Querydsl

Reference Documentation

Timo Westkämper
Samppa Saarela

Querydsl: Reference Documentation

by Timo Westkämper and Samppa Saarela

1.3.0

Legal Notice

Table of Contents

Preface

Querydsl (spell: query diesel) 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 fluentDSL/API like Querydsl.

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

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 Collections, JDO, JDBC 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.expr.Expr.

2. Getting started with Querydsl

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

2.1. Querying JDO sources

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. Support for JDO is in beta phase and still to be considered experimental.

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>0.5.4</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-jdoql</artifactId>
  <version>0.5.4</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>0.2.0</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.

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")));
```

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 EBoolean 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 :

```
QCat cat = QCat.cat;
// delete all cats
new JDOQLDeleteClause(pm, cat).execute();
// delete all cats with kittens
```

```
new JDOQLDeleteClause(pm, cat).where(cat.kittens.isNotEmpty()).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)
```

2.2. Querying JPA/Hibernate sources

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 HQL and Criteria queries. It combines the dynamic nature of Criteria queries with the expressiveness of HQL 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>0.5.4</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-hql</artifactId>
  <version>0.5.4</version>
</dependency>
```

And now, configure the Maven APT plugin :

```
<project>
  <build>
  <plugins>
    ...
    <plugin>
      <groupId>com.mysema.maven</groupId>
      <artifactId>maven-apt-plugin</artifactId>
      <version>0.3.0</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 JPAQL/HQL query instances and instances of the query domain model.

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 with HQL

For the HQL-module HibernateQuery is the main Query implementation. It is instantiated like this :

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

To use the JPA API instead of the Hibernate API, you can instantiate a JPAQuery like this :

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

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

```
QCustomer customer = QCustomer.customer;
HQLQuery 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 just use the HQLQuery 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")));
```

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

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

Using joins

Querydsl supports the following join variants in HQL/JPAQL : 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 HQL 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)
    .on(kitten.bodyWeight.gt(10.0))
    .list(cat);
```

With the following HQL 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 HQLQuery 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 EBoolean 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 HQL

```
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 HQL

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

Delete clauses

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

```
QCat cat = QCat.cat;
// delete all cats
new HibernateDeleteClause(session, cat).execute();
// delete all cats with kittens
new HibernateDeleteClause(session, cat).where(cat.kittens.isNotEmpty()).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 HQL follow a simple update-set/where-execute form. Here are some examples :

```
QCat cat = QCat.cat;
// rename cats named Bob to Bobby
new HibernateUpdateClause(session, cat).where(cat.name.eq("Bob"))
    .set(cat.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 HQLSubQuery 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 HQLSubQuery().from(d).unique(d.employees.size().max())
     )).list(department);
```

Another example

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

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>${project.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>export</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <jdbcDriver>org.apache.derby.jdbc.EmbeddedDriver</jdbcDriver>
    <jdbcUrl>jdbc:derby:target/demoDB;create=true</jdbcUrl>
    <packageName>com.mycompany.mydomain</packageName>
    <targetFolder>target/generated-sources/java</targetFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>${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(EConstructor.create(CatDTO.class, cat.id, cat.name));
```

2.3. 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.collections.MiniApi.*;
import static com.mysema.query.alias.Alias.*; // for alias usage
```

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>0.3.0</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>
```

2.4. Querying SQL/JDBC sources

The querydsl-sql module is still in Beta stage and hasn't been tested much, but it is functional and usable.

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

Creating the Querydsl query types

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

```
java.sql.Connection conn; // connection of database containing the schema to use
// obtain Connection etc.

MetaDataExporter exporter = new MetaDataExporter(
  "Q",                              // namePrefix
  "com.myproject.domain",           // target package
  null,                             // schema name pattern
  null,                             // table name pattern
  "target/generated-sources/java"); // target source folder
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.

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>${querydsl.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>export</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <jdbcDriver>org.apache.derby.jdbc.EmbeddedDriver</jdbcDriver>
    <jdbcUrl>jdbc:derby:target/demoDB;create=true</jdbcUrl>
    <!--
      optional elements :
      * namePrefix
      * jdbcUser
      * jdbcPassword
      * schemaPattern
      * tableNamePattern
    -->
    <packageName>com.myproject.domain</packageName>
    <targetFolder>target/generated-sources/java</targetFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>${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.

Querying

Querying with Querydsl SQL is as simple as this :

```
```

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

SQLTemplates dialect = new HSQLDBTemplates(); // SQL-dialect
SQLQuery query = 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'
```

Internally Querydsl SQL uses PreparedStatements, though.

Querydsl uses SQL dialects to customize the SQL serialization needed for different relational databases. The available dialects are *DerbyTemplates*, *HSQLDBTemplates*, *MySQLTemplates* and *OracleTemplates*.

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 EBoolean 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.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 Data manipulation commands

TODO

3. General usage

3.1. 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 PEntity<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 HQL usage :

```
protected HQLQuery from(PEntity<?>... o) {
    return new HqlQueryImpl(session).from(o);
}
```

For JDO usage :

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

3.2. Special expressions

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;
HQLQuery 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 present 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;
HQLQuery query = new HibernateQuery(session);
List<Customer> dtos = qry.from(customer).list(new 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(EConstructor.create(Customer.class, customer.id, customer.name));
```

Complex boolean expressions

To construct complex boolean expressions, use the BooleanBuilder class. It extends EBoolean 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;
Expr<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;
Expr<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.

3.3. Customizations

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 QuerydslConfig annotation, which is described below.

Customization of serialization

The serialization of Querydsl can be customized via QuerydslConfig 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, *listAccessors* to generate listProperty(int index) style methods and *mapAccessors* to generate mapProperty(Key key) style accessor methods.

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.

Custom methods

Querydsl provides the possibility to annotate methods for mirroring in query types. Methods can either be annotated directly in the context of the class where they belong or in query extension interfaces, if the target class is only available for annotation.

Example 1

```
public class Point{
  // ...
}

@QueryExtensions(Point.class)
public interface PointOperations {

   @QueryMethod("geo_distance({0}, {1})")
   int geoDistance(Point otherPoint);

}
```

The first example describes indirect annotation via QueryExtensions usage. Let's assume that Point is a class of an external library which has to be used as such without the possibility of customization in source form.

To make a geoDistance(Point) method available in the Querydsl query type for Point, a query extension interface is used. Via the QueryExtensions annotation the interface is bound to the Point class and via the QueryMethod annotation the geoDistance method is declared to be mirrored into the Point query type with a serialization pattern of "geo_distance({0}, {1})".

The serialization patterns of query methods have the host object itself always as the first argument and the method parameters as further arguments.

Example 2

```
```

```
public class Point{

    @QueryMethod("geo_distance({0}, {1})")
    int geoDistance(Point otherPoint){
       // dummy implementation
        return 0;
    }

}
```

The second example features the same use case as in the first example, but this time the Point class is annotated directly. This approach is feasible, if the related domain type is available for annotation and APT post processing.

3.4. 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.PEntity 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 query types with a single supertype :

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

// from BankAccount extends Account
QBankAccount extends PEntity<BankAccount>{

    public final QAccount _super = new QAccount(this);

    // ...
}
```

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

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

3.5. 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.

3.6. Dynamic path usage

For dynamic path generation the PathBuilder class can be used. It extends PEntity 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 :

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

Map property :

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

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.

To use Querydsl with JDK 5.0 you also need to exclude querydsl-hql-apt and include querydsl-apt-jdk5 like this :

```
<dependency>
 <groupId>com.mysema.querydsl</groupId>
 <artifactId>querydsl-hql</artifactId>
 <version>0.4.4</version>
  <exclusions>
    <exclusion>
      <groupId>com.mysema.querydsl</groupId>
      <artifactId>querydsl-hql-apt</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
 <groupId>com.mysema.querydsl</groupId>
 <artifactId>querydsl-apt-jdk5</artifactId>
 <version>0.4.4</version>
 <scope>provided</scope>
</dependency>
```