.getName()))){
    System.out.println(name);
}
```

All non-primitive and non-String typed properties of aliases are aliases themselves. So you may cascade method calls until you hit a primitive or String type in the dollar-method scope. e.g.

```
$(c.getMate().getName())
```

is transformed into *c.mate.name* internally, but

```
$(c.getMate().getName().toLowerCase())
```

is not transformed properly, since the toLowerCase() invocation is not tracked.

Note also that you may only invoke getters, size(), contains(Object) and get(int) on alias types. All other invocations throw exceptions.

4. Troubleshooting

4.1. Insufficient type arguments

Querydsl needs properly encoded List Set, Collection and Map properties in all code generation scenarios.

When using improperly encoded fields or getters you might the following stacktrace :

```
java.lang.RuntimeException: Caught exception for field com.mysema.query.jdoql.testdomain.Store#products
  at com.mysema.query.apt.Processor$2.visitType(Processor.java:117)
  at com.mysema.query.apt.Processor$2.visitType(Processor.java:80)
  at com.sun.tools.javac.code.Symbol$ClassSymbol.accept(Symbol.java:827)
  at com.mysema.query.apt.Processor.getClassModel(Processor.java:154)
  at com.mysema.query.apt.Processor.process(Processor.java:191)
  ...
Caused by: java.lang.IllegalArgumentException: Insufficient type arguments for List
  at com.mysema.query.apt.APTTypeModel.visitDeclared(APTTypeModel.java:112)
  at com.mysema.query.apt.APTTypeModel.visitDeclared(APTTypeModel.java:40)
  at com.sun.tools.javac.code.Type$ClassType.accept(Type.java:696)
  at com.mysema.query.apt.APTTypeModel.<init>(APTTypeModel.java:55)
  at com.mysema.query.apt.APTTypeModel.get(APTTypeModel.java:48)
  at com.mysema.query.apt.Processor$2.visitType(Processor.java:114)
  ... 35 more
```

Examples of problematic field declarations and their corrections :

```
    private Collection names; // WRONG

    private Collection<String> names; // RIGHT

    private Map employeesByName; // WRONG

    private Map<String,Employee> employeesByName; // RIGHT
```

4.2. JDK5 usage

When compiling your project with JDK 5, you might get the following compilation failure:

```
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Compilation failure
...
class file has wrong version 50.0, should be 49.0
```

The class file version 50.0 is used by Java 6.0, and 49.0 is used by Java 5.0.

Querydsl is tested against JDK 6.0 only, as we use APT extensively, which is available only since JDK 6.0.

If you want to use it with JDK 5.0 you might want to try to compile Querydsl yourself.