

Querydsl

Reference Documentation

Timo Westkämper
Samppa Saarela

Querydsl: Reference Documentation

by Timo Westkämper and Samppa Saarela

0.4.4

Copyright © 2007-2009 Mysema Ltd.

Legal Notice

Copyright © 2007-2009 by Mysema Ltd. This copyrighted material is made available to anyone wishing to use, modify, copy, or redistribute it subject to the terms and conditions of the GNU [Lesser General Public License](#), as published by the Free Software Foundation.

Table of Contents

Preface	v
1. Introduction	1
1.1. Background	1
1.2. Principles	1
2. Getting started with Querydsl	2
2.1. Querying JDO sources	2
Maven integration	2
Using query types	4
Querying with JDOQL	5
2.2. Querying JPA/Hibernate sources	5
Maven integration	5
Using query types	8
Querying with HQL	8
2.3. Querying Collections	9
Make the Querydsl collections API available in your class	9
Use the simple API	9
Use the full API	9
Use the factory methods	9
Use the alias features	10
2.4. Querying SQL/JDBC sources	11
3. Alias usage	12

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-safe bring huge benefits in software development. Domain changes are directly reflected in queries and autocomplete in query construction makes query construction faster and safer.

HQL queries for Hibernate were 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 project's domain types. Also function invocations are constructed in a fully manner.

Consistency is another important principle. The query paths and operations are the same in all implementations and also the Query interfaces are similar.

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 dependency 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 :

```
<http://xslthl.sf.net:tag><dependency></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><groupId></http://xslthl.sf.net:tag>
com.mysema.querydsl
<http://xslthl.sf.net:tag></groupId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><artifactId></http://xslthl.sf.net:tag>
querydsl-jdoql
<http://xslthl.sf.net:tag></artifactId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><version></http://xslthl.sf.net:tag>
0.4.0
<http://xslthl.sf.net:tag></version></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></dependency></http://xslthl.sf.net:tag>
```

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

```
<http://xslthl.sf.net:tag><project></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><build></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><plugins></http://xslthl.sf.net:tag>

...

<http://xslthl.sf.net:tag><plugin></http://xslthl.sf.net:tag>
```

```
<http://xslthl.sf.net:tag><groupId></http://xslthl.sf.net:tag>
com.mysema.maven
<http://xslthl.sf.net:tag></groupId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><artifactId></http://xslthl.sf.net:tag>
maven-apt-plugin
<http://xslthl.sf.net:tag></artifactId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><version></http://xslthl.sf.net:tag>
0.2.0
<http://xslthl.sf.net:tag></version></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><executions></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><execution></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><goals></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><goal></http://xslthl.sf.net:tag>
process
<http://xslthl.sf.net:tag></goal></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></goals></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><configuration></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><outputDirectory></http://xslthl.sf.net:tag>
target/generated-sources/java
<http://xslthl.sf.net:tag></outputDirectory></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><processor></http://xslthl.sf.net:tag>
com.mysema.query.jdoql.apt.JDOAnnotationProcessor
<http://xslthl.sf.net:tag></processor></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></configuration></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></execution></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></executions></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></plugin></http://xslthl.sf.net:tag>

...
```

```
<http://xslthl.sf.net:tag></plugins></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></build></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></project></http://xslthl.sf.net:tag>
```

The `JDOAnnotationProcessor` finds domain types annotated with the `javax.jdo.annotations.PersistenceCapable` annotation and generates Querydsl query types for them. Run `mvn eclipse:eclipse` or clean install and you will get your Query types generated into `target/generated-sources/java`.

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 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");
```

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

```
customer.firstName;
```

The generated Querydsl query types are based on a builtin expression type system. The Querydsl expression archetypes are presented in more detailed on this page : [Querydsl expressions].

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?

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.

Maven integration

Add the following dependency 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 :

```
<http://xslthl.sf.net:tag><dependency></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><groupId></http://xslthl.sf.net:tag>
com.mysema.querydsl
<http://xslthl.sf.net:tag></groupId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><artifactId></http://xslthl.sf.net:tag>
querydsl-hql
<http://xslthl.sf.net:tag></artifactId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><version></http://xslthl.sf.net:tag>
0.4.0
<http://xslthl.sf.net:tag></version></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></dependency></http://xslthl.sf.net:tag>
```

And now, configure the Maven APT plugin :

```
<http://xslthl.sf.net:tag><project></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><build></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><plugins></http://xslthl.sf.net:tag>

...

<http://xslthl.sf.net:tag><plugin></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><groupId></http://xslthl.sf.net:tag>
com.mysema.maven
<http://xslthl.sf.net:tag></groupId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><artifactId></http://xslthl.sf.net:tag>
maven-apt-plugin
<http://xslthl.sf.net:tag></artifactId></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><version></http://xslthl.sf.net:tag>
0.3.0
<http://xslthl.sf.net:tag></version></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><executions></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><execution></http://xslthl.sf.net:tag>
```

```
<http://xslthl.sf.net:tag><goals></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><goal></http://xslthl.sf.net:tag>
process
<http://xslthl.sf.net:tag></goal></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></goals></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><configuration></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><outputDirectory></http://xslthl.sf.net:tag>
target/generated-sources/java
<http://xslthl.sf.net:tag></outputDirectory></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag><processor></http://xslthl.sf.net:tag>
com.mysema.query.hql.apt.JPAAnnotationProcessor
<http://xslthl.sf.net:tag></processor></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></configuration></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></execution></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></executions></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></plugin></http://xslthl.sf.net:tag>

...

<http://xslthl.sf.net:tag></plugins></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></build></http://xslthl.sf.net:tag>

<http://xslthl.sf.net:tag></project></http://xslthl.sf.net:tag>
```

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

Run `mvn eclipse:eclipse` or `clean install` and you will get your Query types generated into `target/generated-sources/java`. 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 HQLQueryImpl is the main Query implementation. It is instantiated like this :

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

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

```
QCustomer customer = QCustomer.customer;
HQLQuery query = new HQLQueryImpl (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?

2.3. Querying Collections

To use the querydsl-collections module with generated query types, add an appropriate APT configuration like in the querydsl-hql or querydsl-jdoql setup setup.

Make the Querydsl collections API available in your class

Add the following static import :

```
import static com.mysema.query.collections.MinApi.*;
```

Use the simple API

```
List<Cat> cats = ...; // some value
QCat cat = new QCat("cat");
for (Cat cat : select(cats, cat.kittens.size().gt(0))) {
    System.out.println(cat.getName());
}
```

Use the full API

And here are some examples on using the full API

```
from(cat,cats).iterate(cat.name);
from(cat,cats).iterate(cat.kittens);
from(cat,cats).where(cat.kittens.size().gt(0)).iterate(cat.name);
from(cat,cats).where(cat.name.eq("Kitty")).iterate(cat.name);
from(cat,cats).where(cat.name.like("Kitt%")).iterate(cat.name);
```

Use the factory methods

If you use the MiniApi without available expressions you can use the factory methods of the MiniApi, which are accessible via the dollar-method, e.g.

```
for (String s : from($"str"), "a","ab","cd","de").where($"str").startsWith("a")).iterate($"str")){
```

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

which prints out

```
a  
ab
```

For multiple variables the provided argument identifies the created path.

And here an example with a created integer path :

```
for (Integer i : from($0),1,2,3,4).where($0.lt(4)).iterate($0)){  
    System.out.println(i);  
}
```

which prints out

```
1  
2  
3
```

And last but not least mixed arguments

```
for (Object o : from(1,2,"abc",5,3).where($().ne("abc")).iterate($())){  
    System.out.println(o);  
}  
{
```

which prints out

```
1  
2  
5  
3
```

Use the alias features

The alias usage builds on the factory methods for Expressions and extends them to support property access. Here are some examples.

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);
}
```

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.

2.4. Querying SQL/JDBC sources

TODO

3. 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.GrammarWithAlias.$;
import static com.mysema.query.alias.GrammarWithAlias.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.