CHAPTER 8

Doctrine Query Language

Introduction

Earlier in this book, we learned about repositories, containers for entities of a specific type. They are used to look up entities by specific criteria, do updates or deletes, and so on. They do their work with the help of finder methods, which are implemented using Doctrine's own entity query language DQL, the Doctrine Query Language. Strictly speaking, DQL is not bound to finder methods or repositories; however, it is usually good practice to put all DQL statements there, just to keep things organized.

DQL itself is a language to query entities. It looks much like SQL, which makes learning DQL easier, but it isn't SQL. While DQL statements can be written as a string like this

```
1   <?php
2   // [..]
3   $query = $em->createQuery(
4            'SELECT u FROM Entity\User u WHERE u.lastName = "Mustermann"'
5   );
6
7   $users = $query->getResult();
```

it is much more convenient to use the *query builder*, especially when constructing dynamic queries:

```
 1   <?php
 2   // [..]
 3   $qb = $this->_em->createQueryBuilder();
 4
 5   $qb->select('u')
 6           ->from('Entity\User', 'u')
 7           ->where($qb->expr()->eq('u.lastName', '?1'))
 8           ->setParameter(1, "Mustermann");
 9
10   $users = $qb->getQuery()->getResult();
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_8

We assume that `$this->_em` holds a reference to the entity manager (which is true for every repository that extends `Doctrine\ORM\EntityRepository`). The entity manager is capable of providing a query builder, which in turn can be used to programmatically construct a query. Thanks to its fluent interface, the code looks pretty elegant. In contrast to the first example, we also utilize value parameters and the `Expr` class, which we will look at in detail in a minute.

Retrieving Results

When executing a query, multiple options exist for retrieving results. When calling `getResult()` on a query object, a PHP array is returned containing all matching entity objects. Alternatively, `getArrayResult()` can be used to get all data in the form of an array. No objects are returned, only all entities' data as an array in a container array. This is useful when dealing with large datasets or for simple display tasks, where no objects are needed in the processing. The method `getScalarResult()` returns a similar result, but fully flat, not nested at all. When a single result is desired, calling `getSingleResult()` or `getSingleScalarResult()` will do. Method `getOneOrNullResult()` may be used if null is desired when no match was found.

If a query includes objects as well as scalar values as well, the result set returned is called "mixed":

```
<?php
// [..]
$qb = $this->_em->createQueryBuilder();

$qb->select('u')
        ->addSelect($qb->expr()->concat('u.firstName', 'u.lastName'))
        ->from('Entity\User', 'u')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->setParameter(1, "Mustermann");

$users = $qb->getQuery()->getResult();
```

In this query, we not only retrieve entities, but also concatenate the user's first and last names. The result of the query looks like this:

```
array
        [0]
              [0] => Object
              [1] => "Max Mustermann"
        [1]
              // ..
```

The result set can be limited, via `setFirstResult($offset)` and `setMaxResults($limit)`, as when building a pagination feature.

Another feature Doctrine 2 offers is retrieving partial objects, entities which have been only partially recreated from the database. To retrieve a partial object, a special syntax is required:

```
<?php
// [..]
$qb = $this->_em->createQueryBuilder();

$qb->select('partial u.{id, firstName}')
        ->from('Entity\User', 'u')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->setParameter(1, "Mustermann");

$users = $qb->getQuery()->getResult();
```

This way, we get back partly reconstituted `User` objects from the database. When omitting the `partial` syntax and simply stating individual fields, the result is a plain array without objects:

```
array(1) {
        [0]=>
                array(2) {
                        ["id"]=> int(1)
                        ["firstName"]=> string(3) "Max"
                }
}
```

It's important to always include the identifier (`id` in this case) in a partial object definition. Otherwise an exception is thrown.

While Partial objects can be very helpful, such as while tweaking the performance of an app, they can be problematic as well. Code dealing with partial objects needs to be aware of the fact that no "real" entities are returned, and certain fields or associations might not be available. Use partial objects with care.

Constructing Basic Queries

The query builder provides methods for the different parts of a query, such as the one shown above:

- public function select($select = null);
- public function delete($delete = null, $alias = null);
- public function update($update = null, $alias = null);
- public function set($key, $value);
- public function from($from, $alias = null);
- public function where($where);
- public function andWhere($where);
- public function orWhere($where);
- public function groupBy($groupBy);

- public function addGroupBy($groupBy);
- public function having($having);
- public function andHaving($having);
- public function orHaving($having);
- public function orderBy($sort, $order = null);
- public function addOrderBy($sort, $order = null);

Expressions like the ones shown above are built using an `Expr` object, which is provided by the query builder when calling its `expr()` method. The `Expr` object provides several methods with which to construct an expression:

- public function andX($x = null);
- public function orX($x = null);
- public function eq($x, $y);
- public function neq($x, $y);
- public function lt($x, $y);
- public function lte($x, $y);
- public function gt($x, $y);
- public function gte($x, $y);
- public function isNull($x);
- public function isNotNull($x);
- public function prod($x, $y);
- public function diff($x, $y);
- public function sum($x, $y);
- public function quot($x, $y);
- public function exists($subquery);
- public function all($subquery);
- public function some($subquery);
- public function any($subquery);
- public function not($restriction);
- public function in($x, $y);
- public function notIn($x, $y);
- public function like($x, $y);
- public function between($val, $x, $y);

- public function trim($x);
- public function concat($x, $y);
- public function lower($x);
- public function upper($x);
- public function length($x);
- public function avg($x);
- public function max ($x);
- public function substr($x, $from, $len);
- public function min($x);
- public function abs($x);
- public function sqrt($x);
- public function count($x);
- public function countDistinct($x);

Expressions are used in the `SELECT`, `WHERE`, `HAVING` or `GROUP` part of a query. However, the query shown above can also be created without using the `Expr` class:

```
1   <?php
2   $qb = $this->_em->createQueryBuilder();
3
4   $qb->select('u')
5           ->addSelect("CONCAT(u.firstName, u.lastName)")
6           ->from('Entity\User', 'u')
7           ->where('u.lastName = ?1')
8           ->setParameter(1, "Mustermann");
9
10  $users = $qb->getQuery()->getResult();
```

This might be necessary sometimes, since not every function or arithmetic operator can be constructed via the `Expr` class. The following aggregate functions are allowed in `SELECT` and `GROUP BY` clauses:

- AVG
- COUNT
- MIN
- MAX
- SUM

The following functions are supported in `SELECT`, `WHERE`, and `HAVING` clauses:

- IDENTITY
- ABS(arithmetic_expression)
- CONCAT(str1, str2)
- CURRENT_DATE()
- CURRENT_TIME()
- CURRENT_TIMESTAMP()
- LENGTH(str)
- LOCATE(needle, haystack [, offset])
- LOWER(str)
- MOD(a, b)
- SIZE(collection)
- SQRT(q)
- SUBSTRING(str, start [, length])
- TRIM([LEADING | TRAILING | BOTH] [“trchar” FROM] str)
- UPPER(str)
- DATE_ADD(date, days, unit)
- DATE_SUB(date, days, unit)
- DATE_DIFF(date1, date2)

Values can be given for placeholders within queries via `setParameter()` or `setParameters($array)`. In the example shown above, number placeholders are used (starting with a “?” symbol). Alternatively, a string placeholder may be used (starting with a “:” symbol):

```
1   <?php
2   // [..]
3   $qb = $this->_em->createQueryBuilder();
4
5   $qb->select('u')
6           ->addSelect($qb->expr()->concat('u.firstName', 'u.lastName'))
7           ->from('Entity\User', 'u')
8           ->where($qb->expr()->eq('u.lastName', ':lastName'))
9           ->setParameter("lastName", "Mustermann");
10
11    $users = $qb->getQuery()->getResult();
```

Whichever way is preferred, one needs to stick to it within a query. Mixing is not allowed.

Constructing Join Queries

Doctrine 2 supports two different types of joins. While regular joins are needed, for example, to limit results via a `WHERE` clause, so-called *fetch joins* are used to fetch related entities for further usage:

```
<?php
// [..]
$qb = $this->_em->createQueryBuilder();

$qb->select('u', 'c')
        ->from('Entity\User', 'u')
        ->leftJoin('u.contactData', 'c')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->setParameter(1, "Mustermann");

$users = $qb->getQuery()->getResult();
```

The `select('u', 'c')` makes the join a fetch join. The referenced `ContactData` object is part of the result set. However, when omitting the `'c'` in the `select` clause, the `ContactData` object is not part of the result anymore, however, it can still be used, for example, within the `where` clause:

```
<?php
// [..]

$qb = $this->_em->createQueryBuilder();

$qb->select('u')
        ->from('Entity\User', 'u')
        ->leftJoin('u.contactData', 'c')
        ->where($qb->expr()->eq('u.lastName', '?1'))
        ->andWhere($qb->expr()->eq('c.email', '?2'))
        ->setParameter(1, "Mustermann")
        ->setParameter(2, "max.mustermann@example.com");

$users = $qb->getQuery()->getResult();
```

Summary

We covered the most fundamental aspects of Doctrine's very own query language. As said earlier, DQL itself is a language to query entities. It looks much like SQL-which makes learning DQL easier- but it isn't SQL. When working with Doctrine, a proper understanding of DQL is needed.