Doctrine 2 ORM

https://www.doctrine-project.org/projects/doctrine-orm/en/2.8/index.html

Tutorials

Reference

Instalar

```
composer require doctrine/orm
```

Welcome to Doctrine 2 ORM's documentation!

The Doctrine documentation is comprised of tutorials, a reference section and cookbook articles that explain different parts of the Object Relational mapper.

Doctrine DBAL and Doctrine Common both have their own documentation.

Getting Help

If this documentation is not helping to answer questions you have about Doctrine ORM don't panic. You can get help from different sources:

- There is a FAQ with answers to frequent questions.
- The Doctrine Mailing List

- Slack chat room #orm
- Report a bug on GitHub.
- On Twitter with `#doctrine2`
- On StackOverflow

If you need more structure over the different topics you can browse the table of contents .

Getting Started

Mapping Objects onto a Database

Working with Objects

Advanced Topics

Tutorials

Changelogs

Cookbook

Installation and Configuration

Doctrine can be installed with Composer.

Define the following requirement in your `composer.json` file:

```
{
    "require": {
        "doctrine/orm": "*"
    }
}
```

Then call `composer install` from your command line. If you don't know how Composer works, check out their Getting Started to set up.

Class loading

Autoloading is taken care of by Composer. You just have to include the composer autoload file in your project:

```
<?php
// bootstrap.php
// Include Composer Autoload (relative to project root).
require_once "vendor/autoload.php";
```

Obtaining an EntityManager

Once you have prepared the class loading, you acquire an *EntityManager* instance. The EntityManager class is the primary access point to ORM functionality provided by Doctrine.

```
<?php
// bootstrap.php
require_once "vendor/autoload.php";

use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

$paths = array("/path/to/entity-files");
$isDevMode = false;

// the connection configuration
$dbParams = array(
    'driver'   => 'pdo_mysql',
    'user'     => 'root',
    'password' => '',
    'dbname'   => 'foo',
);

$config = Setup::createAnnotationMetadataConfiguration($paths, $isDevMode);
$entityManager = EntityManager::create($dbParams, $config);
```

Or if you prefer XML:

<?php $paths = array("/path/to/xml-mappings"); $config = Setup::createXMLMetadataConfiguration($paths, $isDevMode); $entityManager = EntityManager::create($dbParams, $config);

Or if you prefer YAML:

```
<?php
$paths = array("/path/to/yml-mappings");
$config = Setup::createYAMLMetadataConfiguration($paths, $isDevMode);
$entityManager = EntityManager::create($dbParams, $config);
```

> If you want to use yml mapping you should add yaml dependency to your `composer.json`:
>
> ---
>
> symfony/yaml: *

Inside the `Setup` methods several assumptions are made:

- If `$isDevMode` is true caching is done in memory with the `ArrayCache`. Proxy objects are recreated on every request.
- If `$isDevMode` is false, check for Caches in the order APC, Xcache, Memcache (127.0.0.1:11211), Redis (127.0.0.1:6379) unless `$cache` is passed as fourth argument.
- If `$isDevMode` is false, set then proxy classes have to be explicitly created through the command line.
- If third argument `$proxyDir` is not set, use the systems temporary directory.

If you want to configure Doctrine in more detail, take a look at the Advanced Configuration section.

You can learn more about the database connection configuration in the Doctrine DBAL connection configuration reference.

Setting up the Commandline Tool

Doctrine ships with a number of command line tools that are very helpful during development. You can call this command from the Composer binary directory:

```
php vendor/bin/doctrine
```

You need to register your applications EntityManager to the console tool to make use of the tasks by creating a `cli-config.php` file with the following content:

```
<?php
use Doctrine\ORM\Tools\Console\ConsoleRunner;

// replace with file to your own project bootstrap
require_once 'bootstrap.php';

// replace with mechanism to retrieve EntityManager in your app
$entityManager = GetEntityManager();

return ConsoleRunner::createHelperSet($entityManager);
```

Frequently Asked Questions

This FAQ is a work in progress. We will add lots of questions and not answer them right away just to remember what is often asked. If you stumble across an unanswered question please write a mail to the mailing-list or join the #doctrine channel on Freenode IRC.

Database Schema

How do I set the charset and collation for MySQL tables?

You can't set these values inside the annotations, yml or xml mapping files. To make a database work with the default charset and collation you should configure MySQL to use it as default charset, or create the database with charset and collation details. This way they get inherited to all newly created database tables and columns.

Entity Classes

How can I add default values to a column?

Doctrine does not support to set the default values in columns through the DEFAULT keyword in SQL. This is not necessary however, you can just use your class properties as default values. These are then used upon insert:

```
class User
{
    const STATUS_DISABLED = 0;
    const STATUS_ENABLED = 1;
    private $algorithm = "sha1";
    private $status = self:STATUS_DISABLED;
}
```

Mapping

Why do I get exceptions about unique constraint failures during `$em->flush()`?

Doctrine does not check if you are re-adding entities with a primary key that already exists or adding entities to a collection twice. You have to check for both conditions yourself in the code before calling `$em->flush()` if you know that unique constraint failures can occur.

In Symfony2 for example there is a Unique Entity Validator to achieve this task.

For collections you can check with `$collection->contains($entity)` if an entity is already part of this collection. For a FETCH=LAZY collection this will initialize the collection, however for FETCH=EXTRA_LAZY this method will use SQL to determine if this entity is already part of the collection.

Associations

What is wrong when I get an InvalidArgumentException A new entity was found through the relationship..?

This exception is thrown during `EntityManager#flush()` when there exists an object in the identity map that contains a reference to an object that Doctrine does not know about. Say for example you grab a User-entity from the database with a specific id and set a completely new object into one of the associations of the User object. If you then call `EntityManager#flush()` without letting Doctrine know about this new object using `EntityManager#persist($newObject)` you will see this exception.

You can solve this exception by:

- Calling `EntityManager#persist($newObject)` on the new object
- Using cascade=persist on the association that contains the new object

How can I filter an association?

You should use DQL queries to query for the filtered set of entities.

I call clear() on a One-To-Many collection but the entities are not deleted

This is an expected behavior that has to do with the inverse/owning side handling of Doctrine. By definition a One-To-Many association is on the inverse side, that means changes to it will not be recognized by Doctrine.

If you want to perform the equivalent of the clear operation you have to iterate the collection and set the owning side many-to-one reference to NULL as well to detach all entities from the collection. This will trigger the appropriate UPDATE statements on the database.

How can I add columns to a many-to-many table?

The many-to-many association is only supporting foreign keys in the table definition To work with many-to-many tables containing extra columns you have to use the foreign keys as primary keys feature of Doctrine ORM.

See the tutorial on composite primary keys for more information.

How can i paginate fetch-joined collections?

If you are issuing a DQL statement that fetches a collection as well you cannot easily iterate over this collection using a LIMIT statement (or vendor equivalent).

Doctrine does not offer a solution for this out of the box but there are several extensions that do:

- DoctrineExtensions
- Pagerfanta

Why does pagination not work correctly with fetch joins?

Pagination in Doctrine uses a LIMIT clause (or vendor equivalent) to restrict the results. However when fetch-joining this is not returning the correct number of results since joining with a one-to-many or many-to-many association multiplies the number of rows by the number of associated entities.

See the previous question for a solution to this task.

Inheritance

Can I use Inheritance with Doctrine ORM?

Yes, you can use Single- or Joined-Table Inheritance in ORM.

See the documentation chapter on inheritance mapping for the details.

Why does Doctrine not create proxy objects for my inheritance hierarchy?

If you set a many-to-one or one-to-one association target-entity to any parent class of an inheritance hierarchy Doctrine does not know what PHP class the foreign is actually of. To find this out it has to execute a SQL query to look this information up in the database.

EntityGenerator

Why does the EntityGenerator not do X?

The EntityGenerator is not a full fledged code-generator that solves all tasks. Code-Generation is not a first-class priority in Doctrine 2 anymore (compared to Doctrine 1). The EntityGenerator is supposed to kick-start you, but not towards 100%.

Why does the EntityGenerator not generate inheritance correctly?

Just from the details of the discriminator map the EntityGenerator cannot guess the inheritance hierarchy. This is why the generation of inherited entities does not fully work. You have to adjust some additional code to get this one working correctly.

Performance

Why is an extra SQL query executed every time I fetch an entity with a one-to-one relation?

If Doctrine detects that you are fetching an inverse side one-to-one association it has to execute an additional query to load this object, because it cannot know if there is no such object (setting null) or if it should set a proxy and which id this proxy has.

To solve this problem currently a query has to be executed to find out this information.

Doctrine Query Language

What is DQL?

DQL stands for Doctrine Query Language, a query language that very much looks like SQL but has some important benefits when using Doctrine:

- It uses class names and fields instead of tables and columns, separating concerns between backend and your object model.
- It utilizes the metadata defined to offer a range of shortcuts when writing. For example you do not have to specify the ON clause of joins, since Doctrine already knows about them.
- It adds some functionality that is related to object management and transforms them into SQL.

It also has some drawbacks of course:

- The syntax is slightly different to SQL so you have to learn and remember the differences.
- To be vendor independent it can only implement a subset of all the existing SQL dialects. Vendor specific functionality and optimizations cannot be used through DQL unless implemented by you explicitly.
- For some DQL constructs subselects are used which are known to be slow in MySQL.

Can I sort by a function (for example ORDER BY RAND()) in DQL?

No, it is not supported to sort by function in DQL. If you need this functionality you should either use a native-query or come up with another solution. As a side note: Sorting with ORDER BY RAND() is painfully slow starting with 1000 rows.

Is it better to write DQL or to generate it with the query builder?

The purpose of the `QueryBuilder` is to generate DQL dynamically, which is useful when you have optional filters, conditional joins, etc.

But the `QueryBuilder` is not an alternative to DQL, it actually generates DQL queries at runtime, which are then interpreted by Doctrine. This means that using the `QueryBuilder` to build and run a query is actually always slower than only running the corresponding DQL query.

So if you only need to generate a query and bind parameters to it, you should use plain DQL, as this is a simpler and much more readable solution. You should only use the `QueryBuilder` when you can't achieve what you want to do with a DQL query.

A Query fails, how can I debug it?

First, if you are using the QueryBuilder you can use `$queryBuilder->getDQL()` to get the DQL string of this query. The corresponding SQL you can get from the Query instance by calling `$query->getSQL()`.

```
<?php
$dql = "SELECT u FROM User u";
$query = $entityManager->createQuery($dql);
var_dump($query->getSQL());

$qb = $entityManager->createQueryBuilder();
$qb->select('u')->from('User', 'u');
var_dump($qb->getDQL());
```

Basic Mapping

This guide explains the basic mapping of entities and properties. After working through this guide you should know:

- How to create PHP objects that can be saved to the database with Doctrine;
- How to configure the mapping between columns on tables and properties on entities;
- What Doctrine mapping types are;
- Defining primary keys and how identifiers are generated by Doctrine;
- How quoting of reserved symbols works in Doctrine.

Mapping of associations will be covered in the next chapter on Association Mapping.

Guide Assumptions

You should have already installed and configure Doctrine.

Creating Classes for the Database

Every PHP object that you want to save in the database using Doctrine is called an Entity. The term Entity describes objects that have an identity over many independent requests. This identity is usually achieved by assigning a unique identifier to an entity. In this tutorial the following `Message` PHP class will serve as the example Entity:

```
<?php
class Message
{
  private $id;
  private $text;
  private $postedAt;
}
```

Because Doctrine is a generic library, it only knows about your entities because you will describe their existence and structure using mapping metadata, which is configuration that tells Doctrine how your entity should be stored in the database. The documentation will often speak of mapping something, which means writing the mapping metadata that describes your entity.

Doctrine provides several different ways to specify object-relational mapping metadata:

- Docblock Annotations
- XML
- YAML
- PHP code

This manual will usually show mapping metadata via docblock annotations, though many examples also show the equivalent configuration in YAML and XML.

All metadata drivers perform equally. Once the metadata of a class has been read from the source (annotations, xml or yaml) it is stored in an instance of the `Doctrine\ORM\Mapping\ClassMetadata` class and these instances are stored in the metadata cache. If you're not using a metadata cache (not recommended!) then the XML driver is the fastest.

Marking our `Message` class as an entity for Doctrine is straightforward:

PHP, XML e YAML
<?php
/** @Entity */
class Message
{
  //...
}

With no additional information, Doctrine expects the entity to be saved into a table with the same name as the class in our case `Message`. You can change this by configuring information about the table:

- *PHP*

```
<?php
/**
 * @Entity
 * @Table(name="message")
 */
class Message
{
    //...
}
```

Now the class `Message` will be saved and fetched from the table `message`.

Property Mapping

The next step after marking a PHP class as an entity is mapping its properties to columns in a table.

To configure a property use the `@Column` docblock annotation. The `type` attribute specifies the Doctrine Mapping Type to use for the field. If the type is not specified, `string` is used as the default.

- *PHP*

```
<?php
/** @Entity */
class Message
{
    /** @Column(type="integer") */
    private $id;
    /** @Column(length=140) */
    private $text;
    /** @Column(type="datetime", name="posted_at") */
    private $postedAt;
}
```

When we don't explicitly specify a column name via the `name` option, Doctrine assumes the field name is also the column name. This means that:

- the `id` property will map to the column `id` using the type `integer`;
- the `text` property will map to the column `text` with the default mapping type `string`;
- the `postedAt` property will map to the `posted_at` column with the `datetime` type.

The Column annotation has some more attributes. Here is a complete list:

- `type`: (optional, defaults to 'string') The mapping type to use for the column.
- `name`: (optional, defaults to field name) The name of the column in the database.
- `length`: (optional, default 255) The length of the column in the database. (Applies only if a string-valued column is used).
- `unique`: (optional, default FALSE) Whether the column is a unique key.
- `nullable`: (optional, default FALSE) Whether the database column is nullable.
- `precision`: (optional, default 0) The precision for a decimal (exact numeric) column (applies only for decimal column), which is the maximum number of digits that are stored for the values.
- `scale`: (optional, default 0) The scale for a decimal (exact numeric) column (applies only for decimal column), which represents the number of digits to the right of the decimal point and must not be greater than *precision*.
- `columnDefinition`: (optional) Allows to define a custom DDL snippet that is used to create the column. Warning: This normally confuses the SchemaTool to always detect the column as changed.
- `options`: (optional) Key-value pairs of options that get passed to the underlying database platform when generating DDL statements.

Doctrine Mapping Types

The `type` option used in the `@Column` accepts any of the existing Doctrine types or even your own custom types. A Doctrine type defines the conversion between PHP and SQL types, independent from the database vendor you are using. All Mapping Types that ship with Doctrine are fully portable between the supported database systems.

As an example, the Doctrine Mapping Type `string` defines the mapping from a PHP string to a SQL VARCHAR (or VARCHAR2 etc. depending on the RDBMS brand). Here is a quick overview of the built-in mapping types:

- `string`: Type that maps a SQL VARCHAR to a PHP string.
- `integer`: Type that maps a SQL INT to a PHP integer.
- `smallint`: Type that maps a database SMALLINT to a PHP integer.
- `bigint`: Type that maps a database BIGINT to a PHP string.
- `boolean`: Type that maps a SQL boolean or equivalent (TINYINT) to a PHP boolean.
- `decimal`: Type that maps a SQL DECIMAL to a PHP string.
- `date`: Type that maps a SQL DATETIME to a PHP DateTime object.
- `time`: Type that maps a SQL TIME to a PHP DateTime object.
- `datetime`: Type that maps a SQL DATETIME/TIMESTAMP to a PHP DateTime object.
- `datetimetz`: Type that maps a SQL DATETIME/TIMESTAMP to a PHP DateTime object with timezone.
- `text`: Type that maps a SQL CLOB to a PHP string.
- `object`: Type that maps a SQL CLOB to a PHP object using `serialize()` and `unserialize()`
- `array`: Type that maps a SQL CLOB to a PHP array using `serialize()` and `unserialize()`

- `simple_array`: Type that maps a SQL CLOB to a PHP array using `implode()` and `explode()`, with a comma as delimiter. *IMPORTANT* Only use this type if you are sure that your values cannot contain a ",".
- `json_array`: Type that maps a SQL CLOB to a PHP array using `json_encode()` and `json_decode()`
- `float`: Type that maps a SQL Float (Double Precision) to a PHP double. *IMPORTANT*: Works only with locale settings that use decimal points as separator.
- `guid`: Type that maps a database GUID/UUID to a PHP string. Defaults to varchar but uses a specific type if the platform supports it.
- `blob`: Type that maps a SQL BLOB to a PHP resource stream

A cookbook article shows how to define your own custom mapping types.

DateTime and Object types are compared by reference, not by value. Doctrine updates this values if the reference changes and therefore behaves as if these objects are immutable value objects.

All Date types assume that you are exclusively using the default timezone set by date_default_timezone_set() or by the php.ini configuration `date.timezone`. Working with different timezones will cause troubles and unexpected behavior.

If you need specific timezone handling you have to handle this in your domain, converting all the values back and forth from UTC. There is also a cookbook entry on working with datetimes that gives hints for implementing multi timezone applications.

Identifiers / Primary Keys

Every entity class must have an identifier/primary key. You can select the field that serves as the identifier with the `@Id` annotation.

- *PHP*

```
<?php
class Message
{
    /**
     * @Id
     * @Column(type="integer")
     * @GeneratedValue
     */
    private $id;
    //...
}
```

In most cases using the automatic generator strategy (`@GeneratedValue`) is what you want. It defaults to the identifier generation mechanism your current database vendor prefers: AUTO_INCREMENT with MySQL, sequences with PostgreSQL
and Oracle and so on.

Identifier Generation Strategies

The previous example showed how to use the default identifier generation strategy without knowing the underlying database with the AUTO-detection strategy. It is also possible to specify the identifier generation strategy more explicitly, which allows you to make use of some additional features.

Here is the list of possible generation strategies:

- `AUTO` (default): Tells Doctrine to pick the strategy that is preferred by the used database platform. The preferred strategies are IDENTITY for MySQL, SQLite, MsSQL and SQL Anywhere and SEQUENCE for Oracle and PostgreSQL. This strategy provides full portability.
- `SEQUENCE`: Tells Doctrine to use a database sequence for ID generation. This strategy does currently not provide full portability. Sequences are supported by Oracle, PostgreSql and SQL Anywhere.
- `IDENTITY`: Tells Doctrine to use special identity columns in the database that generate a value on insertion of a row. This strategy does currently not provide full portability and is supported by the following platforms: MySQL/SQLite/SQL Anywhere (AUTO_INCREMENT), MSSQL (IDENTITY) and PostgreSQL (SERIAL).
- `UUID`: Tells Doctrine to use the built-in Universally Unique Identifier generator. This strategy provides full portability.
- `TABLE`: Tells Doctrine to use a separate table for ID generation. This strategy provides full portability. ***This strategy is not yet implemented!***
- `NONE`: Tells Doctrine that the identifiers are assigned (and thus generated) by your code. The assignment must take place before a new entity is passed to `EntityManager#persist`. NONE is the same as leaving off the @GeneratedValue entirely.
- `CUSTOM`: With this option, you can use the `@CustomIdGenerator` annotation. It will allow you to pass a class of your own to generate the identifiers.

Sequence Generator

The Sequence Generator can currently be used in conjunction with Oracle or Postgres and allows some additional configuration options besides specifying the sequence's name:

- *PHP*

```
<?php
class Message
{
    /**
     * @Id
     * @GeneratedValue(strategy="SEQUENCE")
     * @SequenceGenerator(sequenceName="message_seq", initialValue=1,
allocationSize=100)
     */
    protected $id = null;
    //...
}
```

The initial value specifies at which value the sequence should start.

The allocationSize is a powerful feature to optimize INSERT performance of Doctrine. The allocationSize specifies by how much values the sequence is incremented whenever the next value is retrieved. If this is larger than 1 (one) Doctrine can generate identifier values for the allocationSizes amount of entities. In the above example with `allocationSize=100` Doctrine ORM would only need to access the sequence once to generate the identifiers for 100 new entities.

The default allocationSize for a @SequenceGenerator is currently 10.

The allocationSize is detected by SchemaTool and transformed into an INCREMENT BY clause in the CREATE SEQUENCE statement. For a database schema created manually (and not SchemaTool) you have to make sure that the allocationSize configuration option is never larger than the actual sequences INCREMENT BY value, otherwise you may get duplicate keys.

It is possible to use strategy=AUTO and at the same time specifying a @SequenceGenerator. In such a case, your custom sequence settings are used in the case where the preferred strategy of the underlying platform is SEQUENCE, such as for Oracle and PostgreSQL.

Composite Keys

With Doctrine ORM you can use composite primary keys, using `@Id` on more then one column. Some restrictions exist opposed to using a single identifier in this case: The use of the `@GeneratedValue` annotation is not supported, which means you can only use composite keys if you generate the primary key values yourself before calling `EntityManager#persist()` on the entity.

More details on composite primary keys are discussed in a dedicated tutorial.

Quoting Reserved Words

Sometimes it is necessary to quote a column or table name because of reserved word conflicts. Doctrine does not quote identifiers automatically, because it leads to more problems than it would solve. Quoting tables and column names needs to be done explicitly using ticks in the definition.

```
<?php
/** @Column(name="`number`", type="integer") */
private $number;
```

Doctrine will then quote this column name in all SQL statements according to the used database platform.

Identifier Quoting does not work for join column names or discriminator column names unless you are using a custom `QuoteStrategy`.

For more control over column quoting the `Doctrine\ORM\Mapping\QuoteStrategy` interface was introduced in ORM. It is invoked for every column, table, alias and other SQL names. You can implement the QuoteStrategy and set it by calling `Doctrine\ORM\Configuration#setQuoteStrategy()`.

The ANSI Quote Strategy was added, which assumes quoting is not necessary for any SQL name. You can use it with the following code:

```
<?php
use Doctrine\ORM\Mapping\AnsiQuoteStrategy;

$configuration->setQuoteStrategy(new AnsiQuoteStrategy());
```

Association Mapping

This chapter explains mapping associations between objects.

Instead of working with foreign keys in your code, you will always work with references to objects instead and Doctrine will convert those references to foreign keys internally.

- A reference to a single object is represented by a foreign key.
- A collection of objects is represented by many foreign keys pointing to the object holding the collection

This chapter is split into three different sections.

- A list of all the possible association mapping use-cases is given.
- Association Mapping are explained that simplify the use-case examples.
- Association Mapping are introduced that contain entities in associations.

One tip for working with relations is to read the relation from left to right, where the left word refers to the current Entity. For example:

- OneToMany - One instance of the current Entity has Many instances (references) to the referred Entity.
- ManyToOne - Many instances of the current Entity refer to One instance of the referred Entity.
- OneToOne - One instance of the current Entity refers to One instance of the referred Entity.

See below for all the possible relations.

An association is considered to be unidirectional if only one side of the association has a property referring to the other side.

To gain a full understanding of associations you should also read about owning and inverse sides of associations

Many-To-One, Unidirectional

A many-to-one association is the most common association between objects. Example: Many Users have One Address:

- *PHP*

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * @ManyToOne(targetEntity="Address")
     * @JoinColumn(name="address_id", referencedColumnName="id")
     */
    private $address;
}

/** @Entity */
class Address
{
    // ...
}
```

The above `@JoinColumn` is optional as it would default to `address_id` and `id` anyways. You can omit it and let it use the defaults.

Generated MySQL Schema:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    address_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE Address (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

ALTER TABLE User ADD FOREIGN KEY (address_id) REFERENCES Address(id);
```

One-To-One, Unidirectional

Here is an example of a one-to-one association with a `Product` entity that references one `Shipment` entity.

- *PHP*

```
<?php
/** @Entity */
class Product
{
    // ...

    /**
     * One Product has One Shipment.
     * @OneToOne(targetEntity="Shipment")
     * @JoinColumn(name="shipment_id", referencedColumnName="id")
     */
    private $shipment;

    // ...
}

/** @Entity */
class Shipment
{
    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Product (
    id INT AUTO_INCREMENT NOT NULL,
    shipment_id INT DEFAULT NULL,
    UNIQUE INDEX UNIQ_6FBC94267FE4B2B (shipment_id),
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Shipment (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Product ADD FOREIGN KEY (shipment_id) REFERENCES Shipment(id);
```

One-To-One, Bidirectional

Here is a one-to-one relationship between a `Customer` and a `Cart`. The `Cart` has a reference back to the `Customer` so it is bidirectional.

Here we see the `mappedBy` and `inversedBy` annotations for the first time. They are used to tell Doctrine which property on the other side refers to the object.

- *PHP*

```
<?php
/** @Entity */
class Customer
{
    // ...

    /**
     * One Customer has One Cart.
     * @OneToOne(targetEntity="Cart", mappedBy="customer")
     */
    private $cart;

    // ...
}

/** @Entity */
class Cart
{
    // ...

    /**
     * One Cart has One Customer.
     * @OneToOne(targetEntity="Customer", inversedBy="cart")
     * @JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;

    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Cart (
   id INT AUTO_INCREMENT NOT NULL,
   customer_id INT DEFAULT NULL,
   PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Customer (
   id INT AUTO_INCREMENT NOT NULL,
   PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Cart ADD FOREIGN KEY (customer_id) REFERENCES Customer(id);
```

We had a choice of sides on which to place the `inversedBy` attribute. Because it is on the `Cart`, that is the owning side of the relation, and thus holds the foreign key.

One-To-One, Self-referencing

You can define a self-referencing one-to-one relationships like below.

```
<?php
/** @Entity */
class Student
{
    // ...

    /**
     * One Student has One Mentor.
     * @OneToOne(targetEntity="Student")
     * @JoinColumn(name="mentor_id", referencedColumnName="id")
     */
    private $mentor;

    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

With the generated MySQL Schema:

```
CREATE TABLE Student (
   id INT AUTO_INCREMENT NOT NULL,
   mentor_id INT DEFAULT NULL,
   PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Student ADD FOREIGN KEY (mentor_id) REFERENCES Student(id);
```

One-To-Many, Bidirectional

A one-to-many association has to be bidirectional, unless you are using a join table. This is because the many side in a one-to-many association holds the foreign key, making it the owning side. Doctrine needs the many side defined in order to understand the association.

This bidirectional mapping requires the `mappedBy` attribute on the one side and the `inversedBy` attribute on the many side.

This means there is no difference between a bidirectional one-to-many and a bidirectional many-to-one.

- *PHP*

```
<?php
use Doctrine\Common\Collections\ArrayCollection;

/** @Entity */
class Product
{
    // ...
    /**
     * One product has many features. This is the inverse side.
     * @OneToMany(targetEntity="Feature", mappedBy="product")
     */
    private $features;
    // ...

    public function __construct() {
        $this->features = new ArrayCollection();
    }
}

/** @Entity */
class Feature
{
    // ...
    /**
     * Many features have one product. This is the owning side.
     * @ManyToOne(targetEntity="Product", inversedBy="features")
     * @JoinColumn(name="product_id", referencedColumnName="id")
     */
    private $product;
    // ...
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Product (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Feature (
    id INT AUTO_INCREMENT NOT NULL,
    product_id INT DEFAULT NULL,
    PRIMARY KEY(id)
```

) ENGINE = InnoDB;
ALTER TABLE Feature ADD FOREIGN KEY (product_id) REFERENCES Product(id);

One-To-Many, Unidirectional with Join Table

A unidirectional one-to-many association can be mapped through a join table. From Doctrine's point of view, it is simply mapped as a unidirectional many-to-many whereby a unique constraint on one of the join columns enforces the one-to-many cardinality.

The following example sets up such a unidirectional one-to-many association:

- *PHP*

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * Many User have Many Phonenumbers.
     * @ManyToMany(targetEntity="Phonenumber")
     * @JoinTable(name="users_phonenumbers",
     *      joinColumns={@JoinColumn(name="user_id", 
referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="phonenumber_id", 
referencedColumnName="id", unique=true)}
     *      )
     */
    private $phonenumbers;

    public function __construct()
    {
        $this->phonenumbers = new \Doctrine\Common\Collections\
ArrayCollection();
    }

    // ...
}

/** @Entity */
class Phonenumber
{
    // ...
}
```

Generates the following MySQL Schema:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE users_phonenumbers (
    user_id INT NOT NULL,
    phonenumber_id INT NOT NULL,
    UNIQUE INDEX users_phonenumbers_phonenumber_id_uniq (phonenumber_id),
    PRIMARY KEY(user_id, phonenumber_id)
) ENGINE = InnoDB;

CREATE TABLE Phonenumber (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

ALTER TABLE users_phonenumbers ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE users_phonenumbers ADD FOREIGN KEY (phonenumber_id) REFERENCES
Phonenumber(id);
```

One-To-Many, Self-referencing

You can also setup a one-to-many association that is self-referencing. In this example we setup a hierarchy of `Category` objects by creating a self referencing relationship. This effectively models a hierarchy of categories and from the database perspective is known as an adjacency list approach.

- *PHP*

```
<?php
/** @Entity */
class Category
{
    // ...
    /**
     * One Category has Many Categories.
     * @OneToMany(targetEntity="Category", mappedBy="parent")
     */
    private $children;

    /**
     * Many Categories have One Category.
     * @ManyToOne(targetEntity="Category", inversedBy="children")
     * @JoinColumn(name="parent_id", referencedColumnName="id")
     */
    private $parent;
    // ...

    public function __construct() {
        $this->children = new \Doctrine\Common\Collections\ArrayCollection();
    }
}
```

Note that the @JoinColumn is not really necessary in this example, as the defaults would be the same.

Generated MySQL Schema:

```
CREATE TABLE Category (
    id INT AUTO_INCREMENT NOT NULL,
    parent_id INT DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Category ADD FOREIGN KEY (parent_id) REFERENCES Category(id);
```

Many-To-Many, Unidirectional

Real many-to-many associations are less common. The following example shows a unidirectional association between User and Group entities:

- *PHP*

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * Many Users have Many Groups.
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="users_groups",
     *      joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="group_id",
referencedColumnName="id")}
     *      )
     */
    private $groups;

    // ...

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\ArrayCollection();
    }
}

/** @Entity */
class Group
{
    // ...
}
```

Generated MySQL Schema:

```
CREATE TABLE User (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE users_groups (
    user_id INT NOT NULL,
    group_id INT NOT NULL,
    PRIMARY KEY(user_id, group_id)
) ENGINE = InnoDB;
CREATE TABLE Group (
    id INT AUTO_INCREMENT NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE users_groups ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE users_groups ADD FOREIGN KEY (group_id) REFERENCES Group(id);
```

Why are many-to-many associations less common? Because frequently you want to associate additional attributes with an association, in which case you introduce an association class. Consequently, the direct many-to-many association disappears and is replaced by one-to-many/many-to-one associations between the 3 participating classes.

Many-To-Many, Bidirectional

Here is a similar many-to-many relationship as above except this one is bidirectional.

- *PHP*

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * Many Users have Many Groups.
     * @ManyToMany(targetEntity="Group", inversedBy="users")
     * @JoinTable(name="users_groups")
     */
    private $groups;

    public function __construct() {
        $this->groups = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}

/** @Entity */
class Group
{
    // ...
    /**
     * Many Groups have Many Users.
     * @ManyToMany(targetEntity="User", mappedBy="groups")
     */
    private $users;

    public function __construct() {
        $this->users = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}
```

The MySQL schema is exactly the same as for the Many-To-Many uni-directional case above.

Owning and Inverse Side on a ManyToMany Association

For Many-To-Many associations you can chose which entity is the owning and which the inverse side. There is a very simple semantic rule to decide which side is more suitable to be the owning side from a developers perspective. You only have to ask yourself which entity is responsible for the connection management, and pick that as the owning side.

Take an example of two entities `Article` and `Tag`. Whenever you want to connect an Article to a Tag and vice-versa, it is mostly the Article that is responsible for this relation. Whenever you add a new article, you want to connect it with existing or new tags. Your Create Article form will probably support this notion and allow specifying the tags directly. This is why you should pick the Article as owning side, as it makes the code more understandable:

```
<?php
class Article
{
    private $tags;

    public function addTag(Tag $tag)
    {
        $tag->addArticle($this); // synchronously updating inverse side
        $this->tags[] = $tag;
    }
}

class Tag
{
    private $articles;

    public function addArticle(Article $article)
    {
        $this->articles[] = $article;
    }
}
```

This allows to group the tag adding on the `Article` side of the association:

```
<?php
$article = new Article();
$article->addTag($tagA);
$article→addTag($tagB);
```

Many-To-Many, Self-referencing

You can even have a self-referencing many-to-many association. A common scenario is where a `User` has friends and the target entity of that relationship is a `User` so it is self referencing. In this example it is bidirectional so `User` has a field named `$friendsWithMe` and `$myFriends`.

```
<?php
/** @Entity */
class User
{
    // ...

    /**
     * Many Users have Many Users.
     * @ManyToMany(targetEntity="User", mappedBy="myFriends")
     */
    private $friendsWithMe;

    /**
     * Many Users have many Users.
     * @ManyToMany(targetEntity="User", inversedBy="friendsWithMe")
     * @JoinTable(name="friends",
     *      joinColumns={@JoinColumn(name="user_id", 
referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="friend_user_id", 
referencedColumnName="id")}
     *      )
     */
    private $myFriends;

    public function __construct() {
        $this->friendsWithMe = new \Doctrine\Common\Collections\
ArrayCollection();
        $this->myFriends = new \Doctrine\Common\Collections\ArrayCollection();
    }

    // ...
}
```

Generated MySQL Schema:

CREATE TABLE User (
  id INT AUTO_INCREMENT NOT NULL,
  PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE friends (
  user_id INT NOT NULL,
  friend_user_id INT NOT NULL,
  PRIMARY KEY(user_id, friend_user_id)
) ENGINE = InnoDB;
ALTER TABLE friends ADD FOREIGN KEY (user_id) REFERENCES User(id);
ALTER TABLE friends ADD FOREIGN KEY (friend_user_id) REFERENCES User(id);

Mapping Defaults

The `@JoinColumn` and `@JoinTable` definitions are usually optional and have sensible default values. The defaults for a join column in a one-to-one/many-to-one association is as follows:

```
name: "_id"
referencedColumnName: "id"
```

As an example, consider this mapping:

- *PHP*

<?php /
** @OneToOne(targetEntity="Shipment") */
private $shipment;

- *XML*
- *YAML*

This is essentially the same as the following, more verbose, mapping:

PHP

```
<?php
```

```
/**
 * One Product has One Shipment.
 * @OneToOne(targetEntity="Shipment")
 * @JoinColumn(name="shipment_id", referencedColumnName="id")
 */
private $shipment;
```

The @JoinTable definition used for many-to-many mappings has similar defaults. As an example, consider this mapping:

- *PHP*

```
<?php
class User
{
    //...
    /** @ManyToMany(targetEntity="Group") */
    private $groups;
    //...
}
```

This is essentially the same as the following, more verbose, mapping:

- *PHP*

```
<?php
class User
{
    //...
    /**
     * Many Users have Many Groups.
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="User_Group",
     *      joinColumns={@JoinColumn(name="User_id",
referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="Group_id",
referencedColumnName="id")}
     *      )
     */
    private $groups;
    //...
}
```

In that case, the name of the join table defaults to a combination of the simple, unqualified class names of the participating classes, separated by an underscore character. The names of the join columns default to the simple, unqualified class name of the targeted class followed by _id. The referencedColumnName always defaults to id, just as in one-to-one or many-to-one mappings.

If you accept these defaults, you can reduce the mapping code to a minimum.

Collections

Unfortunately, PHP arrays, while being great for many things, are missing features that make them suitable for lazy loading in the context of an ORM. This is why in all the examples of many-valued associations in this manual we will make use of a `Collection` interface and its default implementation `ArrayCollection` that are both defined in the `Doctrine\Common\`

`Collections` namespace. A collection implements the PHP interfaces `ArrayAccess`, `Traversable` and `Countable`.

> The Collection interface and ArrayCollection class, like everything else in the Doctrine namespace, are neither part of the ORM, nor the DBAL, it is a plain PHP class that has no outside dependencies apart from dependencies on PHP itself (and the SPL). Therefore using this class in your model and elsewhere does not introduce a coupling to the ORM.

Initializing Collections

You should always initialize the collections of your `@OneToMany` and `@ManyToMany` associations in the constructor of your entities:

```
<?php
use Doctrine\Common\Collections\Collection;
use Doctrine\Common\Collections\ArrayCollection;

/** @Entity */
class User
{
    /**
     * Many Users have Many Groups.
     * @var Collection
     * @ManyToMany(targetEntity="Group")
     */
    private $groups;

    public function __construct()
    {
        $this->groups = new ArrayCollection();
    }

    public function getGroups()
    {
        return $this->groups;
    }
}
```

The following code will then work even if the Entity hasn't been associated with an EntityManager yet:

<?php
$group = new Group();
$user = new User();
$user→getGroups()→add($group);

Inheritance Mapping

Mapped Superclasses

A mapped superclass is an abstract or concrete class that provides persistent entity state and mapping information for its subclasses, but which is not itself an entity. Typically, the purpose of such a mapped superclass is to define state and mapping information that is common to multiple entity classes.

Mapped superclasses, just as regular, non-mapped classes, can appear in the middle of an otherwise mapped inheritance hierarchy (through Single Table Inheritance or Class Table Inheritance).

A mapped superclass cannot be an entity, it is not query-able and persistent relationships defined by a mapped superclass must be unidirectional (with an owning side only). This means that One-To-Many associations are not possible on a mapped superclass at all. Furthermore Many-To-Many associations are only possible if the mapped superclass is only used in exactly one entity at the moment. For further support of inheritance, the single or joined table inheritance features have to be used.

Example:

```
<?php
/** @MappedSuperclass */
class MappedSuperclassBase
{
    /** @Column(type="integer") */
    protected $mapped1;
    /** @Column(type="string") */
    protected $mapped2;
    /**
     * @OneToOne(targetEntity="MappedSuperclassRelated1")
     * @JoinColumn(name="related1_id", referencedColumnName="id")
     */
    protected $mappedRelated1;

    // ... more fields and methods
}

/** @Entity */
class EntitySubClass extends MappedSuperclassBase
{
    /** @Id @Column(type="integer") */
    private $id;
    /** @Column(type="string") */
    private $name;

    // ... more fields and methods
}
```

The DDL for the corresponding database schema would look something like this (this is for SQLite):

CREATE TABLE EntitySubClass (mapped1 INTEGER NOT NULL, mapped2 TEXT NOT NULL, id INTEGER NOT NULL, name TEXT NOT NULL, related1_id INTEGER DEFAULT NULL, PRIMARY KEY(id))

As you can see from this DDL snippet, there is only a single table for the entity subclass. All the mappings from the mapped superclass were inherited to the subclass as if they had been defined on that class directly.

Single Table Inheritance

Single Table Inheritance is an inheritance mapping strategy where all classes of a hierarchy are mapped to a single database table. In order to distinguish which row represents which type in the hierarchy a so-called discriminator column is used.

Example:

- *PHP*

```
<?php
namespace MyProject\Model;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/**
 * @Entity
 */
class Employee extends Person
{
    // ...
}
```

Things to note:

- The @InheritanceType and @DiscriminatorColumn must be specified on the topmost class that is part of the mapped entity hierarchy.
- The @DiscriminatorMap specifies which values of the discriminator column identify a row as being of a certain type. In the case above a value of "person" identifies a row as being of type `Person` and "employee" identifies a row as being of type `Employee`.
- All entity classes that is part of the mapped entity hierarchy (including the topmost class) should be specified in the @DiscriminatorMap. In the case above Person class included.
- The names of the classes in the discriminator map do not need to be fully qualified if the classes are contained in the same namespace as the entity class on which the discriminator map is applied.
- If no discriminator map is provided, then the map is generated automatically. The automatically generated discriminator map
contains the lowercase short name of each class as key.

Design-time considerations

This mapping approach works well when the type hierarchy is fairly simple and stable. Adding a new type to the hierarchy and adding fields to existing supertypes simply involves adding new columns to the table, though in large deployments this may have an adverse impact on the index and column layout inside the database.

Performance impact

This strategy is very efficient for querying across all types in the hierarchy or for specific types. No table joins are required, only a WHERE clause listing the type identifiers. In particular, relationships involving types that employ this mapping strategy are very performing.

There is a general performance consideration with Single Table Inheritance: If the target-entity of a many-to-one or one-to-one
association is an STI entity, it is preferable for performance reasons that it
be a leaf entity in the inheritance hierarchy, (ie. have no subclasses).
Otherwise Doctrine *CANNOT* create proxy instances of this entity and will *ALWAYS* load the entity eagerly.

SQL Schema considerations

For Single-Table-Inheritance to work in scenarios where you are using either a legacy database schema or a self-written database schema you have to make sure that all columns that are not in the root entity but in any of the different sub-entities has to allow null values. Columns that have NOT NULL constraints have to be on the root entity of the single-table inheritance hierarchy.

Class Table Inheritance

Class Table Inheritance is an inheritance mapping strategy where each class in a hierarchy is mapped to several tables: its own table and the tables of all parent classes. The table of a child class is linked to the table of a parent class through a foreign key constraint. Doctrine ORM implements this strategy through the use of a discriminator column in the topmost table of the hierarchy because this is the easiest way to achieve polymorphic queries with Class Table Inheritance.

Example:

```
<?php
namespace MyProject\Model;

/**
 * @Entity
 * @InheritanceType("JOINED")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/** @Entity */
class Employee extends Person
{
    // ...
}
```

Things to note:

- The @InheritanceType, @DiscriminatorColumn and @DiscriminatorMap must be specified on the topmost class that is part of the mapped entity hierarchy.

- The @DiscriminatorMap specifies which values of the discriminator column identify a row as being of which type. In the case above a value of "person" identifies a row as being of type `Person` and "employee" identifies a row as being of type `Employee`.
- The names of the classes in the discriminator map do not need to be fully qualified if the classes are contained in the same namespace as the entity class on which the discriminator map is applied.
- If no discriminator map is provided, then the map is generated automatically. The automatically generated discriminator map
contains the lowercase short name of each class as key.

When you do not use the SchemaTool to generate the required SQL you should know that deleting a class table inheritance makes use of the foreign key property `ON DELETE CASCADE` in all database implementations. A failure to implement this yourself will lead to dead rows in the database.

Design-time considerations

Introducing a new type to the hierarchy, at any level, simply involves interjecting a new table into the schema. Subtypes of that type will automatically join with that new type at runtime. Similarly, modifying any entity type in the hierarchy by adding, modifying or removing fields affects only the immediate table mapped to that type. This mapping strategy provides the greatest flexibility at design time, since changes to any type are always limited to that type's dedicated table.

Performance impact

This strategy inherently requires multiple JOIN operations to perform just about any query which can have a negative impact on performance, especially with large tables and/or large hierarchies. When partial objects are allowed, either globally or on the specific query, then querying for any type will not cause the tables of subtypes to be OUTER JOINed which can increase performance but the resulting partial objects will not fully load themselves on access of any subtype fields, so accessing fields of subtypes after such a query is not safe.

There is a general performance consideration with Class Table Inheritance: If the target-entity of a many-to-one or one-to-one
association is a CTI entity, it is preferable for performance reasons that it
be a leaf entity in the inheritance hierarchy, (ie. have no subclasses).
Otherwise Doctrine *CANNOT* create proxy instances of this entity and will *ALWAYS* load the entity eagerly.

SQL Schema considerations

For each entity in the Class-Table Inheritance hierarchy all the mapped fields have to be columns on the table of this entity. Additionally each child table has to have an id column that matches the id column definition on the root table (except for any sequence or auto-increment details). Furthermore each child table has to have a foreign key pointing from the id column to the root table id column and cascading on delete.

Overrides

Used to override a mapping for an entity field or relationship. Can only be applied to an entity that extends a mapped superclass or uses a trait to override a relationship or field mapping defined by the mapped superclass or trait.

It is not possible to override attributes or associations in entity to entity inheritance scenarios, because this can cause unforseen edge case behavior and increases complexity in ORM internal classes.

Association Override

Override a mapping for an entity relationship.

Could be used by an entity that extends a mapped superclass to override a relationship mapping defined by the mapped superclass.

Example:

- *PHP*

```
<?php
// user mapping
namespace MyProject\Model;
/**
 * @MappedSuperclass
 */
class User
{
    //other fields mapping

    /**
     * @ManyToMany(targetEntity="Group", inversedBy="users")
     * @JoinTable(name="users_groups",
     *  joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *  inverseJoinColumns={@JoinColumn(name="group_id",
referencedColumnName="id")}
     * )
     */
    protected $groups;

    /**
     * @ManyToOne(targetEntity="Address")
     * @JoinColumn(name="address_id", referencedColumnName="id")
     */
    protected $address;
}

// admin mapping
namespace MyProject\Model;
/**
 * @Entity
 * @AssociationOverrides({
 *      @AssociationOverride(name="groups",
 *          joinTable=@JoinTable(
 *              name="users_admingroups",
 *              joinColumns=@JoinColumn(name="adminuser_id"),
 *              inverseJoinColumns=@JoinColumn(name="admingroup_id")
 *          )
 *      ),
 *      @AssociationOverride(name="address",
 *          joinColumns=@JoinColumn(
 *              name="adminaddress_id", referencedColumnName="id"
 *          )
 *      )
 * })
 */
class Admin extends User
{
}
```

Things to note:

- The "association override" specifies the overrides base on the property name.
- This feature is available for all kind of associations. (OneToOne, OneToMany, ManyToOne, ManyToMany)
- The association type *CANNOT* be changed.
- The override could redefine the joinTables or joinColumns depending on the association type.
- The override could redefine inversedBy to reference more than one extended entity.
- The override could redefine fetch to modify the fetch strategy of the extended entity.

Attribute Override

Override the mapping of a field.

Could be used by an entity that extends a mapped superclass to override a field mapping defined by the mapped superclass.

- *PHP*

```
<?php
// user mapping
namespace MyProject\Model;
/**
 * @MappedSuperclass
 */
class User
{
    /** @Id @GeneratedValue @Column(type="integer", name="user_id", length=150)
*/
    protected $id;

    /** @Column(name="user_name", nullable=true, unique=false, length=250) */
    protected $name;

    // other fields mapping
}

// guest mapping
namespace MyProject\Model;
/**
 * @Entity
 * @AttributeOverrides({
 *      @AttributeOverride(name="id",
 *          column=@Column(
 *              name     = "guest_id",
 *              type     = "integer",
 *              length   = 140
 *          )
 *      ),
 *      @AttributeOverride(name="name",
 *          column=@Column(
 *              name     = "guest_name",
 *              nullable = false,
 *              unique   = true,
 *              length   = 240
 *          )
 *      )
 * })
 */
class Guest extends User
{
}
```

Things to note:

- The "attribute override" specifies the overrides base on the property name.
- The column type *CANNOT* be changed. If the column type is not equal you get a `MappingException`
- The override can redefine all the attributes except the type.

Query the Type

It may happen that the entities of a special type should be queried. Because there is no direct access to the discriminator column, Doctrine provides the `INSTANCE OF` construct.

The following example shows how to use `INSTANCE OF`. There is a three level hierarchy with a base entity `NaturalPerson` which is extended by `Staff` which in turn is extended by `Technician`.

Querying for the staffs without getting any technicians can be achieved by this DQL:

```
<?php
$query = $em->createQuery("SELECT staff FROM MyProject\Model\Staff staff WHERE
staff NOT INSTANCE OF MyProject\Model\Technician");
$staffs = $query->getResult();
```

Working with Objects

In this chapter we will help you understand the `EntityManager` and the `UnitOfWork`. A Unit of Work is similar to an object-level transaction. A new Unit of Work is implicitly started when an EntityManager is initially created or after `EntityManager#flush()` has been invoked. A Unit of Work is committed (and a new one started) by invoking `EntityManager#flush()`.

A Unit of Work can be manually closed by calling EntityManager#close(). Any changes to objects within this Unit of Work that have not yet been persisted are lost.

It is very important to understand that only `EntityManager#flush()` ever causes write operations against the database to be executed. Any other methods such as `EntityManager#persist($entity)` or `EntityManager#remove($entity)` only notify the UnitOfWork to perform these operations during flush.

Not calling `EntityManager#flush()` will lead to all changes during that request being lost.

Doctrine does NEVER touch the public API of methods in your entity classes (like getters and setters) nor the constructor method. Instead, it uses reflection to get/set data from/to your entity objects. When Doctrine fetches data from DB and saves it back, any code put in your get/set methods won't be implicitly taken into account.

Entities and the Identity Map

Entities are objects with identity. Their identity has a conceptual meaning inside your domain. In a CMS application each article has a unique id. You can uniquely identify each article by that id.

Take the following example, where you find an article with the headline Hello World with the ID 1234:

```
<?php
$article = $entityManager->find('CMS\Article', 1234);
$article->setHeadline('Hello World dude!');

$article2 = $entityManager->find('CMS\Article', 1234);
echo $article2->getHeadline();
```

In this case the Article is accessed from the entity manager twice, but modified in between. Doctrine ORM realizes this and will only ever give you access to one instance of the Article with ID 1234, no matter how often do you retrieve it from the EntityManager and even no matter what kind of Query method you are using (find, Repository Finder or DQL). This is called Identity Map pattern, which means Doctrine keeps a map of each entity and ids that have been retrieved per PHP request and keeps returning you the same instances.

In the previous example the echo prints Hello World dude! to the screen. You can even verify that `$article` and `$article2` are indeed pointing to the same instance by running the following code:

```
<?php
if ($article === $article2) {
    echo "Yes we are the same!";
}
```

Sometimes you want to clear the identity map of an EntityManager to start over. We use this regularly in our unit-tests to enforce loading objects from the database again instead of serving them from the identity map. You can call `EntityManager#clear()` to achieve this result.

Entity Object Graph Traversal

Although Doctrine allows for a complete separation of your domain model (Entity classes) there will never be a situation where objects are missing when traversing associations. You can walk all the associations inside your entity models as deep as you want.

Take the following example of a single `Article` entity fetched from newly opened EntityManager.

```
<?php
/** @Entity */
```

```
class Article
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;

    /** @Column(type="string") */
    private $headline;

    /** @ManyToOne(targetEntity="User") */
    private $author;

    /** @OneToMany(targetEntity="Comment", mappedBy="article") */
    private $comments;

    public function __construct()
    {
        $this->comments = new ArrayCollection();
    }

    public function getAuthor() { return $this->author; }
    public function getComments() { return $this->comments; }
}

$article = $em->find('Article', 1);
```

This code only retrieves the `Article` instance with id 1 executing a single SELECT statement against the articles table in the database. You can still access the associated properties author and comments and the associated objects they contain.

This works by utilizing the lazy loading pattern. Instead of passing you back a real Author instance and a collection of comments Doctrine will create proxy instances for you. Only if you access these proxies for the first time they will go through the EntityManager and load their state from the database.

This lazy-loading process happens behind the scenes, hidden from your code. See the following code:

```
<?php
$article = $em->find('Article', 1);

// accessing a method of the user instance triggers the lazy-load
echo "Author: " . $article->getAuthor()->getName() . "\n";

// Lazy Loading Proxies pass instanceof tests:
if ($article->getAuthor() instanceof User) {
    // a User Proxy is a generated "UserProxy" class
}

// accessing the comments as an iterator triggers the lazy-load
// retrieving ALL the comments of this article from the database
// using a single SELECT statement
foreach ($article->getComments() as $comment) {
    echo $comment->getText() . "\n\n";
}

// Article::$comments passes instanceof tests for the Collection interface
// But it will NOT pass for the ArrayCollection interface
if ($article->getComments() instanceof \Doctrine\Common\Collections\Collection)
{
    echo "This will always be true!";
}
```

A slice of the generated proxy classes code looks like the following piece of code. A real proxy class override ALL public methods along the lines of the `getName()` method shown below:

```
<?php
class UserProxy extends User implements Proxy
{
    private function _load()
    {
        // lazy loading code
    }

    public function getName()
    {
        $this->_load();
        return parent::getName();
    }
    // .. other public methods of User
}
```

Traversing the object graph for parts that are lazy-loaded will easily trigger lots of SQL queries and will perform badly if used to heavily. Make sure to use DQL to fetch-join all the parts of the object-graph that you need as efficiently as possible.

Persisting entities

An entity can be made persistent by passing it to the `EntityManager#persist($entity)` method. By applying the persist operation on some entity, that entity becomes MANAGED, which means that its persistence is from now on managed by an EntityManager. As a result the persistent state of such an entity will subsequently be properly synchronized with the database when `EntityManager#flush()` is invoked.

Invoking the `persist` method on an entity does NOT cause an immediate SQL INSERT to be issued on the database. Doctrine applies a strategy called transactional write-behind, which means that it will delay most SQL commands until `EntityManager#flush()` is invoked which will then issue all necessary SQL statements to synchronize your objects with the database in the most efficient way and a single, short transaction, taking care of maintaining referential integrity.

Example:

```
<?php
$user = new User;
$user->setName('Mr.Right');
$em->persist($user);
$em->flush();
```

Generated entity identifiers / primary keys are guaranteed to be available after the next successful flush operation that involves the entity in question. You can not rely on a generated identifier to be available directly after invoking `persist`. The inverse is also true. You can not rely on a generated identifier being not available after a failed flush operation.

The semantics of the persist operation, applied on an entity X, are as follows:

- If X is a new entity, it becomes managed. The entity X will be entered into the database as a result of the flush operation.
- If X is a preexisting managed entity, it is ignored by the persist operation. However, the persist operation is cascaded to entities referenced by X, if the relationships from X to these other entities are mapped with cascade=PERSIST or cascade=ALL (see "Working with Associations").
- If X is a removed entity, it becomes managed.
- If X is a detached entity, an exception will be thrown on flush.

Do not pass detached entities to the persist operation. The persist operation always considers entities that are not yet known to the `EntityManager` as new entities (refer to the `STATE_NEW` constant inside the `UnitOfWork`).

Removing entities

An entity can be removed from persistent storage by passing it to the `EntityManager#remove($entity)` method. By applying the `remove` operation on some entity, that entity becomes REMOVED, which means that its persistent state will be deleted once `EntityManager#flush()` is invoked.

Just like `persist`, invoking `remove` on an entity does NOT cause an immediate SQL DELETE to be issued on the database. The entity will be deleted on the next invocation of `EntityManager#flush()` that involves that entity. This means that entities scheduled for removal can still be queried for and appear in query and collection results. See the section on Database and UnitOfWork Out-Of-Sync for more information.

Example:

```
<?php
$em->remove($user);
$em->flush();
```

The semantics of the remove operation, applied to an entity X are as follows:

- If X is a new entity, it is ignored by the remove operation. However, the remove operation is cascaded to entities referenced by X, if the relationship from X to these other entities is mapped with cascade=REMOVE or cascade=ALL (see "Working with Associations").
- If X is a managed entity, the remove operation causes it to become removed. The remove operation is cascaded to entities referenced by X, if the relationships from X to these other entities is mapped with cascade=REMOVE or cascade=ALL (see "Working with Associations").
- If X is a detached entity, an InvalidArgumentException will be thrown.
- If X is a removed entity, it is ignored by the remove operation.
- A removed entity X will be removed from the database as a result of the flush operation.

After an entity has been removed its in-memory state is the same as before the removal, except for generated identifiers.

Removing an entity will also automatically delete any existing records in many-to-many join tables that link this entity. The action taken depends on the value of the `@joinColumn` mapping attribute onDelete. Either Doctrine issues a dedicated `DELETE` statement for records of each join table or it depends on the foreign key semantics of onDelete=CASCADE.

Deleting an object with all its associated objects can be achieved in multiple ways with very different performance impacts.

1. If an association is marked as `CASCADE=REMOVE` Doctrine ORM will fetch this association. If its a Single association it will pass this entity to `EntityManager#remove()`. If the association is a collection, Doctrine will loop over all its elements and pass them to`EntityManager#remove()`. In both cases the cascade remove semantics are applied recursively. For large object graphs this removal strategy can be very costly.
2. Using a DQL `DELETE` statement allows you to delete multiple entities of a type with a single command and without hydrating these entities. This can be very efficient to delete large object graphs from the database.
3. Using foreign key semantics `onDelete="CASCADE"` can force the database to remove all associated objects internally. This strategy is a bit tricky to get right but can be very powerful and fast. You should be aware however that using strategy 1 (`CASCADE=REMOVE`) completely by-passes any foreign key `onDelete=CASCADE` option, because Doctrine will fetch and remove all associated entities explicitly nevertheless.

Calling `remove` on an entity will remove the object from the identiy map and therefore detach it. Querying the same entity again, for example
via a lazy loaded relation, will return a new object.

Detaching entities

An entity is detached from an EntityManager and thus no longer managed by invoking the `EntityManager#detach($entity)` method on it or by cascading the detach operation to it. Changes made to the detached entity, if any (including removal of the entity), will not be synchronized to the database after the entity has been detached.

Doctrine will not hold on to any references to a detached entity.

Example:

```
<?php
$em->detach($entity);
```

The semantics of the detach operation, applied to an entity X are as follows:

- If X is a managed entity, the detach operation causes it to become detached. The detach operation is cascaded to entities referenced by X, if the relationships from X to these other entities is mapped with cascade=DETACH or cascade=ALL (see "Working with Associations"). Entities which previously referenced X will continue to reference X.
- If X is a new or detached entity, it is ignored by the detach operation.
- If X is a removed entity, the detach operation is cascaded to entities referenced by X, if the relationships from X to these other entities is mapped with cascade=DETACH or

cascade=ALL (see "Working with Associations"). Entities which previously referenced X will continue to reference X.

There are several situations in which an entity is detached automatically without invoking the `detach` method:

- When `EntityManager#clear()` is invoked, all entities that are currently managed by the EntityManager instance become detached.
- When serializing an entity. The entity retrieved upon subsequent unserialization will be detached (This is the case for all entities that are serialized and stored in some cache, i.e. when using the Query Result Cache).

The `detach` operation is usually not as frequently needed and used as `persist` and `remove`.

Merging entities

Merging entities refers to the merging of (usually detached) entities into the context of an EntityManager so that they become managed again. To merge the state of an entity into an EntityManager use the `EntityManager#merge($entity)` method. The state of the passed entity will be merged into a managed copy of this entity and this copy will subsequently be returned.

Example:

```
<?php
$detachedEntity = unserialize($serializedEntity); // some detached entity
$entity = $em->merge($detachedEntity);
// $entity now refers to the fully managed copy returned by the merge operation.
// The EntityManager $em now manages the persistence of $entity as usual.
```

When you want to serialize/unserialize entities you have to make all entity properties protected, never private. The reason for this is, if you serialize a class that was a proxy instance before, the private variables won't be serialized and a PHP Notice is thrown.

The semantics of the merge operation, applied to an entity X, are as follows:

- If X is a detached entity, the state of X is copied onto a pre-existing managed entity instance X' of the same identity.
- If X is a new entity instance, a new managed copy X' will be created and the state of X is copied onto this managed instance.
- If X is a removed entity instance, an InvalidArgumentException will be thrown.
- If X is a managed entity, it is ignored by the merge operation, however, the merge operation is cascaded to entities referenced by relationships from X if these relationships have been mapped with the cascade element value MERGE or ALL (see "Working with Associations").
- For all entities Y referenced by relationships from X having the cascade element value MERGE or ALL, Y is merged recursively as Y'. For all such Y referenced by X, X' is set to reference Y'. (Note that if X is managed then X is the same object as X'.)
- If X is an entity merged to X', with a reference to another entity Y, where cascade=MERGE or cascade=ALL is not specified, then navigation of the same association from X' yields a reference to a managed object Y' with the same persistent identity as Y.

The `merge` operation will throw an `OptimisticLockException` if the entity being merged uses optimistic locking through a version field and the versions of the entity being merged and the managed copy don't match. This usually means that the entity has been modified while being detached.

The `merge` operation is usually not as frequently needed and used as `persist` and `remove`. The most common scenario for the `merge` operation is to reattach entities to an EntityManager that come from some cache (and are therefore detached) and you want to modify and persist such an entity.

If you need to perform multiple merges of entities that share certain subparts of their object-graphs and cascade merge, then you have to call `EntityManager#clear()` between the successive calls to `EntityManager#merge()`. Otherwise you might end up with multiple copies of the same object in the database, however with different ids.

If you load some detached entities from a cache and you do not need to persist or delete them or otherwise make use of them without the need for persistence services there is no need to use `merge`. I.e. you can simply pass detached objects from a cache directly to the view.

Synchronization with the Database

The state of persistent entities is synchronized with the database on flush of an `EntityManager` which commits the underlying `UnitOfWork`. The synchronization involves writing any updates to persistent entities and their relationships to the database. Thereby bidirectional relationships are persisted based on the references held by the owning side of the relationship as explained in the Association Mapping chapter.

When `EntityManager#flush()` is called, Doctrine inspects all managed, new and removed entities and will perform the following operations.

Effects of Database and UnitOfWork being Out-Of-Sync

As soon as you begin to change the state of entities, call persist or remove the contents of the UnitOfWork and the database will drive out of sync. They can only be synchronized by calling `EntityManager#flush()`. This section describes the effects of database and UnitOfWork being out of sync.

- Entities that are scheduled for removal can still be queried from the database. They are returned from DQL and Repository queries and are visible in collections.
- Entities that are passed to `EntityManager#persist` do not turn up in query results.
- Entities that have changed will not be overwritten with the state from the database. This is because the identity map will detect the construction of an already existing entity and assumes its the most up to date version.

`EntityManager#flush()` is never called implicitly by Doctrine. You always have to trigger it manually.

Synchronizing New and Managed Entities

The flush operation applies to a managed entity with the following semantics:

- The entity itself is synchronized to the database using a SQL UPDATE statement, only if at least one persistent field has changed.
- No SQL updates are executed if the entity did not change.

The flush operation applies to a new entity with the following semantics:

- The entity itself is synchronized to the database using a SQL INSERT statement.

For all (initialized) relationships of the new or managed entity the following semantics apply to each associated entity X:

- If X is new and persist operations are configured to cascade on the relationship, X will be persisted.
- If X is new and no persist operations are configured to cascade on the relationship, an exception will be thrown as this indicates a programming error.
- If X is removed and persist operations are configured to cascade on the relationship, an exception will be thrown as this indicates a programming error (X would be re-persisted by the cascade).
- If X is detached and persist operations are configured to cascade on the relationship, an exception will be thrown (This is semantically the same as passing X to persist()).

Synchronizing Removed Entities

The flush operation applies to a removed entity by deleting its persistent state from the database. No cascade options are relevant for removed entities on flush, the cascade remove option is already executed during `EntityManager#remove($entity)`.

The size of a Unit of Work

The size of a Unit of Work mainly refers to the number of managed entities at a particular point in time.

The cost of flushing

How costly a flush operation is, mainly depends on two factors:

- The size of the EntityManager's current UnitOfWork.
- The configured change tracking policies

You can get the size of a UnitOfWork as follows:

```
<?php
$uowSize = $em->getUnitOfWork()->size();
```

The size represents the number of managed entities in the Unit of Work. This size affects the performance of flush() operations due to change tracking (see Change Tracking Policies) and, of course, memory consumption, so you may want to check it from time to time during development.

Do not invoke `flush` after every change to an entity or every single invocation of persist/remove/merge/... This is an anti-pattern and unnecessarily reduces the performance of your application. Instead, form units of work that operate on your objects and call `flush` when you are done. While serving a single HTTP request there should be usually no need for invoking `flush` more than 0-2 times.

Direct access to a Unit of Work

You can get direct access to the Unit of Work by calling `EntityManager#getUnitOfWork()`. This will return the UnitOfWork instance the EntityManager is currently using.

```
<?php
$uow = $em->getUnitOfWork();
```

Directly manipulating a UnitOfWork is not recommended. When working directly with the UnitOfWork API, respect methods marked as INTERNAL by not using them and carefully read the API documentation.

Entity State

As outlined in the architecture overview an entity can be in one of four possible states: NEW, MANAGED, REMOVED, DETACHED. If you explicitly need to find out what the current state of an entity is in the context of a certain `EntityManager` you can ask the underlying `UnitOfWork`:

```
<?php
switch ($em->getUnitOfWork()->getEntityState($entity)) {
    case UnitOfWork::STATE_MANAGED:
        ...
    case UnitOfWork::STATE_REMOVED:
        ...
    case UnitOfWork::STATE_DETACHED:
        ...
    case UnitOfWork::STATE_NEW:
        ...
}
```

An entity is in MANAGED state if it is associated with an `EntityManager` and it is not REMOVED.

An entity is in REMOVED state after it has been passed to `EntityManager#remove()` until the next flush operation of the same EntityManager. A REMOVED entity is still associated with an `EntityManager` until the next flush operation.

An entity is in DETACHED state if it has persistent state and identity but is currently not associated with an `EntityManager`.

An entity is in NEW state if has no persistent state and identity and is not associated with an `EntityManager` (for example those just created via the new operator).

Querying

Doctrine ORM provides the following ways, in increasing level of power and flexibility, to query for persistent objects. You should always start with the simplest one that suits your needs.

By Primary Key

The most basic way to query for a persistent object is by its identifier / primary key using the `EntityManager#find($entityName, $id)` method. Here is an example:

```
<?php
// $em instanceof EntityManager
$user = $em->find('MyProject\Domain\User', $id);
```

The return value is either the found entity instance or null if no instance could be found with the given identifier.

Essentially, `EntityManager#find()` is just a shortcut for the following:

```
<?php
// $em instanceof EntityManager
$user = $em→getRepository('MyProject\Domain\User')→find($id);
```

`EntityManager#getRepository($entityName)` returns a repository object which provides many ways to retrieve entities of the specified type. By default, the repository instance is of type `Doctrine\ORM\EntityRepository`. You can also use custom repository classes as shown later.

By Simple Conditions

To query for one or more entities based on several conditions that form a logical conjunction, use the `findBy` and `findOneBy` methods on a repository as follows:

```
<?php
// $em instanceof EntityManager

// All users that are 20 years old
$users = $em->getRepository('MyProject\Domain\User')->findBy(array('age' =>
20));

// All users that are 20 years old and have a surname of 'Miller'
$users = $em->getRepository('MyProject\Domain\User')->findBy(array('age' => 20,
'surname' => 'Miller'));

// A single user by its nickname
$user = $em->getRepository('MyProject\Domain\User')->findOneBy(array('nickname'
=> 'romanb'));
```

You can also load by owning side associations through the repository:

```
<?php
$number = $em->find('MyProject\Domain\Phonenumber', 1234);
$user = $em->getRepository('MyProject\Domain\User')->findOneBy(array('phone' =>
$number->getId()));
```

The `EntityRepository#findBy()` method additionally accepts orderings, limit and offset as second to fourth parameters:

```
<?php
$tenUsers = $em->getRepository('MyProject\Domain\User')->findBy(array('age' =>
20), array('name' => 'ASC'), 10, 0);
```

If you pass an array of values Doctrine will convert the query into a WHERE field IN (..) query automatically:

```
<?php
$users = $em->getRepository('MyProject\Domain\User')->findBy(array('age' =>
array(20, 30, 40)));
// translates roughly to: SELECT * FROM users WHERE age IN (20, 30, 40)
```

An EntityRepository also provides a mechanism for more concise calls through its use of `__call`. Thus, the following two examples are equivalent:

```
<?php
// A single user by its nickname
$user = $em->getRepository('MyProject\Domain\User')->findOneBy(array('nickname'
=> 'romanb'));

// A single user by its nickname (__call magic)
$user = $em->getRepository('MyProject\Domain\User')-
>findOneByNickname('romanb');
```

Additionally, you can just count the result of the provided conditions when you don't really need the data:

```
<?php
// Check there is no user with nickname
$availableNickname = 0 === $em->getRepository('MyProject\Domain\User')-
>count(['nickname' => 'nonexistent']);
```

By Criteria

The Repository implement the `Doctrine\Common\Collections\Selectable` interface. That means you can build `Doctrine\Common\Collections\Criteria` and pass them to the `matching($criteria)` method.

See section `Filtering collections` of chapter Working with Associations

By Eager Loading

Whenever you query for an entity that has persistent associations and these associations are mapped as EAGER, they will automatically be loaded together with the entity being queried and is thus immediately available to your application.

By Lazy Loading

Whenever you have a managed entity instance at hand, you can traverse and use any associations of that entity that are configured LAZY as if they were in-memory already. Doctrine will automatically load the associated objects on demand through the concept of lazy-loading.

By DQL

The most powerful and flexible method to query for persistent objects is the Doctrine Query Language, an object query language. DQL enables you to query for persistent objects in the language of objects. DQL understands classes, fields, inheritance and associations. DQL is syntactically very similar to the familiar SQL but *it is not SQL*.

A DQL query is represented by an instance of the `Doctrine\ORM\Query` class. You create a query using `EntityManager#createQuery($dql)`. Here is a simple example:

```
<?php
// $em instanceof EntityManager

// All users with an age between 20 and 30 (inclusive).
$q = $em->createQuery("select u from MyDomain\Model\User u where u.age >= 20 and
u.age <= 30");
$users = $q->getResult();
```

Note that this query contains no knowledge about the relational schema, only about the object model. DQL supports positional as well as named parameters, many functions, (fetch) joins, aggregates, subqueries and much more. Detailed information about DQL and its syntax as well as the Doctrine class can be found in the dedicated chapter. For programmatically building up queries based on conditions that are only known at runtime, Doctrine provides the special `Doctrine\ORM\QueryBuilder` class. While this a powerful tool, it also brings more complexity to your code compared to plain DQL, so you should only use it when you need it. More information on constructing queries with a QueryBuilder can be found in Query Builder chapter.

By Native Queries

As an alternative to DQL or as a fallback for special SQL statements native queries can be used. Native queries are built by using a hand-crafted SQL query and a ResultSetMapping that describes how the SQL result set should be transformed by Doctrine. More information about native queries can be found in the dedicated chapter.

Custom Repositories

By default the EntityManager returns a default implementation of `Doctrine\ORM\EntityRepository` when you call `EntityManager#getRepository($entityClass)`. You can overwrite this behaviour by specifying the class name of your own Entity Repository in the Annotation, XML or YAML metadata. In large applications that require lots of specialized DQL queries using a custom repository is one recommended way of grouping these queries in a central location.

```
<?php
namespace MyDomain\Model;

use Doctrine\ORM\EntityRepository;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="MyDomain\Model\UserRepository")
 */
class User
{
}

class UserRepository extends EntityRepository
{
    public function getAllAdminUsers()
    {
        return $this->_em->createQuery('SELECT u FROM MyDomain\Model\User u
WHERE u.status = "admin"')
                         ->getResult();
    }
}
```

You can access your repository now by calling:

```
<?php
// $em instanceof EntityManager
$admins = $em->getRepository('MyDomain\Model\User')->getAllAdminUsers();
```

Working with Associations

Associations between entities are represented just like in regular object-oriented PHP code using references to other objects or collections of objects.

Changes to associations in your code are not synchronized to the database directly, only when calling `EntityManager#flush()`.

There are other concepts you should know about when working with associations in Doctrine:

- If an entity is removed from a collection, the association is removed, not the entity itself. A collection of entities always only represents the association to the containing entities, not the entity itself.
- When a bidirectional association is updated, Doctrine only checks on one of both sides for these changes. This is called the owning side of the association.
- A property with a reference to many entities has to be instances of the `Doctrine\Common\Collections\Collection` interface.

Association Example Entities

We will use a simple comment system with Users and Comments as entities to show examples of association management. See the PHP docblocks of each association in the following example for information about its type and if it's the owning or inverse side.

```
<?php
/** @Entity */
class User
{
    /** @Id @GeneratedValue @Column(type="string") */
    private $id;

    /**
     * Bidirectional - Many users have Many favorite comments (OWNING SIDE)
     *
     * @ManyToMany(targetEntity="Comment", inversedBy="userFavorites")
     * @JoinTable(name="user_favorite_comments")
     */
    private $favorites;

    /**
     * Unidirectional - Many users have marked many comments as read
     *
     * @ManyToMany(targetEntity="Comment")
     * @JoinTable(name="user_read_comments")
     */
    private $commentsRead;

    /**
     * Bidirectional - One-To-Many (INVERSE SIDE)
     *
     * @OneToMany(targetEntity="Comment", mappedBy="author")
     */
    private $commentsAuthored;

    /**
     * Unidirectional - Many-To-One
     *
     * @ManyToOne(targetEntity="Comment")
     */
    private $firstComment;
}

/** @Entity */
class Comment
{
    /** @Id @GeneratedValue @Column(type="string") */
    private $id;

    /**
     * Bidirectional - Many comments are favorited by many users (INVERSE SIDE)
     *
     * @ManyToMany(targetEntity="User", mappedBy="favorites")
     */
    private $userFavorites;

    /**
     * Bidirectional - Many Comments are authored by one user (OWNING SIDE)
     *
     * @ManyToOne(targetEntity="User", inversedBy="commentsAuthored")
     */
     private $author;
}
```

This two entities generate the following MySQL Schema (Foreign Key definitions omitted):

```
CREATE TABLE User (
    id VARCHAR(255) NOT NULL,
    firstComment_id VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE Comment (
    id VARCHAR(255) NOT NULL,
    author_id VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;

CREATE TABLE user_favorite_comments (
    user_id VARCHAR(255) NOT NULL,
    favorite_comment_id VARCHAR(255) NOT NULL,
    PRIMARY KEY(user_id, favorite_comment_id)
) ENGINE = InnoDB;

CREATE TABLE user_read_comments (
    user_id VARCHAR(255) NOT NULL,
    comment_id VARCHAR(255) NOT NULL,
    PRIMARY KEY(user_id, comment_id)
) ENGINE = InnoDB;
```

Establishing Associations

Establishing an association between two entities is straight-forward. Here are some examples for the unidirectional relations of the `User`:

```
<?php
class User
{
    // ...
    public function getReadComments() {
         return $this->commentsRead;
    }

    public function setFirstComment(Comment $c) {
        $this->firstComment = $c;
    }
}
```

The interaction code would then look like in the following snippet (`$em` here is an instance of the EntityManager):

```
<?php
$user = $em->find('User', $userId);

// unidirectional many to many
$comment = $em->find('Comment', $readCommentId);
$user->getReadComments()->add($comment);

$em->flush();

// unidirectional many to one
$myFirstComment = new Comment();
$user->setFirstComment($myFirstComment);

$em->persist($myFirstComment);
$em->flush();
```

In the case of bi-directional associations you have to update the fields on both sides:

```
<?php
class User
{
    // ..

    public function getAuthoredComments() {
        return $this->commentsAuthored;
    }

    public function getFavoriteComments() {
        return $this->favorites;
    }
}

class Comment
{
    // ...

    public function getUserFavorites() {
        return $this->userFavorites;
    }

    public function setAuthor(User $author = null) {
        $this->author = $author;
    }
}

// Many-to-Many
$user->getFavorites()->add($favoriteComment);
$favoriteComment->getUserFavorites()->add($user);

$em->flush();

// Many-To-One / One-To-Many Bidirectional
$newComment = new Comment();
$user->getAuthoredComments()->add($newComment);
$newComment->setAuthor($user);

$em->persist($newComment);
$em->flush();
```

Notice how always both sides of the bidirectional association are updated. The previous unidirectional associations were simpler to handle.

Removing Associations

Removing an association between two entities is similarly straight-forward. There are two strategies to do so, by key and by element. Here are some examples:

```
<?php
// Remove by Elements
$user->getComments()->removeElement($comment);
$comment->setAuthor(null);

$user->getFavorites()->removeElement($comment);
$comment->getUserFavorites()->removeElement($user);

// Remove by Key
$user->getComments()->remove($ithComment);
$comment->setAuthor(null);
```

You need to call `$em->flush()` to make persist these changes in the database permanently.

Notice how both sides of the bidirectional association are always updated. Unidirectional associations are consequently simpler to handle.

Also note that if you use type-hinting in your methods, you will have to specify a nullable type, i.e. `setAddress(?Address $address)`, otherwise `setAddress(null)` will fail to remove the association. Another way to deal with this is to provide a special method, like `removeAddress()`. This can also provide better encapsulation as it hides the internal meaning of not having an address.

When working with collections, keep in mind that a Collection is essentially an ordered map (just like a PHP array). That is why the `remove` operation accepts an index/key. `removeElement` is a separate method that has O(n) complexity using `array_search`, where n is the size of the map.

> Since Doctrine always only looks at the owning side of a bidirectional association for updates, it is not necessary for write operations that an inverse collection of a bidirectional one-to-many or many-to-many association is updated. This knowledge can often be used to improve performance by avoiding the loading of the inverse collection.

You can also clear the contents of a whole collection using the `Collections::clear()` method. You should be aware that using this method can lead to a straight and optimized database delete or update call during the flush operation that is not aware of entities that have been re-added to the collection.

Say you clear a collection of tags by calling `$post->getTags()->clear();` and then call `$post->getTags()->add($tag)`. This will not recognize the tag having already been added previously and will consequently issue two separate database calls.

Association Management Methods

It is generally a good idea to encapsulate proper association management inside the entity classes. This makes it easier to use the class correctly and can encapsulate details about how the association is maintained.

The following code shows updates to the previous User and Comment example that encapsulate much of the association management code:

```
<?php
class User
{
    //...
    public function markCommentRead(Comment $comment) {
        // Collections implement ArrayAccess
        $this->commentsRead[] = $comment;
    }

    public function addComment(Comment $comment) {
        if (count($this->commentsAuthored) == 0) {
            $this->setFirstComment($comment);
        }
        $this->comments[] = $comment;
        $comment->setAuthor($this);
    }

    private function setFirstComment(Comment $c) {
        $this->firstComment = $c;
    }

    public function addFavorite(Comment $comment) {
        $this->favorites->add($comment);
        $comment->addUserFavorite($this);
    }

    public function removeFavorite(Comment $comment) {
        $this->favorites->removeElement($comment);
        $comment->removeUserFavorite($this);
    }
}

class Comment
{
    // ..

    public function addUserFavorite(User $user) {
        $this->userFavorites[] = $user;
    }

    public function removeUserFavorite(User $user) {
        $this->userFavorites->removeElement($user);
    }
}
```

You will notice that `addUserFavorite` and `removeUserFavorite` do not call `addFavorite` and `removeFavorite`, thus the bidirectional association is strictly-speaking still incomplete. However if you would naively add the `addFavorite` in `addUserFavorite`, you end up with an infinite loop, so more work is needed. As you can see, proper bidirectional association management in plain OOP is a non-trivial task and encapsulating all the details inside the classes can be challenging.

If you want to make sure that your collections are perfectly encapsulated you should not return them from a `getCollectionName()` method directly, but call `$collection->toArray()`. This way a client programmer for the entity cannot circumvent the logic you implement on your entity for association management. For example:

```
<?php
class User {
    public function getReadComments() {
        return $this->commentsRead->toArray();
    }
}
```

This will however always initialize the collection, with all the performance penalties given the size. In some scenarios of large collections it might even be a good idea to completely hide the read access behind methods on the EntityRepository.

There is no single, best way for association management. It greatly depends on the requirements of your concrete domain model as well as your preferences.

Synchronizing Bidirectional Collections

In the case of Many-To-Many associations you as the developer have the responsibility of keeping the collections on the owning and inverse side in sync when you apply changes to them. Doctrine can only guarantee a consistent state for the hydration, not for your client code.

Using the User-Comment entities from above, a very simple example can show the possible caveats you can encounter:

```
<?php
$user->getFavorites()->add($favoriteComment);
```

```
// not calling $favoriteComment->getUserFavorites()->add($user);

$user->getFavorites()->contains($favoriteComment); // TRUE
$favoriteComment->getUserFavorites()->contains($user); // FALSE
```

There are two approaches to handle this problem in your code:

1. Ignore updating the inverse side of bidirectional collections, BUT never read from them in requests that changed their state. In the next request Doctrine hydrates the consistent collection state again.
2. Always keep the bidirectional collections in sync through association management methods. Reads of the Collections directly after changes are consistent then.

Transitive persistence / Cascade Operations

Doctrine ORM provides a mechanism for transitive persistence through cascading of certain operations. Each association to another entity or a collection of entities can be configured to automatically cascade the following operations to the associated entities: `persist`, `remove`, `merge`, `detach`, `refresh` or `all`.

The main use case for `cascade: persist` is to avoid exposing associated entities to your PHP application. Continuing with the User-Comment example of this chapter, this is how the creation of a new user and a new comment might look like in your controller (without `cascade: persist`):

```
<?php
$user = new User();
$myFirstComment = new Comment();
$user->addComment($myFirstComment);

$em->persist($user);
$em->persist($myFirstComment); // required, if `cascade: persist` is not set
$em->flush();
```

Note that the Comment entity is instantiated right here in the controller. To avoid this, `cascade: persist` allows you to hide the Comment entity from the controller, only accessing it through the User entity:

```
<?php
// User entity
class User
{
    private $id;
    private $comments;

    public function __construct()
    {
        $this->id = User::new();
        $this->comments = new ArrayCollection();
    }

    public function comment(string $text, DateTimeInterface $time) : void
    {
        $newComment = Comment::create($text, $time);
        $newComment->setUser($this);
        $this->comments->add($newComment);
    }

    // ...
}
```

If you then set up the cascading to the `User#commentsAuthored` property...

```
<?php
class User
{
    //...
    /**
     * Bidirectional - One-To-Many (INVERSE SIDE)
     *
     * @OneToMany(targetEntity="Comment", mappedBy="author", cascade={"persist",
"remove"})
     */
    private $commentsAuthored;
    //...
}
```

..you can now create a user and an associated comment like this:

```
<?php
$user = new User();
$user->comment('Lorem ipsum', new DateTime());

$em->persist($user);
$em->flush();
```

The idea of `cascade: persist` is not to save you any lines of code in the controller. If you instantiate the comment object in the controller (i.e. don't set up the user entity as shown above), even with `cascade: persist` you still have to call `$myFirstComment->setUser($user);`.

Thanks to `cascade: remove`, you can easily delete a user and all linked comments without having to loop through them:

```
<?php
$user = $em->find('User', $deleteUserId);

$em->remove($user);
$em->flush();
```

Cascade operations are performed in memory. That means collections and related entities are fetched into memory (even if they are marked as lazy) when the cascade operation is about to be performed. This approach allows entity lifecycle events to be performed for each of these operations.

However, pulling object graphs into memory on cascade can cause considerable performance overhead, especially when the cascaded collections are large. Make sure to weigh the benefits and downsides of each cascade operation that you define.

To rely on the database level cascade operations for the delete operation instead, you can configure each join column with the onDelete option.

Even though automatic cascading is convenient, it should be used with care. Do not blindly apply `cascade=all` to all associations as it will unnecessarily degrade the performance of your application. For each cascade operation that gets activated, Doctrine also applies that operation to the association, be it single or collection valued.

Persistence by Reachability: Cascade Persist

There are additional semantics that apply to the Cascade Persist operation. During each `flush()` operation Doctrine detects if there are new entities in any collection and three possible cases can happen:

1. New entities in a collection marked as `cascade: persist` will be directly persisted by Doctrine.
2. New entities in a collection not marked as `cascade: persist` will produce an Exception and rollback the `flush()` operation.
3. Collections without new entities are skipped.

This concept is called Persistence by Reachability: New entities that are found on already managed entities are automatically persisted as long as the association is defined as `cascade: persist`.

Orphan Removal

There is another concept of cascading that is relevant only when removing entities from collections. If an Entity of type `A` contains references to privately owned Entities `B` then if the reference from `A` to `B` is removed the entity `B` should also be removed, because it is not used anymore.

OrphanRemoval works with one-to-one, one-to-many and many-to-many associations.

> When using the `orphanRemoval=true` option Doctrine makes the assumption that the entities are privately owned and will **NOT** be reused by other entities. If you neglect this assumption your entities will get deleted by Doctrine even if you assigned the orphaned entity to another one.

As a better example consider an Addressbook application where you have Contacts, Addresses and StandingData:

```
<?php

namespace Addressbook;

use Doctrine\Common\Collections\ArrayCollection;

/**
 * @Entity
 */
class Contact
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;

    /** @OneToOne(targetEntity="StandingData", orphanRemoval=true) */
    private $standingData;
    /** @OneToMany(targetEntity="Address", mappedBy="contact",
orphanRemoval=true) */
    private $addresses;

    public function __construct()
    {
        $this->addresses = new ArrayCollection();
    }

    public function newStandingData(StandingData $sd)
    {
        $this->standingData = $sd;
    }

    public function removeAddress($pos)
    {
        unset($this->addresses[$pos]);
    }
}
```

Now two examples of what happens when you remove the references:

```
<?php

$contact = $em->find("Addressbook\Contact", $contactId);
$contact->newStandingData(new StandingData("Firstname", "Lastname", "Street"));
$contact->removeAddress(1);

$em->flush();
```

n this case you have not only changed the `Contact` entity itself but
you have also removed the references for standing data and as well as one
address reference. When flush is called not only are the references removed
but both the old standing data and the one address entity are also deleted
from the database.

Filtering Collections

Collections have a filtering API that allows to slice parts of data from a collection. If the collection has not been loaded from the database yet, the filtering API can work on the SQL level to make optimized access to large collections.

```
<?php

use Doctrine\Common\Collections\Criteria;

$group          = $entityManager->find('Group', $groupId);
$userCollection = $group->getUsers();

$criteria = Criteria::create()
    ->where(Criteria::expr()->eq("birthday", "1982-02-17"))
    ->orderBy(array("username" => Criteria::ASC))
    ->setFirstResult(0)
    ->setMaxResults(20)
;

$birthdayUsers = $userCollection->matching($criteria);
```

You can move the access of slices of collections into dedicated methods of an entity. For example `Group#getTodaysBirthdayUsers()`.

The Criteria has a limited matching language that works both on the SQL and on the PHP collection level. This means you can use collection matching interchangeably, independent of in-memory or sql-backed collections.

```
<?php

use Doctrine\Common\Collections;

class Criteria
{
    /**
     * @return Criteria
     */
    static public function create();
    /**
     * @param Expression $where
     * @return Criteria
     */
    public function where(Expression $where);
    /**
     * @param Expression $where
     * @return Criteria
     */
    public function andWhere(Expression $where);
    /**
     * @param Expression $where
     * @return Criteria
     */
    public function orWhere(Expression $where);
    /**
     * @param array $orderings
     * @return Criteria
     */
    public function orderBy(array $orderings);
    /**
     * @param int $firstResult
     * @return Criteria
     */
    public function setFirstResult($firstResult);
    /**
     * @param int $maxResults
     * @return Criteria
     */
    public function setMaxResults($maxResults);
    public function getOrderings();
    public function getWhereExpression();
    public function getFirstResult();
    public function getMaxResults();
}
```

You can build expressions through the ExpressionBuilder. It has the following methods:

- `andX($arg1, $arg2, ...)`
- `orX($arg1, $arg2, ...)`
- `eq($field, $value)`
- `gt($field, $value)`

- `lt($field, $value)`
- `lte($field, $value)`
- `gte($field, $value)`
- `neq($field, $value)`
- `isNull($field)`
- `in($field, array $values)`
- `notIn($field, array $values)`
- `contains($field, $value)`
- `memberOf($value, $field)`
- `startsWith($field, $value)`
- `endsWith($field, $value)`

There is a limitation on the compatibility of Criteria comparisons. You have to use scalar values only as the value in a comparison or the behaviour between different backends is not the same.

Events

Doctrine ORM features a lightweight event system that is part of the Common package. Doctrine uses it to dispatch system events, mainly lifecycle events. You can also use it for your own custom events.

The Event System

The event system is controlled by the `EventManager`. It is the central point of Doctrine's event listener system. Listeners are registered on the manager and events are dispatched through the manager.

```
<?php
$evm = new EventManager();
```

Now we can add some event listeners to the `$evm`. Let's create a `TestEvent` class to play around with.

```
<?php
class TestEvent
{
    const preFoo = 'preFoo';
    const postFoo = 'postFoo';

    private $_evm;

    public $preFooInvoked = false;
    public $postFooInvoked = false;

    public function __construct($evm)
    {
        $evm->addEventListener(array(self::preFoo, self::postFoo), $this);
    }
    public function preFoo(EventArgs $e)
    {
        $this->preFooInvoked = true;
    }

    public function postFoo(EventArgs $e)
    {
        $this->postFooInvoked = true;
    }
}

// Create a new instance
$test = new TestEvent($evm);
```

Events can be dispatched by using the `dispatchEvent()` method.

```
<?php
$evm->dispatchEvent(TestEvent::preFoo);
$evm->dispatchEvent(TestEvent::postFoo);
```

You can easily remove a listener with the `removeEventListener()` method.

```
<?php
$evm->removeEventListener(array(self::preFoo, self::postFoo), $this);
```

The Doctrine ORM event system also has a simple concept of event subscribers. We can define a simple `TestEventSubscriber` class which implements the `\Doctrine\Common\EventSubscriber` interface and implements a `getSubscribedEvents()` method which returns an array of events it should be subscribed to.

```
<?php
class TestEventSubscriber implements \Doctrine\Common\EventSubscriber
{
    public $preFooInvoked = false;

    public function preFoo()
    {
        $this->preFooInvoked = true;
    }

    public function getSubscribedEvents()
    {
        return array(TestEvent::preFoo);
    }
}

$eventSubscriber = new TestEventSubscriber();
$evm->addEventSubscriber($eventSubscriber);
```

The array to return in the `getSubscribedEvents` method is a simple array with the values being the event names. The subscriber must have a method that is named exactly like the event.

Now when you dispatch an event, any event subscribers will be notified for that event.

```
<?php
$evm->dispatchEvent(TestEvent::preFoo);
```

Now you can test the `$eventSubscriber` instance to see if the `preFoo()` method was invoked.

```
<?php
if ($eventSubscriber->preFooInvoked) {
    echo 'pre foo invoked!';
}
```

Naming convention

Events being used with the Doctrine ORM EventManager are best named with camelcase and the value of the corresponding constant should be the name of the constant itself, even with spelling. This has several reasons:

- It is easy to read.
- Simplicity.
- Each method within an EventSubscriber is named after the corresponding constant's value. If the constant's name and value differ it contradicts the intention of using the constant and makes your code harder to maintain.

An example for a correct notation can be found in the example `TestEvent` above.

Lifecycle Events

The `EntityManager` and `UnitOfWork` classes trigger a bunch of events during the life-time of their registered entities.

- `preRemove` - The `preRemove` event occurs for a given entity before the respective `EntityManager` remove operation for that entity is executed. It is not called for a DQL `DELETE` statement.
- `postRemove` - The `postRemove` event occurs for an entity after the entity has been deleted. It will be invoked after the database delete operations. It is not called for a DQL `DELETE` statement.
- `prePersist` - The `prePersist` event occurs for a given entity before the respective `EntityManager` persist operation for that entity is executed. It should be noted that this event is only triggered on *initial* persist of an entity (i.e. it does not trigger on future updates).
- `postPersist` - The `postPersist` event occurs for an entity after the entity has been made persistent. It will be invoked after the database insert operations. Generated primary key values are available in the postPersist event.
- `preUpdate` - The `preUpdate` event occurs before the database update operations to entity data. It is not called for a DQL `UPDATE` statement nor when the computed changeset is empty.
- `postUpdate` - The `postUpdate` event occurs after the database update operations to entity data. It is not called for a DQL `UPDATE` statement.
- `postLoad` - The postLoad event occurs for an entity after the entity has been loaded into the current `EntityManager` from the database or after the refresh operation has been applied to it.

- `loadClassMetadata` - The `loadClassMetadata` event occurs after the mapping metadata for a class has been loaded from a mapping source (annotations/xml/yaml). This event is not a lifecycle callback.
- `onClassMetadataNotFound` - Loading class metadata for a particular requested class name failed. Manipulating the given event args instance allows providing fallback metadata even when no actual metadata exists or could be found. This event is not a lifecycle callback.
- `preFlush` - The `preFlush` event occurs at the very beginning of a flush operation.
- `onFlush` - The `onFlush` event occurs after the change-sets of all managed entities are computed. This event is not a lifecycle callback.
- `postFlush` - The `postFlush` event occurs at the end of a flush operation. This event is not a lifecycle callback.
- `onClear` - The `onClear` event occurs when the `EntityManager#clear()` operation is invoked, after all references to entities have been removed from the unit of work. This event is not a lifecycle callback.

Note that, when using `Doctrine\ORM\AbstractQuery#toIterable()`, `postLoad` events will be executed immediately after objects are being hydrated, and therefore associations are not guaranteed to be initialized. It is not safe to combine usage of `Doctrine\ORM\AbstractQuery#toIterable()` and `postLoad` event handlers.

Note that the `postRemove` event or any events triggered after an entity removal can receive an uninitializable proxy in case you have configured an entity to cascade remove relations. In this case, you should load yourself the proxy in the associated pre event.

You can access the Event constants from the `Events` class in the ORM package.

```
<?php
use Doctrine\ORM\Events;
echo Events::preUpdate;
```

These can be hooked into by two different types of event listeners:

- Lifecycle Callbacks are methods on the entity classes that are called when the event is triggered. They receive some kind of `EventArgs` instance.
- Lifecycle Event Listeners and Subscribers are classes with specific callback methods that receives some kind of `EventArgs` instance.

The `EventArgs` instance received by the listener gives access to the entity, `EntityManager` instance and other relevant data.

All Lifecycle events that happen during the `flush()` of an `EntityManager` have very specific constraints on the allowed operations that can be executed. Please read the Events section very carefully to understand which operations are allowed in which lifecycle event.

Lifecycle Callbacks

Lifecycle Callbacks are defined on an entity class. They allow you to trigger callbacks whenever an instance of that entity class experiences a relevant lifecycle event. More than one callback can be defined for each lifecycle event. Lifecycle Callbacks are best used for simple operations specific to a particular entity class's lifecycle.

Note that Licecycle Callbacks are not supported for Embeddables.

```
<?php

/** @Entity @HasLifecycleCallbacks */
class User
{
    // ...

    /**
     * @Column(type="string", length=255)
     */
    public $value;

    /** @Column(name="created_at", type="string", length=255) */
    private $createdAt;

    /** @PrePersist */
    public function doStuffOnPrePersist()
    {
        $this->createdAt = date('Y-m-d H:i:s');
    }

    /** @PrePersist */
    public function doOtherStuffOnPrePersist()
    {
        $this->value = 'changed from prePersist callback!';
    }

    /** @PostPersist */
    public function doStuffOnPostPersist()
    {
        $this->value = 'changed from postPersist callback!';
    }

    /** @PostLoad */
    public function doStuffOnPostLoad()
    {
        $this->value = 'changed from postLoad callback!';
    }

    /** @PreUpdate */
    public function doStuffOnPreUpdate()
    {
        $this->value = 'changed from preUpdate callback!';
    }
}
```

Note that the methods set as lifecycle callbacks need to be public and, when using these annotations, you have to apply the `@HasLifecycleCallbacks` marker annotation on the entity class.

If you want to register lifecycle callbacks from YAML or XML you can do it with the following.

```
User:
  type: entity
  fields:
# ...
    name:
      type: string(50)
  lifecycleCallbacks:
    prePersist: [ doStuffOnPrePersist, doOtherStuffOnPrePersist ]
    postPersist: [ doStuffOnPostPersist ]
```

In YAML the `key` of the lifecycleCallbacks entry is the event that you are triggering on and the value is the method (or methods) to call. The allowed event types are the ones listed in the previous Lifecycle Events section.

XML would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-
mapping"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-
mapping
                          https://www.doctrine-project.org/schemas/orm/doctrine-
mapping.xsd">

    <entity name="User">

        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist" method="doStuffOnPrePersist"/>
            <lifecycle-callback type="postPersist"
method="doStuffOnPostPersist"/>
        </lifecycle-callbacks>

    </entity>

</doctrine-mapping>
```

In XML the `type` of the lifecycle-callback entry is the event that you are triggering on and the `method` is the method to call. The allowed event types are the ones listed in the previous Lifecycle Events section.

When using YAML or XML you need to remember to create public methods to match the callback names you defined. E.g. in these examples `doStuffOnPrePersist()`, `doOtherStuffOnPrePersist()` and `doStuffOnPostPersist()` methods need to be defined on your `User` model.

```
<?php
// ...

class User
{
  // ...

  public function doStuffOnPrePersist()
  {
    // ...
  }

  public function doOtherStuffOnPrePersist()
  {
    // ...
  }

  public function doStuffOnPostPersist()
  {
    // ...
  }
}
```

Lifecycle Callbacks Event Argument

The triggered event is also given to the lifecycle-callback.

With the additional argument you have access to the `EntityManager` and `UnitOfWork` APIs inside these callback methods.

```
<?php
// ...

class User
{
    public function preUpdate(PreUpdateEventArgs $event)
    {
        if ($event->hasChangedField('username')) {
            // Do something when the username is changed.
        }
    }
}
```

Listening and subscribing to Lifecycle Events

Lifecycle event listeners are much more powerful than the simple lifecycle callbacks that are defined on the entity classes. They sit at a level above the entities and allow you to implement re-usable behaviors across different entity classes.

Note that they require much more detailed knowledge about the inner workings of the `EntityManager` and `UnitOfWork` classes. Please read the Events section carefully if you are trying to write your own listener.

For event subscribers, there are no surprises. They declare the lifecycle events in their `getSubscribedEvents` method and provide public methods that expect the relevant arguments.

A lifecycle event listener looks like the following:

```
<?php
use Doctrine\Persistence\Event\LifecycleEventArgs;

class MyEventListener
{
    public function preUpdate(LifecycleEventArgs $args)
    {
        $entity = $args->getObject();
        $entityManager = $args->getObjectManager();

        // perhaps you only want to act on some "Product" entity
        if ($entity instanceof Product) {
            // do something with the Product
        }
    }
}
```

A lifecycle event subscriber may look like this:

```
<?php
use Doctrine\ORM\Events;
use Doctrine\EventSubscriber;
use Doctrine\Persistence\Event\LifecycleEventArgs;

class MyEventSubscriber implements EventSubscriber
{
    public function getSubscribedEvents()
    {
        return array(
            Events::postUpdate,
        );
    }

    public function postUpdate(LifecycleEventArgs $args)
    {
        $entity = $args->getObject();
        $entityManager = $args->getObjectManager();

        // perhaps you only want to act on some "Product" entity
        if ($entity instanceof Product) {
            // do something with the Product
        }
    }
```

Lifecycle events are triggered for all entities. It is the responsibility of the listeners and subscribers to check if the entity is of a type it wants to handle.

To register an event listener or subscriber, you have to hook it into the EventManager that is passed to the EntityManager factory:

```
<?php
$eventManager = new EventManager();
$eventManager->addEventListener(array(Events::preUpdate), new
MyEventListener());
$eventManager->addEventSubscriber(new MyEventSubscriber());

$entityManager = EntityManager::create($dbOpts, $config, $eventManager);
```

You can also retrieve the event manager instance after the EntityManager was created:

```
<?php
$entityManager->getEventManager()->addEventListener(array(Events::preUpdate),
new MyEventListener());
$entityManager->getEventManager()->addEventSubscriber(new MyEventSubscriber());
```

Implementing Event Listeners

This section explains what is and what is not allowed during specific lifecycle events of the `UnitOfWork` class. Although you get passed the `EntityManager` instance in all of these events, you have to follow these restrictions very carefully since operations in the wrong event may produce lots of different errors, such as inconsistent data and lost updates/persists/removes.

For the described events that are also lifecycle callback events the restrictions apply as well, with the additional restriction that (prior to version 2.4) you do not have access to the `EntityManager` or `UnitOfWork` APIs inside these events.

prePersist

There are two ways for the `prePersist` event to be triggered. One is obviously when you call `EntityManager#persist()`. The event is also called for all cascaded associations.

There is another way for `prePersist` to be called, inside the `flush()` method when changes to associations are computed and this association is marked as cascade persist. Any new entity found during this operation is also persisted and `prePersist` called on it. This is called persistence by reachability.

In both cases you get passed a `LifecycleEventArgs` instance which has access to the entity and the entity manager.

The following restrictions apply to `prePersist`:

- If you are using a PrePersist Identity Generator such as sequences the ID value will *NOT* be available within any PrePersist events.
- Doctrine will not recognize changes made to relations in a prePersist event. This includes modifications to collections such as additions, removals or replacement.

preRemove

The `preRemove` event is called on every entity when its passed to the `EntityManager#remove()` method. It is cascaded for all associations that are marked as cascade delete.

There are no restrictions to what methods can be called inside the `preRemove` event, except when the remove method itself was called during a flush operation.

preFlush

`preFlush` is called at `EntityManager#flush()` before anything else. `EntityManager#flush()` should not be called inside its listeners, since `preFlush` event is dispatched in it, which would result in infinite loop.

```
<?php

use Doctrine\ORM\Event\PreFlushEventArgs;

class PreFlushExampleListener
{
    public function preFlush(PreFlushEventArgs $args)
    {
        // ...
    }
}
```

onFlush

OnFlush is a very powerful event. It is called inside `EntityManager#flush()` after the changes to all the managed entities and their associations have been computed. This means, the `onFlush` event has access to the sets of:

- Entities scheduled for insert
- Entities scheduled for update
- Entities scheduled for removal
- Collections scheduled for update
- Collections scheduled for removal

To make use of the `onFlush` event you have to be familiar with the internal `UnitOfWork` API, which grants you access to the previously mentioned sets. See this example:

```
<?php
class FlushExampleListener
{
    public function onFlush(OnFlushEventArgs $eventArgs)
    {
        $em = $eventArgs->getEntityManager();
        $uow = $em->getUnitOfWork();

        foreach ($uow->getScheduledEntityInsertions() as $entity) {

        }

        foreach ($uow->getScheduledEntityUpdates() as $entity) {

        }

        foreach ($uow->getScheduledEntityDeletions() as $entity) {

        }

        foreach ($uow->getScheduledCollectionDeletions() as $col) {

        }

        foreach ($uow->getScheduledCollectionUpdates() as $col) {

        }
    }
}
```

The following restrictions apply to the onFlush event:

- If you create and persist a new entity in `onFlush`, then calling `EntityManager#persist()` is not enough. You have to execute an additional call to `$unitOfWork->computeChangeSet($classMetadata, $entity)`.

- Changing primitive fields or associations requires you to explicitly trigger a re-computation of the changeset of the affected entity. This can be done by calling `$unitOfWork->recomputeSingleEntityChangeSet($classMetadata, $entity)`.

postFlush

`postFlush` is called at the end of `EntityManager#flush()`. `EntityManager#flush()` can **NOT** be called safely inside its listeners.

```
<?php

use Doctrine\ORM\Event\PostFlushEventArgs;

class PostFlushExampleListener
{
    public function postFlush(PostFlushEventArgs $args)
    {
        // ...
    }
}
```

preUpdate

PreUpdate is the most restrictive to use event, since it is called right before an update statement is called for an entity inside the `EntityManager#flush()` method. Note that this event is not triggered when the computed changeset is empty.

Changes to associations of the updated entity are never allowed in this event, since Doctrine cannot guarantee to correctly handle referential integrity at this point of the flush operation. This event has a powerful feature however, it is executed with a `PreUpdateEventArgs` instance, which contains a reference to the computed change-set of this entity.

This means you have access to all the fields that have changed for this entity with their old and new value. The following methods are available on the `PreUpdateEventArgs`:

- `getEntity()` to get access to the actual entity.
- `getEntityChangeSet()` to get a copy of the changeset array. Changes to this returned array do not affect updating.
- `hasChangedField($fieldName)` to check if the given field name of the current entity changed.
- `getOldValue($fieldName)` and `getNewValue($fieldName)` to access the values of a field.
- `setNewValue($fieldName, $value)` to change the value of a field to be updated.

A simple example for this event looks like:

```
<?php
class NeverAliceOnlyBobListener
{
    public function preUpdate(PreUpdateEventArgs $eventArgs)
    {
        if ($eventArgs->getEntity() instanceof User) {
            if ($eventArgs->hasChangedField('name') && $eventArgs-
>getNewValue('name') == 'Alice') {
                $eventArgs->setNewValue('name', 'Bob');
            }
        }
    }
}
```

You could also use this listener to implement validation of all the fields that have changed. This is more efficient than using a lifecycle callback when there are expensive validations to call:

```
<?php
class ValidCreditCardListener
{
    public function preUpdate(PreUpdateEventArgs $eventArgs)
    {
        if ($eventArgs->getEntity() instanceof Account) {
            if ($eventArgs->hasChangedField('creditCard')) {
                $this->validateCreditCard($eventArgs-
>getNewValue('creditCard'));
            }
        }
    }

    private function validateCreditCard($no)
    {
        // throw an exception to interrupt flush event. Transaction will be
rolled back.
    }
}
```

Restrictions for this event:

- Changes to associations of the passed entities are not recognized by the flush operation anymore.
- Changes to fields of the passed entities are not recognized by the flush operation anymore, use the computed change-set passed to the event to modify primitive field values, e.g. use `$eventArgs->setNewValue($field, $value);` as in the Alice to Bob example above.
- Any calls to `EntityManager#persist()` or `EntityManager#remove()`, even in combination with the `UnitOfWork` API are strongly discouraged and don't work as expected outside the flush operation.

postUpdate, postRemove, postPersist

The three post events are called inside `EntityManager#flush()`. Changes in here are not relevant to the persistence in the database, but you can use these events to alter non-persistable items, like non-mapped fields, logging or even associated classes that are not directly mapped by Doctrine.

postLoad

This event is called after an entity is constructed by the EntityManager.

Entity listeners

An entity listener is a lifecycle listener class used for an entity.

- The entity listener's mapping may be applied to an entity class or mapped superclass.
- An entity listener is defined by mapping the entity class with the corresponding mapping.

- *PHP*

```
<?php
namespace MyProject\Entity;

/** @Entity @EntityListeners({"UserListener"}) */
class User
{
    // ....
}
```

Entity listeners class

An `Entity Listener` could be any class, by default it should be a class with a no-arg constructor.

- Different from Events an `Entity Listener` is invoked just to the specified entity
- An entity listener method receives two arguments, the entity instance and the lifecycle event.
- The callback method can be defined by naming convention or specifying a method mapping.
- When a listener mapping is not given the parser will use the naming convention to look for a matching method, e.g. it will look for a public `preUpdate()` method if you are listening to the `preUpdate` event.
- When a listener mapping is given the parser will not look for any methods using the naming convention.

```
<?php
class UserListener
{
    public function preUpdate(User $user, PreUpdateEventArgs $event)
    {
        // Do something on pre update.
    }
}
```

To define a specific event listener method (one that does not follow the naming convention) you need to map the listener method using the event type mapping:

- *PHP*

```
<?php
class UserListener
{
    /** @PrePersist */
    public function prePersistHandler(User $user, LifecycleEventArgs $event)
{ // ... }

    /** @PostPersist */
    public function postPersistHandler(User $user, LifecycleEventArgs $event)
{ // ... }

    /** @PreUpdate */
    public function preUpdateHandler(User $user, PreUpdateEventArgs $event) { //
... }

    /** @PostUpdate */
    public function postUpdateHandler(User $user, LifecycleEventArgs $event)
{ // ... }

    /** @PostRemove */
    public function postRemoveHandler(User $user, LifecycleEventArgs $event)
{ // ... }

    /** @PreRemove */
    public function preRemoveHandler(User $user, LifecycleEventArgs $event) { //
... }

    /** @PreFlush */
    public function preFlushHandler(User $user, PreFlushEventArgs $event)
{ // ... }

    /** @PostLoad */
    public function postLoadHandler(User $user, LifecycleEventArgs $event)
{ // ... }
}
```

The order of execution of multiple methods for the same event (e.g. multiple @PrePersist) is not guaranteed.

Entity listeners resolver

Doctrine invokes the listener resolver to get the listener instance.

- A resolver allows you register a specific entity listener instance.
- You can also implement your own resolver by extending `Doctrine\ORM\Mapping\DefaultEntityListenerResolver` or implementing `Doctrine\ORM\Mapping\EntityListenerResolver`

Specifying an entity listener instance :

```
<?php
// User.php

/** @Entity @EntityListeners({"UserListener"}) */
class User
{
    // ....
}

// UserListener.php
class UserListener
{
    public function __construct(MyService $service)
    {
        $this->service = $service;
    }

    public function preUpdate(User $user, PreUpdateEventArgs $event)
    {
        $this->service->doSomething($user);
    }
```

}

// register a entity listener.
$listener = $container->get('user_listener');
$em→getConfiguration()→getEntityListenerResolver()→register($listener);

Implementing your own resolver :

```
<?php
class MyEntityListenerResolver extends \Doctrine\ORM\Mapping\
DefaultEntityListenerResolver
{
    public function __construct($container)
    {
        $this->container = $container;
    }

    public function resolve($className)
    {
        // resolve the service id by the given class name;
        $id = 'user_listener';

        return $this->container->get($id);
    }
}

// Configure the listener resolver only before instantiating the EntityManager
$configurations->setEntityListenerResolver(new MyEntityListenerResolver);
EntityManager::create(.., $configurations, ..);
```

Load ClassMetadata Event

When the mapping information for an entity is read, it is populated in to a `ClassMetadataInfo` instance. You can hook in to this process and manipulate the instance.

```
<?php
$test = new TestEvent();
$metadataFactory = $em->getMetadataFactory();
$evm = $em->getEventManager();
$evm->addEventListener(Events::loadClassMetadata, $test);

class TestEvent
{
    public function loadClassMetadata(\Doctrine\ORM\Event\
LoadClassMetadataEventArgs $eventArgs)
    {
        $classMetadata = $eventArgs->getClassMetadata();
        $fieldMapping = array(
            'fieldName' => 'about',
            'type' => 'string',
            'length' => 255
        );
        $classMetadata->mapField($fieldMapping);
    }
}
```

Doctrine Internals explained

Object relational mapping is a complex topic and sufficiently understanding how Doctrine works internally helps you use its full power.

How Doctrine keeps track of Objects

Doctrine uses the Identity Map pattern to track objects. Whenever you fetch an object from the database, Doctrine will keep a reference to this object inside its UnitOfWork. The array holding all the entity references is two-levels deep and has the keys root entity name and id. Since Doctrine allows composite keys the id is a sorted, serialized version of all the key columns.

This allows Doctrine room for optimizations. If you call the EntityManager and ask for an entity with a specific ID twice, it will return the same instance:

```
public function testIdentityMap()
{
    $objectA = $this->entityManager->find('EntityName', 1);
    $objectB = $this->entityManager->find('EntityName', 1);

    $this->assertSame($objectA, $objectB)
}
```

Only one SELECT query will be fired against the database here. In the second `EntityManager#find()` call Doctrine will check the identity map first and doesn't need to make that database roundtrip.

Even if you get a proxy object first then fetch the object by the same id you will still end up with the same reference:

```
public function testIdentityMapReference()
{
    $objectA = $this->entityManager->getReference('EntityName', 1);
    // check for proxyinterface
    $this->assertInstanceOf('Doctrine\ORM\Proxy\Proxy', $objectA);

    $objectB = $this->entityManager->find('EntityName', 1);

    $this->assertSame($objectA, $objectB)
}
```

The identity map being indexed by primary keys only allows shortcuts when you ask for objects by primary key. Assume you have the following `persons` table:

```
id | name
-------------
1  | Benjamin
2  | Bud
```

Take the following example where two consecutive calls are made against a repository to fetch an entity by a set of criteria:

```
public function testIdentityMapRepositoryFindBy()
{
    $repository = $this->entityManager->getRepository('Person');
    $objectA = $repository->findOneBy(array('name' => 'Benjamin'));
    $objectB = $repository->findOneBy(array('name' => 'Benjamin'));

    $this->assertSame($objectA, $objectB);
}
```

This query will still return the same references and `$objectA` and `$objectB` are indeed referencing the same object. However when checking your SQL logs you will realize that two queries have been executed against the database. Doctrine only knows objects by id, so a query for different criteria has to go to the database, even if it was executed just before.

But instead of creating a second Person object Doctrine first gets the primary key from the row and check if it already has an object inside the UnitOfWork with that primary key. In our example it finds an object and decides to return this instead of creating a new one.

The identity map has a second use-case. When you call `EntityManager#flush` Doctrine will ask the identity map for all objects that are currently managed. This means you don't have to call `EntityManager#persist` over and over again to pass known objects to the EntityManager. This is a NO-OP for known entities, but leads to much code written that is confusing to other developers.

The following code WILL update your database with the changes made to the `Person` object, even if you did not call `EntityManager#persist`:

```
<?php
$user = $entityManager->find("Person", 1);
$user->setName("Guilherme");
$entityManager->flush();
```

How Doctrine Detects Changes

Doctrine is a data-mapper that tries to achieve persistence-ignorance (PI). This means you map php objects into a relational database that don't necessarily know about the database at all. A natural question would now be, how does Doctrine even detect objects have changed?.

For this Doctrine keeps a second map inside the UnitOfWork. Whenever you fetch an object from the database Doctrine will keep a copy of all the properties and associations inside the UnitOfWork. Because variables in the PHP language are subject to copy-on-write the memory usage of a PHP request that only reads objects from the database is the same as if Doctrine did not keep this variable copy. Only if you start changing variables PHP will create new variables internally that consume new memory.

Now whenever you call `EntityManager#flush` Doctrine will iterate over the Identity Map and for each object compares the original property and association values with the values that are currently set on the object. If changes are detected then the object is queued for a SQL UPDATE operation. Only the fields that actually changed are updated.

This process has an obvious performance impact. The larger the size of the UnitOfWork is, the longer this computation takes. There are several ways to optimize the performance of the Flush Operation:

- Mark entities as read only. These entities can only be inserted or removed, but are never updated. They are omitted in the changeset calculation.
- Temporarily mark entities as read only. If you have a very large UnitOfWork but know that a large set of entities has not changed, just mark them as read only with `$entityManager->getUnitOfWork()->markReadOnly($entity)`.
- Flush only a single entity with `$entityManager->flush($entity)`.
- Use Change Tracking Policies to use more explicit strategies of notifying the UnitOfWork what objects/properties changed.

Query Internals

The different ORM Layers

Doctrine ships with a set of layers with different responsibilities. This section gives a short explanation of each layer.

Hydration

Responsible for creating a final result from a raw database statement and a result-set mapping object. The developer can choose which kind of result they wish to be hydrated. Default result-types include:

- SQL to Entities
- SQL to structured Arrays
- SQL to simple scalar result arrays
- SQL to a single result variable

Hydration to entities and arrays is one of the most complex parts of Doctrine algorithm-wise. It can build results with for example:

- Single table selects
- Joins with n:1 or 1:n cardinality, grouping belonging to the same parent.
- Mixed results of objects and scalar values
- Hydration of results by a given scalar value as key.

Persisters

tbr

UnitOfWork

tbr

ResultSetMapping

tbr

DQL Parser

tbr

SQLWalker

tbr

EntityManager

tbr

ClassMetadataFactory

tbr

Association Updates: Owning Side and Inverse Side

When mapping bidirectional associations it is important to understand the concept of the owning and inverse sides. The following general rules apply:

- Relationships may be bidirectional or unidirectional.
- A bidirectional relationship has both an owning side and an inverse side
- A unidirectional relationship only has an owning side.
- Doctrine will **only** check the owning side of an association for changes.

Bidirectional Associations

The following rules apply to **bidirectional** associations:

- The inverse side has to have the `mappedBy` attribute of the OneToOne, OneToMany, or ManyToMany mapping declaration. The mappedBy attribute contains the name of the association-field on the owning side.
- The owning side has to have the `inversedBy` attribute of the OneToOne, ManyToOne, or ManyToMany mapping declaration.
  The inversedBy attribute contains the name of the association-field on the inverse-side.
- ManyToOne is always the owning side of a bidirectional association.
- OneToMany is always the inverse side of a bidirectional association.
- The owning side of a OneToOne association is the entity with the table containing the foreign key.
- You can pick the owning side of a many-to-many association yourself.

Important concepts

Doctrine will only check the owning side of an association for changes.

To fully understand this, remember how bidirectional associations are maintained in the object world. There are 2 references on each side of the association and these 2 references both represent the same association but can change independently of one another. Of course, in a correct application the semantics of the bidirectional association are properly maintained by the application developer (that's their responsibility). Doctrine needs to know which of these two in-memory references is the one that should be persisted and which not. This is what the owning/inverse concept is mainly used for.

Changes made only to the inverse side of an association are ignored. Make sure to update both sides of a bidirectional association (or at least the owning side, from Doctrine's point of view)

The owning side of a bidirectional association is the side Doctrine looks at when determining the state of the association, and consequently whether there is anything to do to update the association in the database.

> Owning side and inverse side are technical concepts of the ORM technology, not concepts of your domain model. What you consider as the owning side in your domain model can be different from what the owning side is for Doctrine. These are unrelated.

Transactions and Concurrency

Transaction Demarcation

Transaction demarcation is the task of defining your transaction boundaries. Proper transaction demarcation is very important because if not done properly it can negatively affect the performance of your application. Many databases and database abstraction layers like PDO by default operate in auto-commit mode, which means that every single SQL statement is wrapped in a small transaction. Without any explicit transaction demarcation from your side, this quickly results in poor performance because transactions are not cheap.

For the most part, Doctrine ORM already takes care of proper transaction demarcation for you: All the write operations (INSERT/UPDATE/DELETE) are queued until `EntityManager#flush()` is invoked which wraps all of these changes in a single transaction.

However, Doctrine ORM also allows (and encourages) you to take over and control transaction demarcation yourself.

These are two ways to deal with transactions when using the Doctrine ORM and are now described in more detail.

Approach 1: Implicitly

The first approach is to use the implicit transaction handling provided by the Doctrine ORM EntityManager. Given the following code snippet, without any explicit transaction demarcation:

```
<?php
// $em instanceof EntityManager
$user = new User;
$user->setName('George');
$em->persist($user);
$em->flush();
```

Since we do not do any custom transaction demarcation in the above code, `EntityManager#flush()` will begin and commit/rollback a transaction. This behavior is made possible by the aggregation of the DML operations by the Doctrine ORM and is sufficient if all the data manipulation that is part of a unit of work happens through the domain model and thus the ORM.

Approach 2: Explicitly

The explicit alternative is to use the `Doctrine\DBAL\Connection` API directly to control the transaction boundaries. The code then looks like this:

```
<?php
// $em instanceof EntityManager
$em->getConnection()->beginTransaction(); // suspend auto-commit
try {
    //... do some work
    $user = new User;
    $user->setName('George');
    $em->persist($user);
    $em->flush();
    $em->getConnection()->commit();
} catch (Exception $e) {
    $em->getConnection()->rollBack();
    throw $e;
}
```

Explicit transaction demarcation is required when you want to include custom DBAL operations in a unit of work or when you want to make use of some methods of the `EntityManager` API that require an active transaction. Such methods will throw a `TransactionRequiredException` to inform you of that requirement.

A more convenient alternative for explicit transaction demarcation is the use of provided control abstractions in the form of `Connection#transactional($func)` and `EntityManager#transactional($func)`. When used, these control abstractions ensure that you never forget to rollback the transaction, in addition to the obvious code reduction. An example that is functionally equivalent to the previously shown code looks as follows:

<?php // $em instanceof EntityManager $em->transactional(function($em) { //... do some work $user = new User; $user->setName('George'); $em->persist($user); });

For historical reasons, `EntityManager#transactional($func)` will return `true` whenever the return value of `$func` is loosely false. Some examples of this include `array()`, `"0"`, `""`, `0`, and `null`.

The difference between `Connection#transactional($func)` and `EntityManager#transactional($func)` is that the latter abstraction flushes the `EntityManager` prior to transaction commit and rolls back the transaction when an exception occurs.

Exception Handling

When using implicit transaction demarcation and an exception occurs during `EntityManager#flush()`, the transaction is automatically rolled back and the `EntityManager` closed.

When using explicit transaction demarcation and an exception occurs, the transaction should be rolled back immediately and the `EntityManager` closed by invoking `EntityManager#close()` and subsequently discarded, as demonstrated in the example above. This can be handled elegantly by the control abstractions shown earlier. Note that when catching `Exception` you should generally re-throw the exception. If you intend to recover from some exceptions, catch them explicitly in earlier catch blocks (but do not forget to rollback the transaction and close the `EntityManager` there as well). All other best practices of exception handling apply similarly (i.e. either log or re-throw, not both, etc.).

As a result of this procedure, all previously managed or removed instances of the `EntityManager` become detached. The state of the detached objects will be the state at the point at which the transaction was rolled back. The state of the objects is in no way rolled back and thus the objects are now out of synch with the database. The application can continue to use the detached objects, knowing that their state is potentially no longer accurate.

If you intend to start another unit of work after an exception has occurred you should do that with a new `EntityManager`.

Locking Support

Doctrine ORM offers support for Pessimistic- and Optimistic-locking strategies natively. This allows to take very fine-grained control over what kind of locking is required for your Entities in your application.

Optimistic Locking

Database transactions are fine for concurrency control during a single request. However, a database transaction should not span across requests, the so-called user think time. Therefore a long-running business transaction that spans multiple requests needs to involve several database transactions. Thus, database transactions alone can no longer control concurrency during such a long-running business transaction. Concurrency control becomes the partial responsibility of the application itself.

Doctrine has integrated support for automatic optimistic locking via a version field. In this approach any entity that should be protected against concurrent modifications during long-running business transactions gets a version field that is either a simple number (mapping type: integer) or a timestamp (mapping type: datetime). When changes to such an entity are persisted at the end of a long-running conversation the version of the entity is compared to the version in the database and if they don't match, an `OptimisticLockException` is thrown, indicating that the entity has been modified by someone else already.

You designate a version field in an entity as follows. In this example we'll use an integer.

- *PHP*

```
<?php
class User
{
    // ...
    /** @Version @Column(type="integer") */
    private $version;
    // ...
}
```

Alternatively a datetime type can be used (which maps to a SQL timestamp or datetime):

- *PHP*

```
<?php
class User
{
    // ...
    /** @Version @Column(type="datetime") */
    private $version;
    // ...
}
```

Version numbers (not timestamps) should however be preferred as they can not potentially conflict in a highly concurrent environment, unlike timestamps where this is a possibility, depending on the resolution of the timestamp on the particular database platform.

When a version conflict is encountered during `EntityManager#flush()`, an `OptimisticLockException` is thrown and the active transaction rolled back (or marked for rollback). This exception can be caught and handled. Potential responses to an OptimisticLockException are to present the conflict to the user or to refresh or reload objects in a new transaction and then retrying the transaction.

With PHP promoting a share-nothing architecture, the time between showing an update form and actually modifying the entity can in the worst scenario be as long as your applications session timeout. If changes happen to the entity in that time frame you want to know directly when retrieving the entity that you will hit an optimistic locking exception:

You can always verify the version of an entity during a request either when calling `EntityManager#find()`:

```
<?php
use Doctrine\DBAL\LockMode;
use Doctrine\ORM\OptimisticLockException;

$theEntityId = 1;
$expectedVersion = 184;

try {
    $entity = $em->find('User', $theEntityId, LockMode::OPTIMISTIC,
$expectedVersion);

    // do the work

    $em->flush();
} catch(OptimisticLockException $e) {
    echo "Sorry, but someone else has already changed this entity. Please apply
the changes again!";
}
```

Or you can use `EntityManager#lock()` to find out:

```
<?php
use Doctrine\DBAL\LockMode;
use Doctrine\ORM\OptimisticLockException;

$theEntityId = 1;
$expectedVersion = 184;

$entity = $em->find('User', $theEntityId);

try {
    // assert version
    $em->lock($entity, LockMode::OPTIMISTIC, $expectedVersion);

} catch(OptimisticLockException $e) {
    echo "Sorry, but someone else has already changed this entity. Please apply
the changes again!";
}
```

Important Implementation Notes

You can easily get the optimistic locking workflow wrong if you compare the wrong versions. Say you have Alice and Bob editing a hypothetical blog post:

- Alice reads the headline of the blog post being "Foo", at optimistic lock version 1 (GET Request)
- Bob reads the headline of the blog post being "Foo", at optimistic lock version 1 (GET Request)
- Bob updates the headline to "Bar", upgrading the optimistic lock version to 2 (POST Request of a Form)
- Alice updates the headline to "Baz", ... (POST Request of a Form)

Now at the last stage of this scenario the blog post has to be read again from the database before Alice's headline can be applied. At this point you will want to check if the blog post is still at version 1 (which it is not in this scenario).

Using optimistic locking correctly, you *have* to add the version as an additional hidden field (or into the SESSION for more safety). Otherwise you cannot verify the version is still the one being originally read from the database when Alice performed her GET request for the blog post. If this happens you might see lost updates you wanted to prevent with Optimistic Locking.

See the example code, The form (GET Request):

```
<?php
$post = $em->find('BlogPost', 123456);

echo '<input type="hidden" name="id" value="' . $post->getId() . '" />';
echo '<input type="hidden" name="version" value="' . $post->getCurrentVersion() . '" />';
```

And the change headline action (POST Request):

```
<?php
$postId = (int)$_GET['id'];
$postVersion = (int)$_GET['version'];
```

```
$post = $em->find('BlogPost', $postId, \Doctrine\DBAL\LockMode::OPTIMISTIC,
$postVersion);
```

Pessimistic Locking

Doctrine ORM supports Pessimistic Locking at the database level. No attempt is being made to implement pessimistic locking inside Doctrine, rather vendor-specific and ANSI-SQL commands are used to acquire row-level locks. Every Entity can be part of a pessimistic lock, there is no special metadata required to use this feature.

However for Pessimistic Locking to work you have to disable the Auto-Commit Mode of your Database and start a transaction around your pessimistic lock use-case using the Approach 2: Explicit Transaction Demarcation described above. Doctrine ORM will throw an Exception if you attempt to acquire an pessimistic lock and no transaction is running.

Doctrine ORM currently supports two pessimistic lock modes:

- Pessimistic Write (`Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE`), locks the underlying database rows for concurrent Read and Write Operations.
- Pessimistic Read (`Doctrine\DBAL\LockMode::PESSIMISTIC_READ`), locks other concurrent requests that attempt to update or lock rows in write mode.

You can use pessimistic locks in three different scenarios:

1. Using `EntityManager#find($className, $id, \Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE)` or `EntityManager#find($className, $id, \Doctrine\DBAL\LockMode::PESSIMISTIC_READ)`
2. Using `EntityManager#lock($entity, \Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE)` or `EntityManager#lock($entity, \Doctrine\DBAL\LockMode::PESSIMISTIC_READ)`
3. Using `Query#setLockMode(\Doctrine\DBAL\LockMode::PESSIMISTIC_WRITE)` or `Query#setLockMode(\Doctrine\DBAL\LockMode::PESSIMISTIC_READ)`

Batch Processing

This chapter shows you how to accomplish bulk inserts, updates and deletes with Doctrine in an efficient way. The main problem with bulk operations is usually not to run out of memory and this is especially what the strategies presented here provide help with.

An ORM tool is not primarily well-suited for mass inserts, updates or deletions. Every RDBMS has its own, most effective way of dealing with such operations and if the options outlined below are not sufficient for your purposes we recommend you use the tools for your particular RDBMS for these bulk operations.

Having an SQL logger enabled when processing batches can have a serious impact on performance and resource usage. To avoid that you should disable it in the DBAL configuration:

```
<?php
$em->getConnection()->getConfiguration()->setSQLLogger(null);
```

Bulk Inserts

Bulk inserts in Doctrine are best performed in batches, taking advantage of the transactional write-behind behavior of an `EntityManager`. The following code shows an example for inserting 10000 objects with a batch size of 20. You may need to experiment with the batch size to find the size that works best for you. Larger batch sizes mean more prepared statement reuse internally but also mean more work during `flush`.

```
<?php
$batchSize = 20;
for ($i = 1; $i <= 10000; ++$i) {
    $user = new CmsUser;
    $user->setStatus('user');
    $user->setUsername('user' . $i);
    $user->setName('Mr.Smith-' . $i);
    $em->persist($user);
    if (($i % $batchSize) === 0) {
        $em->flush();
        $em->clear(); // Detaches all objects from Doctrine!
    }
}
$em->flush(); //Persist objects that did not make up an entire batch
$em->clear();
```

Bulk Updates

There are 2 possibilities for bulk updates with Doctrine.

DQL UPDATE

The by far most efficient way for bulk updates is to use a DQL UPDATE query. Example:

```
<?php
$q = $em->createQuery('update MyProject\Model\Manager m set m.salary = m.salary
* 0.9');
$numUpdated = $q->execute();
```

Iterating results

An alternative solution for bulk updates is to use the `Query#toIterable()` facility to iterate over the query results step by step instead of loading the whole result into memory at once. The following example shows how to do this, combining the iteration with the batching strategy that was already used for bulk inserts:

```
<?php
$batchSize = 20;
$i = 1;
$q = $em->createQuery('select u from MyProject\Model\User u');
foreach ($q->toIterable() as $user) {
    $user->increaseCredit();
    $user->calculateNewBonuses();
    if (($i % $batchSize) === 0) {
        $em->flush(); // Executes all updates.
        $em->clear(); // Detaches all objects from Doctrine!
    }
    ++$i;
}
$em->flush();
```

Iterating results is not possible with queries that fetch-join a collection-valued association. The nature of such SQL result sets is not suitable for incremental hydration.

Results may be fully buffered by the database client/ connection allocating additional memory not visible to the PHP process. For large sets this may easily kill the process for no apparent reason.

Bulk Deletes

There are two possibilities for bulk deletes with Doctrine. You can either issue a single DQL DELETE query or you can iterate over results removing them one at a time.

DQL DELETE

The by far most efficient way for bulk deletes is to use a DQL DELETE query.

Example:

```
<?php
$q = $em->createQuery('delete from MyProject\Model\Manager m where m.salary >
100000');
$numDeleted = $q->execute();
```

Iterating results

An alternative solution for bulk deletes is to use the `Query#toIterable()` facility to iterate over the query results step by step instead of loading the whole result into memory at once. The following example shows how to do this:

```
<?php
$batchSize = 20;
$i = 1;
$q = $em->createQuery('select u from MyProject\Model\User u');
foreach($q->toIterable() as $row) {
    $em->remove($row);
    if (($i % $batchSize) === 0) {
        $em->flush(); // Executes all deletions.
        $em->clear(); // Detaches all objects from Doctrine!
    }
    ++$i;
}
$em->flush();
```

Iterating results is not possible with queries that fetch-join a collection-valued association. The nature of such SQL result sets is not suitable for incremental hydration.

Iterating Large Results for Data-Processing

You can use the `toIterable()` method just to iterate over a large result and no UPDATE or DELETE intention. `$query->toIterable()` returns `iterable` so you can process a large result without memory problems using the following approach:

```
<?php
$q = $this->_em->createQuery('select u from MyProject\Model\User u');
foreach ($q->toIterable() as $row) {
    // do stuff with the data in the row

    // detach from Doctrine, so that it can be Garbage-Collected immediately
    $this->_em->detach($row[0]);
}
```

Iterating results is not possible with queries that fetch-join a collection-valued association. The nature of such SQL result sets is not suitable for incremental hydration.

Doctrine Query Language

DQL stands for Doctrine Query Language and is an Object Query Language derivative that is very similar to the Hibernate Query Language (HQL) or the Java Persistence Query Language (JPQL).

In essence, DQL provides powerful querying capabilities over your object model. Imagine all your objects lying around in some storage (like an object database). When writing DQL queries, think about querying that storage to pick a certain subset of your objects.

A common mistake for beginners is to mistake DQL for being just some form of SQL and therefore trying to use table names and column names or join arbitrary tables together in a query. You need to think about DQL as a query language for your object model, not for your relational schema.

DQL is case in-sensitive, except for namespace, class and field names, which are case sensitive.

Types of DQL queries

DQL as a query language has SELECT, UPDATE and DELETE constructs that map to their corresponding SQL statement types. INSERT statements are not allowed in DQL, because entities and their relations have to be introduced into the persistence context through `EntityManager#persist()` to ensure consistency of your object model.

DQL SELECT statements are a very powerful way of retrieving parts of your domain model that are not accessible via associations. Additionally they allow you to retrieve entities and their associations in one single SQL select statement which can make a huge difference in performance compared to using several queries.

DQL UPDATE and DELETE statements offer a way to execute bulk changes on the entities of your domain model. This is often necessary when you cannot load all the affected entities of a bulk update into memory.

SELECT queries

DQL SELECT clause

Here is an example that selects all users with an age > 20:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.age >
20');
$users = $query->getResult();
```

Lets examine the query:

- `u` is a so called identification variable or alias that refers to the `MyProject\Model\User` class. By placing this alias in the SELECT clause we specify that we want all instances of the User class that are matched by this query to appear in the query result.
- The FROM keyword is always followed by a fully-qualified class name which in turn is followed by an identification variable or alias for that class name. This class designates a root of our query from which we can navigate further via joins (explained later) and path expressions.
- The expression `u.age` in the WHERE clause is a path expression. Path expressions in DQL are easily identified by the use of the '.' operator that is used for constructing paths. The path expression `u.age` refers to the `age` field on the User class.

The result of this query would be a list of User objects where all users are older than 20.

Result format

The composition of the expressions in the SELECT clause also influences the nature of the query result. There are three cases:

All objects

```
SELECT u, p, n FROM Users u...
```

In this case, the result will be an array of User objects because of the FROM clause, with children `p` and `n` hydrated because of their inclusion in the SELECT clause.

All scalars

```
SELECT u.name, u.address FROM Users u...
```

In this case, the result will be an array of arrays. In the example above, each element of the result array would be an array of the scalar name and address values.

You can select scalars from any entity in the query.

Mixed

```
SELECT u, p.quantity FROM Users u...
```

Here, the result will again be an array of arrays, with each element being an array made up of a User object and the scalar value `p.quantity`.

Multiple FROM clauses are allowed, which would cause the result array elements to cycle through the classes included in the multiple FROM clauses.

> You cannot select other entities unless you also select the root of the selection (which is the first entity in FROM).
>
> For example, `SELECT p,n FROM Users u...` would be wrong because `u` is not part of the SELECT
>
> Doctrine throws an exception if you violate this constraint.

Joins

A SELECT query can contain joins. There are 2 types of JOINs: Regular Joins and Fetch Joins.

Regular Joins: Used to limit the results and/or compute aggregate values.

Fetch Joins: In addition to the uses of regular joins: Used to fetch related entities and include them in the hydrated result of a query.

There is no special DQL keyword that distinguishes a regular join from a fetch join. A join (be it an inner or outer join) becomes a fetch join as soon as fields of the joined entity appear in the SELECT part of the DQL query outside of an aggregate function. Otherwise its a regular join.

Example:

Regular join of the address:

```
<?php
$query = $em->createQuery("SELECT u FROM User u JOIN u.address a WHERE a.city =
'Berlin'");
$users = $query->getResult();
```

Fetch join of the address:

```
<?php
$query = $em->createQuery("SELECT u, a FROM User u JOIN u.address a WHERE a.city
= 'Berlin'");
$users = $query->getResult();
```

When Doctrine hydrates a query with fetch-join it returns the class in the FROM clause on the root level of the result array. In the previous example an array of User instances is returned and the address of each user is fetched and hydrated into the `User#address` variable. If you access the address Doctrine does not need to lazy load the association with another query.

> Doctrine allows you to walk all the associations between all the objects in your domain model. Objects that were not already loaded from the database are replaced with lazy load proxy instances. Non-loaded Collections are also replaced by lazy-load instances that fetch all the contained objects upon first access. However relying on the lazy-load mechanism leads to many small queries executed against the database, which can significantly affect the performance of your application. **Fetch Joins** are the solution to hydrate most or all of the entities that you need

in a single SELECT query.

Named and Positional Parameters

DQL supports both named and positional parameters, however in contrast to many SQL dialects positional parameters are specified with numbers, for example ?1, ?2 and so on. Named parameters are specified with :name1, :name2 and so on.

When referencing the parameters in `Query#setParameter($param, $value)` both named and positional parameters are used **without** their prefixes.

DQL SELECT Examples

This section contains a large set of DQL queries and some explanations of what is happening. The actual result also depends on the hydration mode.

Hydrate all User entities:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u');
$users = $query->getResult(); // array of User objects
```

Retrieve the IDs of all CmsUsers:

<?php $query = $em->createQuery('SELECT u.id FROM CmsUser u'); $ids = $query->getResult(); // array of CmsUser ids

Retrieve the IDs of all users that have written an article:

```
<?php
$query = $em->createQuery('SELECT DISTINCT u.id FROM CmsArticle a JOIN a.user
u');
$ids = $query->getResult(); // array of CmsUser ids
```

Retrieve all articles and sort them by the name of the articles users instance:

```
<?php
$query = $em->createQuery('SELECT a FROM CmsArticle a JOIN a.user u ORDER BY
u.name ASC');
$articles = $query->getResult(); // array of CmsArticle objects
```

Retrieve the Username and Name of a CmsUser:

```
<?php
$query = $em->createQuery('SELECT u.username, u.name FROM CmsUser u');
$users = $query->getResult(); // array of CmsUser username and name values
echo $users[0]['username'];
```

Retrieve a ForumUser and its single associated entity:

```
<?php
$query = $em->createQuery('SELECT u, a FROM ForumUser u JOIN u.avatar a');
$users = $query->getResult(); // array of ForumUser objects with the avatar
association loaded
echo get_class($users[0]->getAvatar());
```

Retrieve a CmsUser and fetch join all the phonenumbers it has:

```
<?php
```

```
$query = $em->createQuery('SELECT u, p FROM CmsUser u JOIN u.phonenumbers p');
$users = $query->getResult(); // array of CmsUser objects with the phonenumbers
association loaded
$phonenumbers = $users[0]->getPhonenumbers();
```

Hydrate a result in Ascending:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id ASC');
$users = $query->getResult(); // array of ForumUser objects
```

Or in Descending Order:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id DESC');
$users = $query->getResult(); // array of ForumUser objects
```

Using Aggregate Functions:

```
<?php
$query = $em->createQuery('SELECT COUNT(u.id) FROM Entities\User u');
$count = $query->getSingleScalarResult();

$query = $em->createQuery('SELECT u, count(g.id) FROM Entities\User u JOIN
u.groups g GROUP BY u.id');
$result = $query->getResult();
```

With WHERE Clause and Positional Parameter:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$users = $query->getResult(); // array of ForumUser objects
```

With WHERE Clause and Named Parameter:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.username = :name');
$query->setParameter('name', 'Bob');
$users = $query->getResult(); // array of ForumUser objects
```

With Nested Conditions in WHERE Clause:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE (u.username = :name
OR u.username = :name2) AND u.id = :id');
$query->setParameters(array(
    'name' => 'Bob',
    'name2' => 'Alice',
    'id' => 321,
));
$users = $query->getResult(); // array of ForumUser objects
```

With COUNT DISTINCT:

```
<?php
$query = $em->createQuery('SELECT COUNT(DISTINCT u.name) FROM CmsUser');
$users = $query->getResult(); // array of ForumUser objects
```

With Arithmetic Expression in WHERE clause:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE ((u.id + 5000) * u.id +
3) < 10000000');
$users = $query->getResult(); // array of ForumUser objects
```

Retrieve user entities with Arithmetic Expression in ORDER clause, using the `HIDDEN` keyword:

```
<?php
$query = $em->createQuery('SELECT u, u.posts_count + u.likes_count AS HIDDEN
score FROM CmsUser u ORDER BY score');
$users = $query->getResult(); // array of User objects
```

Using a LEFT JOIN to hydrate all user-ids and optionally associated article-ids:

```
<?php
$query = $em->createQuery('SELECT u.id, a.id as article_id FROM CmsUser u LEFT
JOIN u.articles a');
$results = $query->getResult(); // array of user ids and every article_id for
each user
```

Restricting a JOIN clause by additional conditions specified by WITH:

```
<?php
$query = $em->createQuery("SELECT u FROM CmsUser u LEFT JOIN u.articles a WITH
a.topic LIKE :foo");
$query->setParameter('foo', '%foo%');
$users = $query->getResult();
```

Using several Fetch JOINs:

```
<?php
$query = $em->createQuery('SELECT u, a, p, c FROM CmsUser u JOIN u.articles a
JOIN u.phonenumbers p JOIN a.comments c');
$users = $query->getResult();
```

BETWEEN in WHERE clause:

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id BETWEEN ?1
AND ?2');
$query->setParameter(1, 123);
$query->setParameter(2, 321);
$usernames = $query->getResult();
```

DQL Functions in WHERE clause:

```
<?php
$query = $em->createQuery("SELECT u.name FROM CmsUser u WHERE TRIM(u.name) =
'someone'");
$usernames = $query->getResult();
```

IN() Expression:

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id IN(46)');
$usernames = $query->getResult();

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id IN (1, 2)');
$users = $query->getResult();
```

```
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id NOT IN (1)');
$users = $query->getResult();
```

CONCAT() DQL Function:

```
<?php
$query = $em->createQuery("SELECT u.id FROM CmsUser u WHERE CONCAT(u.name, 's')
= ?1");
$query->setParameter(1, 'Jess');
$ids = $query->getResult();

$query = $em->createQuery('SELECT CONCAT(u.id, u.name) FROM CmsUser u WHERE u.id
= ?1');
$query->setParameter(1, 321);
$idUsernames = $query->getResult();
```

EXISTS in WHERE clause with correlated Subquery

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE EXISTS (SELECT
p.phonenumber FROM CmsPhonenumber p WHERE p.user = u.id)');
$ids = $query->getResult();
```

Get all users who are members of $group.

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE :groupId MEMBER OF
u.groups');
$query->setParameter('groupId', $group);
$ids = $query->getResult();
```

Get all users that have more than 1 phonenumber

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE SIZE(u.phonenumbers) >
1');
$users = $query->getResult();
```

Get all users that have no phonenumber

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.phonenumbers IS
EMPTY');
$users = $query->getResult();
```

Get all instances of a specific type, for use with inheritance hierarchies:

```
<?php
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\
CompanyPerson u WHERE u INSTANCE OF Doctrine\Tests\Models\Company\
CompanyEmployee');
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\
CompanyPerson u WHERE u INSTANCE OF ?1');
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\
CompanyPerson u WHERE u NOT INSTANCE OF ?1');
```

Get all users visible on a given website that have chosen certain gender:

```
<?php
$query = $em->createQuery('SELECT u FROM User u WHERE u.gender IN (SELECT
IDENTITY(agl.gender) FROM Site s JOIN s.activeGenderList agl WHERE s.id = ?1)');
```

The IDENTITY() DQL function also works for composite primary keys

```
<?php
$query = $em->createQuery("SELECT IDENTITY(c.location, 'latitude') AS latitude,
IDENTITY(c.location, 'longitude') AS longitude FROM Checkpoint c WHERE c.user
= ?1");
```

Joins between entities without associations are available, where you can generate an arbitrary join with the following syntax:

```
<?php
$query = $em->createQuery('SELECT u FROM User u JOIN Banlist b WITH u.email = b.email');
```

The differences between WHERE, WITH and HAVING clauses may be confusing.

- WHERE is applied to the results of an entire query
- WITH is applied to a join as an additional condition. For arbitrary joins (SELECT f, b FROM Foo f, Bar b WITH f.id = b.id) the WITH is required, even if it is 1 = 1
- HAVING is applied to the results of a query after aggregation (GROUP BY)

Partial Object Syntax

By default when you run a DQL query in Doctrine and select only a subset of the fields for a given entity, you do not receive objects back. Instead, you receive only arrays as a flat rectangular result set, similar to how you would if you were just using SQL directly and joining some data.

If you want to select partial objects you can use the `partial` DQL keyword:

```
<?php
$query = $em->createQuery('SELECT partial u.{id, username} FROM CmsUser u');
$users = $query->getResult(); // array of partially loaded CmsUser objects
```

You use the partial syntax when joining as well:

```
<?php
$query = $em->createQuery('SELECT partial u.{id, username}, partial a.{id, name}
FROM CmsUser u JOIN u.articles a');
$users = $query->getResult(); // array of partially loaded CmsUser objects
```

NEW Operator Syntax

Using the `NEW` operator you can construct Data Transfer Objects (DTOs) directly from DQL queries.

- When using `SELECT NEW` you don't need to specify a mapped entity.
- You can specify any PHP class, it only requires that the constructor of this class matches the `NEW` statement.
- This approach involves determining exactly which columns you really need, and instantiating a data-transfer object that contains a constructor with those arguments.

If you want to select data-transfer objects you should create a class:

```
<?php
class CustomerDTO
{
    public function __construct($name, $email, $city, $value = null)
    {
        // Bind values to the object properties.
    }
}
```

And then use the `NEW` DQL keyword :

```
<?php
$query = $em->createQuery('SELECT NEW CustomerDTO(c.name, e.email, a.city) FROM
Customer c JOIN c.email e JOIN c.address a');
$users = $query->getResult(); // array of CustomerDTO
```

```
<?php
$query = $em->createQuery('SELECT NEW CustomerDTO(c.name, e.email, a.city,
SUM(o.value)) FROM Customer c JOIN c.email e JOIN c.address a JOIN c.orders o
GROUP BY c');
$users = $query->getResult(); // array of CustomerDTO
```

Note that you can only pass scalar expressions to the constructor.

Using INDEX BY

The INDEX BY construct is nothing that directly translates into SQL but that affects object and array hydration. After each FROM and JOIN clause you specify by which field this class should be indexed in the result. By default a result is incremented by numerical keys starting with 0. However with INDEX BY you can specify any other column to be the key of your result, it really only makes sense with primary or unique fields though:

```
SELECT u.id, u.status, upper(u.name) nameUpper FROM User u INDEX BY u.id
JOIN u.phonenumbers p INDEX BY p.phonenumber
```

Returns an array of the following kind, indexed by both user-id then phonenumber-id:

```
array
  0 =>
    array
      1 =>
        object(stdClass)[299]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser'
(length=33)
          public 'id' => int 1
          ..
      'nameUpper' => string 'ROMANB' (length=6)
  1 =>
    array
      2 =>
        object(stdClass)[298]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser'
(length=33)
          public 'id' => int 2
          ...
      'nameUpper' => string 'JWAGE' (length=5)
```

UPDATE queries

DQL not only allows to select your Entities using field names, you can also execute bulk updates on a set of entities using an DQL-UPDATE query. The Syntax of an UPDATE query works as expected, as the following example shows:

```
UPDATE MyProject\Model\User u SET u.password = 'new' WHERE u.id IN (1, 2, 3)
```

References to related entities are only possible in the WHERE clause and using sub-selects.

> DQL UPDATE statements are ported directly into a Database UPDATE statement and therefore bypass any locking scheme, events and do not increment the version column. Entities that are already loaded into the persistence context will *NOT* be synced with the updated database state. It is recommended to call `EntityManager#clear()` and retrieve new instances of any affected entity.

DELETE queries

DELETE queries can also be specified using DQL and their syntax is as simple as the UPDATE syntax:

```
DELETE MyProject\Model\User u WHERE u.id = 4
```

The same restrictions apply for the reference of related entities.

> DQL DELETE statements are ported directly into a Database DELETE statement and therefore bypass any events and checks for the version column if they are not explicitly added to the WHERE clause of the query. Additionally Deletes of specified entities are *NOT* cascaded to related entities even if specified in the metadata.

Comments in queries

We can use comments with the SQL syntax of comments.

SELECT u FROM MyProject\Model\User u -- my comment WHERE u.age > 20 -- comment at the end of a line

Functions, Operators, Aggregates

It is possible to wrap both fields and identification values into aggregation and DQL functions. Numerical fields can be part of computations using mathematical operations.

DQL Functions

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

- IDENTITY(single_association_path_expression [, fieldMapping]) - Retrieve the foreign key column of association of the owning side
- ABS(arithmetic_expression)
- CONCAT(str1, str2)
- CURRENT_DATE() - Return the current date

- CURRENT_TIME() - Returns the current time
- CURRENT_TIMESTAMP() - Returns a timestamp of the current date and time.
- LENGTH(str) - Returns the length of the given string
- LOCATE(needle, haystack [, offset]) - Locate the first occurrence of the substring in the string.
- LOWER(str) - returns the string lowercased.
- MOD(a, b) - Return a MOD b.
- SIZE(collection) - Return the number of elements in the specified collection
- SQRT(q) - Return the square-root of q.
- SUBSTRING(str, start [, length]) - Return substring of given string.
- TRIM([LEADING \ BOTH] ['trchar' FROM] str) - Trim the string by the given trim char, defaults to whitespaces.
- UPPER(str) - Return the upper-case of the given string.
- DATE_ADD(date, value, unit) - Add the given time to a given date. (Supported units are SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, YEAR)
- DATE_SUB(date, value, unit) - Subtract the given time from a given date. (Supported units are SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, YEAR)
- DATE_DIFF(date1, date2) - Calculate the difference in days between date1-date2.

Arithmetic operators

You can do math in DQL using numeric values, for example:

```
SELECT person.salary * 1.5 FROM CompanyPerson person WHERE person.salary <
100000
```

Aggregate Functions

The following aggregate functions are allowed in SELECT and GROUP BY clauses: AVG, COUNT, MIN, MAX, SUM

Other Expressions

DQL offers a wide-range of additional expressions that are known from SQL, here is a list of all the supported constructs:

- `ALL/ANY/SOME` - Used in a WHERE clause followed by a sub-select this works like the equivalent constructs in SQL.
- `BETWEEN a AND b` and `NOT BETWEEN a AND b` can be used to match ranges of arithmetic values.
- `IN (x1, x2, ...)` and `NOT IN (x1, x2, ..)` can be used to match a set of given values.
- `LIKE ..` and `NOT LIKE ..` match parts of a string or text using % as a wildcard.
- `IS NULL` and `IS NOT NULL` to check for null values
- `EXISTS` and `NOT EXISTS` in combination with a sub-select

Adding your own functions to the DQL language

By default DQL comes with functions that are part of a large basis of underlying databases. However you will most likely choose a database platform at the beginning of your project and most likely never change it. For this cases you can easily extend the DQL parser with own specialized platform functions.

You can register custom DQL functions in your ORM Configuration:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($name, $class);
$config->addCustomNumericFunction($name, $class);
$config->addCustomDatetimeFunction($name, $class);

$em = EntityManager::create($dbParams, $config);
```

The functions have to return either a string, numeric or datetime value depending on the registered function type. As an example we will add a MySQL specific FLOOR() functionality. All the given classes have to implement the base class :

```
<?php
namespace MyProject\Query\AST;

use \Doctrine\ORM\Query\AST\Functions\FunctionNode;
use \Doctrine\ORM\Query\Lexer;

class MysqlFloor extends FunctionNode
{
    public $simpleArithmeticExpression;

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'FLOOR(' . $sqlWalker->walkSimpleArithmeticExpression(
            $this->simpleArithmeticExpression
        ) . ')';
    }

    public function parse(\Doctrine\ORM\Query\Parser $parser)
    {
        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);

        $this->simpleArithmeticExpression = $parser-
>SimpleArithmeticExpression();

        $parser->match(Lexer::T_CLOSE_PARENTHESIS);
    }
}
```

We will register the function by calling and can then use it:

```
<?php
$config = $em->getConfiguration();
$config->registerNumericFunction('FLOOR', 'MyProject\Query\MysqlFloor');

$dql = "SELECT FLOOR(person.salary * 1.75) FROM CompanyPerson person";
```

Querying Inherited Classes

This section demonstrates how you can query inherited classes and what type of results to expect.

Single Table

Single Table Inheritance is an inheritance mapping strategy where all classes of a hierarchy are mapped to a single database table. In order to distinguish which row represents which type in the hierarchy a so-called discriminator column is used.

First we need to setup an example set of entities to use. In this scenario it is a generic Person and Employee example:

```
<?php
namespace Entities;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    /**
     * @Id @Column(type="integer")
     * @GeneratedValue
     */
    protected $id;

    /**
     * @Column(type="string", length=50)
     */
    protected $name;

    // ...
}

/**
 * @Entity
 */
class Employee extends Person
{
    /**
     * @Column(type="string", length=50)
     */
    private $department;

    // ...
}
```

First notice that the generated SQL to create the tables for these entities looks like the following:

```
CREATE TABLE Person (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    discr VARCHAR(255) NOT NULL,
    department VARCHAR(50) NOT NULL
)
```

Now when persist a new `Employee` instance it will set the discriminator value for us automatically:

```
<?php
$employee = new \Entities\Employee();
$employee->setName('test');
$employee->setDepartment('testing');
$em->persist($employee);
$em->flush();
```

Now lets run a simple query to retrieve the `Employee` we just created:

```
SELECT e FROM Entities\Employee e WHERE e.name = 'test'
```

If we check the generated SQL you will notice it has some special conditions added to ensure that we will only get back `Employee` entities:

```
SELECT p0_.id AS id0, p0_.name AS name1, p0_.department AS department2,
       p0_.discr AS discr3 FROM Person p0_
WHERE (p0_.name = ?) AND p0_.discr IN ('employee')
```

Class Table Inheritance

Class Table Inheritance is an inheritance mapping strategy where each class in a hierarchy is mapped to several tables: its own table and the tables of all parent classes. The table of a child class is linked to the table of a parent class through a foreign key constraint. Doctrine ORM implements this strategy through the use of a discriminator column in the topmost table of the hierarchy because this is the easiest way to achieve polymorphic queries with Class Table Inheritance.

The example for class table inheritance is the same as single table, you just need to change the inheritance type from `SINGLE_TABLE` to `JOINED`:

```
<?php
/**
 * @Entity
 * @InheritanceType("JOINED")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}
```

Now take a look at the SQL which is generated to create the table, you'll notice some differences:

```
CREATE TABLE Person (
   id INT AUTO_INCREMENT NOT NULL,
   name VARCHAR(50) NOT NULL,
   discr VARCHAR(255) NOT NULL,
   PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Employee (
   id INT NOT NULL,
   department VARCHAR(50) NOT NULL,
   PRIMARY KEY(id)
```

) ENGINE = InnoDB;
ALTER TABLE Employee ADD FOREIGN KEY (id) REFERENCES Person(id) ON DELETE CASCADE

The data is split between two tables

- A foreign key exists between the two tables

Now if were to insert the same `Employee` as we did in the `SINGLE_TABLE` example and run the same example query it will generate different SQL joining the `Person` information automatically for you:

```
SELECT p0_.id AS id0, p0_.name AS name1, e1_.department AS department2,
       p0_.discr AS discr3
FROM Employee e1_ INNER JOIN Person p0_ ON e1_.id = p0_.id
WHERE p0_.name = ?
```

The Query class

An instance of the `Doctrine\ORM\Query` class represents a DQL query. You create a Query instance be calling `EntityManager#createQuery($dql)`, passing the DQL query string. Alternatively you can create an empty `Query` instance and invoke `Query#setDQL($dql)` afterwards. Here are some examples:

```
<?php
// $em instanceof EntityManager

// example1: passing a DQL string
$q = $em->createQuery('select u from MyProject\Model\User u');

// example2: using setDQL
$q = $em->createQuery();
$q->setDQL('select u from MyProject\Model\User u');
```

Query Result Formats

The format in which the result of a DQL SELECT query is returned can be influenced by a so-called `hydration mode`. A hydration mode specifies a particular way in which a SQL result set is transformed. Each hydration mode has its own dedicated method on the Query class. Here they are:

- `Query#getResult()`: Retrieves a collection of objects. The result is either a plain collection of objects (pure) or an array where the objects are nested in the result rows (mixed).
- `Query#getSingleResult()`: Retrieves a single object. If the result contains more than one object, an `NonUniqueResultException` is thrown. If the result contains no objects, an `NoResultException` is thrown. The pure/mixed distinction does not apply.
- `Query#getOneOrNullResult()`: Retrieve a single object. If the result contains more than one object, a `NonUniqueResultException` is thrown. If no object is found null will be returned.
- `Query#getArrayResult()`: Retrieves an array graph (a nested array) that is largely interchangeable with the object graph generated by `Query#getResult()` for read-only

purposes. .. note:: An array graph can differ from the corresponding object graph in certain scenarios due to the difference of the identity semantics between arrays and objects.

- `Query#getScalarResult()`: Retrieves a flat/rectangular result set of scalar values that can contain duplicate data. The pure/mixed distinction does not apply.
- `Query#getSingleScalarResult()`: Retrieves a single scalar value from the result returned by the dbms. If the result contains more than a single scalar value, an exception is thrown. The pure/mixed distinction does not apply.

Instead of using these methods, you can alternatively use the general-purpose method `Query#execute(array $params = array(), $hydrationMode = Query::HYDRATE_OBJECT)`. Using this method you can directly supply the hydration mode as the second parameter via one of the Query constants. In fact, the methods mentioned earlier are just convenient shortcuts for the execute method. For example, the method `Query#getResult()` internally invokes execute, passing in `Query::HYDRATE_OBJECT` as the hydration mode.

The use of the methods mentioned earlier is generally preferred as it leads to more concise code.

Pure and Mixed Results

The nature of a result returned by a DQL SELECT query retrieved through `Query#getResult()` or `Query#getArrayResult()` can be of 2 forms: **pure** and **mixed**. In the previous simple examples, you already saw a pure query result, with only objects. By default, the result type is **pure** but **as soon as scalar values, such as aggregate values or other scalar values that do not belong to an entity, appear in the SELECT part of the DQL query, the result becomes mixed**. A mixed result has a different structure than a pure result in order to accommodate for the scalar values.

A pure result usually looks like this:

$dql = "SELECT u FROM User u"; array [0] => Object [1] => Object [2] => Object ...

A mixed result on the other hand has the following general structure:

```
$dql = "SELECT u, 'some scalar string', count(g.id) AS num FROM User u JOIN
u.groups g GROUP BY u.id";

array
    [0]
        [0] => Object
        [1] => "some scalar string"
        ['num'] => 42
        // ... more scalar values, either indexed numerically or with a name
    [1]
        [0] => Object
        [1] => "some scalar string"
        ['num'] => 42
        // ... more scalar values, either indexed numerically or with a name
```

To better understand mixed results, consider the following DQL query:

```
SELECT u, UPPER(u.name) nameUpper FROM MyProject\Model\User u
```

This query makes use of the `UPPER` DQL function that returns a scalar value and because there is now a scalar value in the SELECT clause, we get a mixed result.

Conventions for mixed results are as follows:

- The object fetched in the FROM clause is always positioned with the key '0'.
- Every scalar without a name is numbered in the order given in the query, starting with 1.
- Every aliased scalar is given with its alias-name as the key. The case of the name is kept.
- If several objects are fetched from the FROM clause they alternate every row.

Here is how the result could look like:

```
array
    array
        [0] => User (Object)
        ['nameUpper'] => "ROMAN"
    array
        [0] => User (Object)
        ['nameUpper'] => "JONATHAN"
    ...
```

And here is how you would access it in PHP code:

```
<?php
foreach ($results as $row) {
    echo "Name: " . $row[0]->getName();
    echo "Name UPPER: " . $row['nameUpper'];
}
```

Fetching Multiple FROM Entities

If you fetch multiple entities that are listed in the FROM clause then the hydration will return the rows iterating the different top-level entities.

```
$dql = "SELECT u, g FROM User u, Group g";

array
    [0] => Object (User)
    [1] => Object (Group)
    [2] => Object (User)
    [3] => Object (Group)
```

Hydration Modes

Each of the Hydration Modes makes assumptions about how the result is returned to user land. You should know about all the details to make best use of the different result formats:

The constants for the different hydration modes are:

- Query::HYDRATE_OBJECT
- Query::HYDRATE_ARRAY
- Query::HYDRATE_SCALAR
- Query::HYDRATE_SINGLE_SCALAR

Object Hydration

Object hydration hydrates the result set into the object graph:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
```

$users = $query->getResult(Query::HYDRATE_OBJECT);

Sometimes the behavior in the object hydrator can be confusing, which is why we are listing as many of the assumptions here for reference:

- Objects fetched in a FROM clause are returned as a Set, that means every object is only ever included in the resulting array once. This is the case even when using JOIN or GROUP BY in ways that return the same row for an object multiple times. If the hydrator sees the same object multiple times, then it makes sure it is only returned once.
- If an object is already in memory from a previous query of any kind, then then the previous object is used, even if the database may contain more recent data. Data from the database is discarded. This even happens if the previous object is still an unloaded proxy.

This list might be incomplete.

Array Hydration

You can run the same query with array hydration and the result set is hydrated into an array that represents the object graph:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_ARRAY);
```

You can use the `getArrayResult()` shortcut as well:

```
<?php

$users = $query->getArrayResult();
```

Scalar Hydration

If you want to return a flat rectangular result set instead of an object graph you can use scalar hydration:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_SCALAR);
echo $users[0]['u_id'];
```

The following assumptions are made about selected fields using Scalar Hydration:

1. Fields from classes are prefixed by the DQL alias in the result. A query of the kind 'SELECT u.name ..' returns a key 'u_name' in the result rows.

Single Scalar Hydration

If you have a query which returns just a single scalar value you can use single scalar hydration:

```
<?php
$query = $em->createQuery('SELECT COUNT(a.id) FROM CmsUser u LEFT JOIN
u.articles a WHERE u.username = ?1 GROUP BY u.id');
$query->setParameter(1, 'jwage');
$numArticles = $query->getResult(Query::HYDRATE_SINGLE_SCALAR);
```

You can use the `getSingleScalarResult()` shortcut as well:

```
<?php
$numArticles = $query->getSingleScalarResult();
```

Custom Hydration Modes

You can easily add your own custom hydration modes by first creating a class which extends `AbstractHydrator`:

```
<?php
namespace MyProject\Hydrators;

use Doctrine\ORM\Internal\Hydration\AbstractHydrator;

class CustomHydrator extends AbstractHydrator
{
    protected function _hydrateAll()
    {
        return $this->_stmt->fetchAll(PDO::FETCH_ASSOC);
    }
}
```

Next you just need to add the class to the ORM configuration:

```
<?php
$em->getConfiguration()->addCustomHydrationMode('CustomHydrator', 'MyProject\
Hydrators\CustomHydrator');
```

Now the hydrator is ready to be used in your queries:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$results = $query->getResult('CustomHydrator');
```

Iterating Large Result Sets

There are situations when a query you want to execute returns a very large result-set that needs to be processed. All the previously described hydration modes completely load a result-set into memory which might not be feasible with large result sets. See the Batch Processing section on details how to iterate large result sets.

Functions

The following methods exist on the `AbstractQuery` which both `Query` and `NativeQuery` extend from.

Parameters

Prepared Statements that use numerical or named wildcards require additional parameters to be executable against the database. To pass parameters to the query the following methods can be used:

- `AbstractQuery::setParameter($param, $value)` - Set the numerical or named wildcard to the given value.
- `AbstractQuery::setParameters(array $params)` - Set an array of parameter key-value pairs.

- `AbstractQuery::getParameter($param)`
- `AbstractQuery::getParameters()`

Both named and positional parameters are passed to these methods without their ? or : prefix.

Cache related API

You can cache query results based either on all variables that define the result (SQL, Hydration Mode, Parameters and Hints) or on user-defined cache keys. However by default query results are not cached at all. You have to enable the result cache on a per query basis. The following example shows a complete workflow using the Result Cache API:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.id = ?
1');
$query->setParameter(1, 12);

$query->setResultCacheDriver(new ApcCache());

$query->useResultCache(true)
      ->setResultCacheLifeTime($seconds = 3600);

$result = $query->getResult(); // cache miss

$query->expireResultCache(true);
$result = $query->getResult(); // forced expire, cache miss

$query->setResultCacheId('my_query_result');
$result = $query->getResult(); // saved in given result cache id.

// or call useResultCache() with all parameters:
$query->useResultCache(true, $seconds = 3600, 'my_query_result');
$result = $query->getResult(); // cache hit!

// Introspection
$queryCacheProfile = $query->getQueryCacheProfile();
$cacheDriver = $query->getResultCacheDriver();
$lifetime = $query->getLifetime();
$key = $query->getCacheKey();
```

You can set the Result Cache Driver globally on the `Doctrine\ORM\Configuration` instance so that it is passed to every `Query` and `NativeQuery` instance.

Query Hints

You can pass hints to the query parser and hydrators by using the `AbstractQuery::setHint($name, $value)` method. Currently there exist mostly internal query hints that are not be consumed in userland. However the following few hints are to be used in userland:

- Query::HINT_FORCE_PARTIAL_LOAD - Allows to hydrate objects although not all their columns are fetched. This query hint can be used to handle memory consumption problems with large result-sets that contain char or binary data. Doctrine has no way of implicitly reloading this data. Partially loaded objects have to be passed to `EntityManager::refresh()` if they are to be reloaded fully from the database.

- Query::HINT_REFRESH - This query is used internally by `EntityManager::refresh()` and can be used in userland as well. If you specify this hint and a query returns the data for an entity that is already managed by the UnitOfWork, the fields of the existing entity will be refreshed. In normal operation a result-set that loads data of an already existing entity is discarded in favor of the already existing entity.
- Query::HINT_CUSTOM_TREE_WALKERS - An array of additional `Doctrine\ORM\Query\TreeWalker` instances that are attached to the DQL query parsing process.

Query Cache (DQL Query Only)

Parsing a DQL query and converting it into a SQL query against the underlying database platform obviously has some overhead in contrast to directly executing Native SQL queries. That is why there is a dedicated Query Cache for caching the DQL parser results. In combination with the use of wildcards you can reduce the number of parsed queries in production to zero.

The Query Cache Driver is passed from the `Doctrine\ORM\Configuration` instance to each `Doctrine\ORM\Query` instance by default and is also enabled by default. This also means you don't regularly need to fiddle with the parameters of the Query Cache, however if you do there are several methods to interact with it:

- `Query::setQueryCacheDriver($driver)` - Allows to set a Cache instance
- `Query::setQueryCacheLifeTime($seconds = 3600)` - Set lifetime of the query caching.
- `Query::expireQueryCache($bool)` - Enforce the expiring of the query cache if set to true.
- `Query::getExpireQueryCache()`
- `Query::getQueryCacheDriver()`
- `Query::getQueryCacheLifeTime()`

First and Max Result Items (DQL Query Only)

You can limit the number of results returned from a DQL query as well as specify the starting offset, Doctrine then uses a strategy of manipulating the select query to return only the requested number of results:

- `Query::setMaxResults($maxResults)`
- `Query::setFirstResult($offset)`

If your query contains a fetch-joined collection specifying the result limit methods are not working as you would expect. Set Max Results restricts the number of database result rows, however in the case of fetch-joined collections one root entity might appear in many rows, effectively hydrating less than the specified number of results.

Temporarily change fetch mode in DQL

While normally all your associations are marked as lazy or extra lazy you will have cases where you are using DQL and don't want to fetch join a second, third or fourth level of entities into your result, because of the increased cost of the SQL JOIN. You can mark a many-to-one or one-to-one association as fetched temporarily to batch fetch these entities using a WHERE .. IN query.

```
<?php
$query = $em->createQuery("SELECT u FROM MyProject\User u");
$query->setFetchMode("MyProject\User", "address", \Doctrine\ORM\Mapping\
ClassMetadata::FETCH_EAGER);
$query->execute();
```

Given that there are 10 users and corresponding addresses in the database the executed queries will look something like:

SELECT * FROM users; SELECT * FROM address WHERE id IN (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Changing the fetch mode during a query mostly makes sense for one-to-one and many-to-one relations. In that case, all the necessary IDs are available after the root entity (`user` in the above example) has been loaded. So, one query per association can be executed to fetch all the referred-to entities (`address`).

For one-to-many relations, changing the fetch mode to eager will cause to execute one query **for every root entity loaded**. This gives no improvement over the `lazy` fetch mode which will also initialize the associations on a one-by-one basis once they are accessed.

EBNF

The following context-free grammar, written in an EBNF variant, describes the Doctrine Query Language. You can consult this grammar whenever you are unsure about what is possible with DQL or what the correct syntax for a particular query should be.

Document syntax:

- non-terminals begin with an upper case character
- terminals begin with a lower case character
- parentheses (...) are used for grouping
- square brackets [...] are used for defining an optional part, e.g. zero or one time
- curly brackets {...} are used for repetition, e.g. zero or more times
- double quotation marks "..." define a terminal string
- a vertical bar \| represents an alternative

Terminals

- identifier (name, email, ...) must match `[a-z_][a-z0-9_]*`
- fully_qualified_name (Doctrine\Tests\Models\CMS\CmsUser) matches PHP's fully qualified class names
- aliased_name (CMS:CmsUser) uses two identifiers, one for the namespace alias and one for the class inside it
- string ('foo', 'bar''s house', '%ninja%', ...)
- char ('/', '\\', ' ', ...)
- integer (-1, 0, 1, 34, ...)
- float (-0.23, 0.007, 1.245342E+8, ...)
- boolean (false, true)

Query Language

```
QueryLanguage ::= SelectStatement | UpdateStatement | DeleteStatement
```

Statements

```
SelectStatement ::= SelectClause FromClause [WhereClause] [GroupByClause]
[HavingClause] [OrderByClause]
UpdateStatement ::= UpdateClause [WhereClause]
DeleteStatement ::= DeleteClause [WhereClause]
```

Identifiers

```
/* Alias Identification usage (the "u" of "u.name") */
IdentificationVariable ::= identifier

/* Alias Identification declaration (the "u" of "FROM User u") */
AliasIdentificationVariable :: = identifier

/* identifier that must be a class name (the "User" of "FROM User u"), possibly
as a fully qualified class name or namespace-aliased */
AbstractSchemaName ::= fully_qualified_name | aliased_name | identifier

/* Alias ResultVariable declaration (the "total" of "COUNT(*) AS total") */
AliasResultVariable = identifier

/* ResultVariable identifier usage of mapped field aliases (the "total" of
"COUNT(*) AS total") */
ResultVariable = identifier

/* identifier that must be a field (the "name" of "u.name") */
/* This is responsible to know if the field exists in Object, no matter if it's
a relation or a simple field */
FieldIdentificationVariable ::= identifier

/* identifier that must be a collection-valued association field (to-many) (the
"Phonenumbers" of "u.Phonenumbers") */
CollectionValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be a single-valued association field (to-one) (the
"Group" of "u.Group") */
SingleValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be an embedded class state field */
EmbeddedClassStateField ::= FieldIdentificationVariable

/* identifier that must be a simple state field (name, email, ...) (the "name"
of "u.name") */
/* The difference between this and FieldIdentificationVariable is only
semantical, because it points to a single field (not mapping to a relation) */
SimpleStateField ::= FieldIdentificationVariable
```

Path Expressions

```
/* "u.Group" or "u.Phonenumbers" declarations */
JoinAssociationPathExpression             ::= IdentificationVariable "."
(CollectionValuedAssociationField | SingleValuedAssociationField)

/* "u.Group" or "u.Phonenumbers" usages */
AssociationPathExpression                 ::= CollectionValuedPathExpression |
SingleValuedAssociationPathExpression
```

```
/* "u.name" or "u.Group" */
SingleValuedPathExpression                ::= StateFieldPathExpression |
SingleValuedAssociationPathExpression

/* "u.name" or "u.Group.name" */
StateFieldPathExpression                  ::= IdentificationVariable "."
StateField

/* "u.Group" */
SingleValuedAssociationPathExpression     ::= IdentificationVariable "."
SingleValuedAssociationField

/* "u.Group.Permissions" */
CollectionValuedPathExpression            ::= IdentificationVariable "."
CollectionValuedAssociationField

/* "name" */
StateField                                ::= {EmbeddedClassStateField "."}*
SimpleStateField
```

Clauses

```
SelectClause        ::= "SELECT" ["DISTINCT"] SelectExpression {","
SelectExpression}*
SimpleSelectClause  ::= "SELECT" ["DISTINCT"] SimpleSelectExpression
UpdateClause        ::= "UPDATE" AbstractSchemaName ["AS"]
AliasIdentificationVariable "SET" UpdateItem {"," UpdateItem}*
DeleteClause        ::= "DELETE" ["FROM"] AbstractSchemaName ["AS"]
AliasIdentificationVariable
FromClause          ::= "FROM" IdentificationVariableDeclaration {","
IdentificationVariableDeclaration}*
SubselectFromClause ::= "FROM" SubselectIdentificationVariableDeclaration {","
SubselectIdentificationVariableDeclaration}*
WhereClause         ::= "WHERE" ConditionalExpression
HavingClause        ::= "HAVING" ConditionalExpression
GroupByClause       ::= "GROUP" "BY" GroupByItem {"," GroupByItem}*
OrderByClause       ::= "ORDER" "BY" OrderByItem {"," OrderByItem}*
Subselect           ::= SimpleSelectClause SubselectFromClause [WhereClause]
[GroupByClause] [HavingClause] [OrderByClause]
```

Items

```
UpdateItem  ::= SingleValuedPathExpression "=" NewValue
OrderByItem ::= (SimpleArithmeticExpression | SingleValuedPathExpression |
ScalarExpression | ResultVariable | FunctionDeclaration) ["ASC" | "DESC"]
GroupByItem ::= IdentificationVariable | ResultVariable |
SingleValuedPathExpression
NewValue    ::= SimpleArithmeticExpression | "NULL"
```

From, Join and Index by

```
IdentificationVariableDeclaration          ::= RangeVariableDeclaration
[IndexBy] {Join}*
SubselectIdentificationVariableDeclaration ::= IdentificationVariableDeclaration
RangeVariableDeclaration                   ::= AbstractSchemaName ["AS"]
AliasIdentificationVariable
JoinAssociationDeclaration                 ::= JoinAssociationPathExpression
["AS"] AliasIdentificationVariable [IndexBy]
```

```
Join                                       ::= ["LEFT" ["OUTER"] | "INNER"]
"JOIN" (JoinAssociationDeclaration | RangeVariableDeclaration) ["WITH"
ConditionalExpression]
IndexBy                                    ::= "INDEX" "BY"
StateFieldPathExpression
```

Select Expressions

```
SelectExpression        ::= (IdentificationVariable | ScalarExpression |
AggregateExpression | FunctionDeclaration | PartialObjectExpression | "("
Subselect ")" | CaseExpression | NewObjectExpression) [["AS"] ["HIDDEN"]
AliasResultVariable]
SimpleSelectExpression  ::= (StateFieldPathExpression | IdentificationVariable |
FunctionDeclaration | AggregateExpression | "(" Subselect ")" |
ScalarExpression) [["AS"] AliasResultVariable]
PartialObjectExpression ::= "PARTIAL" IdentificationVariable "." PartialFieldSet
PartialFieldSet         ::= "{" SimpleStateField {"," SimpleStateField}* "}"
NewObjectExpression     ::= "NEW" AbstractSchemaName "(" NewObjectArg {","
NewObjectArg}* ")"
NewObjectArg            ::= ScalarExpression | "(" Subselect ")"
```

Conditional Expressions

```
ConditionalExpression       ::= ConditionalTerm {"OR" ConditionalTerm}*
ConditionalTerm             ::= ConditionalFactor {"AND" ConditionalFactor}*
ConditionalFactor           ::= ["NOT"] ConditionalPrimary
ConditionalPrimary          ::= SimpleConditionalExpression | "("
ConditionalExpression ")"
SimpleConditionalExpression ::= ComparisonExpression | BetweenExpression |
LikeExpression |
                                InExpression | NullComparisonExpression |
ExistsExpression |
                                EmptyCollectionComparisonExpression |
CollectionMemberExpression |
                                InstanceOfExpression
```

Collection Expressions

```
EmptyCollectionComparisonExpression ::= CollectionValuedPathExpression "IS"
["NOT"] "EMPTY"
CollectionMemberExpression          ::= EntityExpression ["NOT"] "MEMBER" ["OF"]
CollectionValuedPathExpression
```

Literal Values

```
Literal     ::= string | char | integer | float | boolean
InParameter ::= Literal | InputParameter
```

Input Parameter

```
InputParameter      ::= PositionalParameter | NamedParameter
PositionalParameter ::= "?" integer
NamedParameter      ::= ":" string
```

Arithmetic Expressions

```
ArithmeticExpression       ::= SimpleArithmeticExpression | "(" Subselect ")"
SimpleArithmeticExpression ::= ArithmeticTerm {("+" | "-") ArithmeticTerm}*
```

```
ArithmeticTerm            ::= ArithmeticFactor {("*" | "/") ArithmeticFactor}*
ArithmeticFactor          ::= [("+" | "-")] ArithmeticPrimary
ArithmeticPrimary         ::= SingleValuedPathExpression | Literal | "("
SimpleArithmeticExpression ")"
                                | FunctionsReturningNumerics |
AggregateExpression | FunctionsReturningStrings
                                | FunctionsReturningDatetime |
IdentificationVariable | ResultVariable
                                | InputParameter | CaseExpression
```

Scalar and Type Expressions

```
ScalarExpression       ::= SimpleArithmeticExpression | StringPrimary |
DateTimePrimary | StateFieldPathExpression | BooleanPrimary | CaseExpression |
InstanceOfExpression
StringExpression       ::= StringPrimary | ResultVariable | "(" Subselect ")"
StringPrimary          ::= StateFieldPathExpression | string | InputParameter |
FunctionsReturningStrings | AggregateExpression | CaseExpression
BooleanExpression      ::= BooleanPrimary | "(" Subselect ")"
BooleanPrimary         ::= StateFieldPathExpression | boolean | InputParameter
EntityExpression       ::= SingleValuedAssociationPathExpression |
SimpleEntityExpression
SimpleEntityExpression ::= IdentificationVariable | InputParameter
DatetimeExpression     ::= DatetimePrimary | "(" Subselect ")"
DatetimePrimary        ::= StateFieldPathExpression | InputParameter |
FunctionsReturningDatetime | AggregateExpression
```

Parts of CASE expressions are not yet implemented.

Aggregate Expressions

```
AggregateExpression ::= ("AVG" | "MAX" | "MIN" | "SUM" | "COUNT") "("
["DISTINCT"] SimpleArithmeticExpression ")"
```

Case Expressions

```
CaseExpression        ::= GeneralCaseExpression | SimpleCaseExpression |
CoalesceExpression | NullifExpression
GeneralCaseExpression ::= "CASE" WhenClause {WhenClause}* "ELSE"
ScalarExpression "END"
WhenClause            ::= "WHEN" ConditionalExpression "THEN" ScalarExpression
SimpleCaseExpression  ::= "CASE" CaseOperand SimpleWhenClause
{SimpleWhenClause}* "ELSE" ScalarExpression "END"
CaseOperand           ::= StateFieldPathExpression | TypeDiscriminator
SimpleWhenClause      ::= "WHEN" ScalarExpression "THEN" ScalarExpression
CoalesceExpression    ::= "COALESCE" "(" ScalarExpression {","
ScalarExpression}* ")"
NullifExpression      ::= "NULLIF" "(" ScalarExpression "," ScalarExpression ")"
```

Other Expressions

QUANTIFIED/BETWEEN/COMPARISON/LIKE/NULL/EXISTS

```
QuantifiedExpression     ::= ("ALL" | "ANY" | "SOME") "(" Subselect ")"
BetweenExpression        ::= ArithmeticExpression ["NOT"] "BETWEEN"
ArithmeticExpression "AND" ArithmeticExpression
ComparisonExpression     ::= ArithmeticExpression ComparisonOperator
( QuantifiedExpression | ArithmeticExpression )
```

```
InExpression             ::= SingleValuedPathExpression ["NOT"] "IN" "("
(InParameter {"," InParameter}* | Subselect) ")"
InstanceOfExpression     ::= IdentificationVariable ["NOT"] "INSTANCE" ["OF"]
(InstanceOfParameter | "(" InstanceOfParameter {"," InstanceOfParameter}* ")")
InstanceOfParameter      ::= AbstractSchemaName | InputParameter
LikeExpression           ::= StringExpression ["NOT"] "LIKE" StringPrimary
["ESCAPE" char]
NullComparisonExpression ::= (InputParameter | NullIfExpression |
CoalesceExpression | AggregateExpression | FunctionDeclaration |
IdentificationVariable | SingleValuedPathExpression | ResultVariable) "IS"
["NOT"] "NULL"
ExistsExpression         ::= ["NOT"] "EXISTS" "(" Subselect ")"
ComparisonOperator       ::= "=" | "<" | "<=" | "<>" | ">" | ">=" | "!="
```

Functions

```
FunctionDeclaration ::= FunctionsReturningStrings | FunctionsReturningNumerics |
FunctionsReturningDateTime

FunctionsReturningNumerics ::=
        "LENGTH" "(" StringPrimary ")" |
        "LOCATE" "(" StringPrimary "," StringPrimary [","
SimpleArithmeticExpression]")" |
        "ABS" "(" SimpleArithmeticExpression ")" |
        "SQRT" "(" SimpleArithmeticExpression ")" |
        "MOD" "(" SimpleArithmeticExpression "," SimpleArithmeticExpression ")"
|
        "SIZE" "(" CollectionValuedPathExpression ")" |
        "DATE_DIFF" "(" ArithmeticPrimary "," ArithmeticPrimary ")" |
        "BIT_AND" "(" ArithmeticPrimary "," ArithmeticPrimary ")" |
        "BIT_OR" "(" ArithmeticPrimary "," ArithmeticPrimary ")"

FunctionsReturningDateTime ::=
        "CURRENT_DATE" |
        "CURRENT_TIME" |
        "CURRENT_TIMESTAMP" |
        "DATE_ADD" "(" ArithmeticPrimary "," ArithmeticPrimary "," StringPrimary
")" |
        "DATE_SUB" "(" ArithmeticPrimary "," ArithmeticPrimary "," StringPrimary
")"

FunctionsReturningStrings ::=
        "CONCAT" "(" StringPrimary "," StringPrimary ")" |
        "SUBSTRING" "(" StringPrimary "," SimpleArithmeticExpression ","
SimpleArithmeticExpression ")" |
        "TRIM" "(" [["LEADING" | "TRAILING" | "BOTH"] [char] "FROM"]
StringPrimary ")" |
        "LOWER" "(" StringPrimary ")" |
        "UPPER" "(" StringPrimary ")" |
        "IDENTITY" "(" SingleValuedAssociationPathExpression {"," string} ")"
```

https://www.doctrine-project.org/projects/doctrine-orm/en/2.8/reference/query-builder.html#the-querybuilder

The QueryBuilder

A `QueryBuilder` provides an API that is designed for conditionally constructing a DQL query in several steps.

It provides a set of classes and methods that is able to programmatically build queries, and also provides a fluent API. This means that you can change between one methodology to the other as you want, or just pick a preferred one.

> The `QueryBuilder` is not an abstraction of DQL, but merely a tool to dynamically build it. You should still use plain DQL when you can, as it is simpler and more readable. More about this in the edba6c47601cd661531fcb1edfd1652353ec95e0.

Constructing a new QueryBuilder object

The same way you build a normal Query, you build a `QueryBuilder` object. Here is an example of how to build a `QueryBuilder` object:

```
<?php
// $em instanceof EntityManager

// example1: creating a QueryBuilder instance
$qb = $em->createQueryBuilder();
```

An instance of QueryBuilder has several informative methods. One good example is to inspect what type of object the `QueryBuilder` is.

```
<?php
// $qb instanceof QueryBuilder

// example2: retrieving type of QueryBuilder
echo $qb->getType(); // Prints: 0
```

There're currently 3 possible return values for `getType()`:

- `QueryBuilder::SELECT`, which returns value 0
- `QueryBuilder::DELETE`, returning value 1
- `QueryBuilder::UPDATE`, which returns value 2

It is possible to retrieve the associated `EntityManager` of the current `QueryBuilder`, its DQL and also a `Query` object when you finish building your DQL.

```
<?php
// $qb instanceof QueryBuilder

// example3: retrieve the associated EntityManager
$em = $qb->getEntityManager();

// example4: retrieve the DQL string of what was defined in QueryBuilder
$dql = $qb->getDql();

// example5: retrieve the associated Query object with the processed DQL
$q = $qb->getQuery();
```

Internally, `QueryBuilder` works with a DQL cache to increase performance. Any changes that may affect the generated DQL actually modifies the state of `QueryBuilder` to a stage we call STATE_DIRTY. One `QueryBuilder` can be in two different states:

- `QueryBuilder::STATE_CLEAN`, which means DQL haven't been altered since last retrieval or nothing were added since its instantiation
- `QueryBuilder::STATE_DIRTY`, means DQL query must (and will) be processed on next retrieval

Working with QueryBuilder

High level API methods

The most straightforward way to build a dynamic query with the `QueryBuilder` is by taking advantage of Helper methods. For all base code, there is a set of useful methods to simplify a programmer's life. To illustrate how to work with them, here is the same example 6 re-written using `QueryBuilder` helper methods:

```
<?php
// $qb instanceof QueryBuilder

$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name', 'ASC');
```

`QueryBuilder` helper methods are considered the standard way to use the `QueryBuilder`. The `$qb->expr()->*` methods can help you build conditional expressions dynamically. Here is a converted example 8 to suggested way to build queries with dynamic conditions:

```
<?php
// $qb instanceof QueryBuilder

$qb->select(array('u')) // string 'u' is converted to array internally
   ->from('User', 'u')
   ->where($qb->expr()->orX(
       $qb->expr()->eq('u.id', '?1'),
       $qb->expr()->like('u.nickname', '?2')
   ))
   ->orderBy('u.surname', 'ASC');
```

Here is a complete list of helper methods available in `QueryBuilder`:

```
<?php
class QueryBuilder
{
    // Example - $qb->select('u')
    // Example - $qb->select(array('u', 'p'))
    // Example - $qb->select($qb->expr()->select('u', 'p'))
    public function select($select = null);

    // addSelect does not override previous calls to select
    //
    // Example - $qb->select('u');
    //              ->addSelect('p.area_code');
    public function addSelect($select = null);

    // Example - $qb->delete('User', 'u')
    public function delete($delete = null, $alias = null);

    // Example - $qb->update('Group', 'g')
    public function update($update = null, $alias = null);

    // Example - $qb->set('u.firstName', $qb->expr()->literal('Arnold'))
    // Example - $qb->set('u.numChilds', 'u.numChilds + ?1')
    // Example - $qb->set('u.numChilds', $qb->expr()->sum('u.numChilds', '?1'))
    public function set($key, $value);

    // Example - $qb->from('Phonenumber', 'p')
    // Example - $qb->from('Phonenumber', 'p', 'p.id')
    public function from($from, $alias, $indexBy = null);

    // Example - $qb->join('u.Group', 'g', Expr\Join::WITH, $qb->expr()-
>eq('u.status_id', '?1'))
    // Example - $qb->join('u.Group', 'g', 'WITH', 'u.status = ?1')
    // Example - $qb->join('u.Group', 'g', 'WITH', 'u.status = ?1', 'g.id')
    public function join($join, $alias, $conditionType = null, $condition =
null, $indexBy = null);

    // Example - $qb->innerJoin('u.Group', 'g', Expr\Join::WITH, $qb->expr()-
>eq('u.status_id', '?1'))
    // Example - $qb->innerJoin('u.Group', 'g', 'WITH', 'u.status = ?1')
    // Example - $qb->innerJoin('u.Group', 'g', 'WITH', 'u.status = ?1',
'g.id')
    public function innerJoin($join, $alias, $conditionType = null, $condition
= null, $indexBy = null);

    // Example - $qb->leftJoin('u.Phonenumbers', 'p', Expr\Join::WITH, $qb-
>expr()->eq('p.area_code', 55))
    // Example - $qb->leftJoin('u.Phonenumbers', 'p', 'WITH', 'p.area_code =
55')
    // Example - $qb->leftJoin('u.Phonenumbers', 'p', 'WITH', 'p.area_code =
55', 'p.id')
    public function leftJoin($join, $alias, $conditionType = null, $condition =
null, $indexBy = null);

    // NOTE: ->where() overrides all previously set conditions
    //
    // Example - $qb->where('u.firstName = ?1', $qb->expr()->eq('u.surname', '?
2'))
    // Example - $qb->where($qb->expr()->andX($qb->expr()->eq('u.firstName', '?
1'), $qb->expr()->eq('u.surname', '?2')))
    // Example - $qb->where('u.firstName = ?1 AND u.surname = ?2')
    public function where($where);

    // NOTE: ->andWhere() can be used directly, without any ->where() before
    //
    // Example - $qb->andWhere($qb->expr()->orX($qb->expr()->lte('u.age', 40),
'u.numChild = 0'))
    public function andWhere($where);

    // Example - $qb->orWhere($qb->expr()->between('u.id', 1, 10));
    public function orWhere($where);

    // NOTE: -> groupBy() overrides all previously set grouping conditions
    //
    // Example - $qb->groupBy('u.id')
    public function groupBy($groupBy);

    // Example - $qb->addGroupBy('g.name')
    public function addGroupBy($groupBy);

    // NOTE: -> having() overrides all previously set having conditions
    //
    // Example - $qb->having('u.salary >= ?1')
    // Example - $qb->having($qb->expr()->gte('u.salary', '?1'))
    public function having($having);

    // Example - $qb->andHaving($qb->expr()->gt($qb->expr()-
>count('u.numChild'), 0))
    public function andHaving($having);

    // Example - $qb->orHaving($qb->expr()->lte('g.managerLevel', '100'))
    public function orHaving($having);

    // NOTE: -> orderBy() overrides all previously set ordering conditions
    //
    // Example - $qb->orderBy('u.surname', 'DESC')
    public function orderBy($sort, $order = null);
    // Example - $qb->addOrderBy('u.firstName')
    public function addOrderBy($sort, $order = null); // Default $order = 'ASC'
}
```

Binding parameters to your query

Doctrine supports dynamic binding of parameters to your query, similar to preparing queries. You can use both strings and numbers as placeholders, although both have a slightly different syntax. Additionally, you must make your choice: Mixing both styles is not allowed. Binding parameters can simply be achieved as follows:

```
<?php
// $qb instanceof QueryBuilder

$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name', 'ASC')
   ->setParameter(1, 100); // Sets ?1 to 100, and thus we will fetch a user
with u.id = 100
```

You are not forced to enumerate your placeholders as the alternative syntax is available:

```
<?php
// $qb instanceof QueryBuilder

$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = :identifier')
   ->orderBy('u.name', 'ASC')
   ->setParameter('identifier', 100); // Sets :identifier to 100, and thus we
will fetch a user with u.id = 100
```

Note that numeric placeholders start with a ? followed by a number while the named placeholders start with a : followed by a string.

Calling `setParameter()` automatically infers which type you are setting as value. This works for integers, arrays of strings/integers, DateTime instances and for managed entities. If you want to set a type explicitly you can call the third argument to `setParameter()` explicitly. It accepts either a PDO type or a DBAL Type name for conversion.

Even though passing DateTime instance is allowed, it impacts performance
as by default there is an attempt to load metadata for object, and if it's not found,
type is inferred from the original value.

```
<?php

use Doctrine\DBAL\Types\Types;

// prevents attempt to load metadata for date time class, improving performance
$qb->setParameter('date', new \DateTimeImmutable(), Types::DATE_IMMUTABLE)
```

If you've got several parameters to bind to your query, you can also use setParameters() instead of setParameter() with the following syntax:

```
<?php

use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\ORM\Query\Parameter;

// $qb instanceof QueryBuilder

// Query here...
$qb->setParameters(new ArrayCollection([
    new Parameter('1', 'value for ?1'),
    new Parameter('2', 'value for ?2')
]));
```

Getting already bound parameters is easy - simply use the above mentioned syntax with getParameter() or getParameters():

```
<?php
// $qb instanceof QueryBuilder

// See example above
$params = $qb->getParameters();
// $params instanceof \Doctrine\Common\Collections\ArrayCollection

// Equivalent to
$param = $qb->getParameter(1);
// $param instanceof \Doctrine\ORM\Query\Parameter
```

Note: If you try to get a parameter that was not bound yet, getParameter() simply returns NULL.

The API of a Query Parameter is:

```
namespace Doctrine\ORM\Query;

class Parameter
{
    public function getName();
    public function getValue();
    public function getType();
    public function setValue($value, $type = null);
}
```

Limiting the Result

To limit a result the query builder has some methods in common with the Query object which can be retrieved from `EntityManager#createQuery()`.

```
<?php
// $qb instanceof QueryBuilder
$offset = (int)$_GET['offset'];
$limit = (int)$_GET['limit'];

$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('orderBy', 'u.name ASC')
   ->setFirstResult( $offset )
   ->setMaxResults( $limit );
```

Executing a Query

The QueryBuilder is a builder object only - it has no means of actually executing the Query. Additionally a set of parameters such as query hints cannot be set on the QueryBuilder itself. This is why you always have to convert a querybuilder instance into a Query object:

```php
<?php
// $qb instanceof QueryBuilder
$query = $qb->getQuery();

// Set additional Query options
$query->setQueryHint('foo', 'bar');
$query->useResultCache('my_cache_id');

// Execute Query
$result = $query->getResult();
$iterableResult = $query->toIterable();
$single = $query->getSingleResult();
$array = $query->getArrayResult();
$scalar = $query->getScalarResult();
$singleScalar = $query->getSingleScalarResult();
```

The Expr class

To workaround some of the issues that `add()` method may cause, Doctrine created a class that can be considered as a helper for building expressions. This class is called `Expr`, which provides a set of useful methods to help build expressions:

```php
<?php
// $qb instanceof QueryBuilder

// example8: QueryBuilder port of:
// "SELECT u FROM User u WHERE u.id = ? OR u.nickname LIKE ? ORDER BY u.name
ASC" using Expr class
$qb->add('select', new Expr\Select(array('u')))
   ->add('from', new Expr\From('User', 'u'))
   ->add('where', $qb->expr()->orX(
       $qb->expr()->eq('u.id', '?1'),
       $qb->expr()->like('u.nickname', '?2')
   ))
   ->add('orderBy', new Expr\OrderBy('u.name', 'ASC'));
```

Although it still sounds complex, the ability to programmatically create conditions are the main feature of `Expr`. Here it is a complete list of supported helper methods available:

```php
<?php
class Expr
{
    /** Conditional objects **/

    // Example - $qb->expr()->andX($cond1 [, $condN])->add(...)->...
    public function andX($x = null); // Returns Expr\AndX instance

    // Example - $qb->expr()->orX($cond1 [, $condN])->add(...)->...
    public function orX($x = null); // Returns Expr\OrX instance


    /** Comparison objects **/

    // Example - $qb->expr()->eq('u.id', '?1') => u.id = ?1
    public function eq($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->neq('u.id', '?1') => u.id <> ?1
    public function neq($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->lt('u.id', '?1') => u.id < ?1
    public function lt($x, $y); // Returns Expr\Comparison instance
    // Example - $qb->expr()->lte('u.id', '?1') => u.id <= ?1
    public function lte($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->gt('u.id', '?1') => u.id > ?1
    public function gt($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->gte('u.id', '?1') => u.id >= ?1
    public function gte($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->isNull('u.id') => u.id IS NULL
    public function isNull($x); // Returns string

    // Example - $qb->expr()->isNotNull('u.id') => u.id IS NOT NULL
    public function isNotNull($x); // Returns string


    /** Arithmetic objects **/

    // Example - $qb->expr()->prod('u.id', '2') => u.id * 2
    public function prod($x, $y); // Returns Expr\Math instance

    // Example - $qb->expr()->diff('u.id', '2') => u.id - 2
    public function diff($x, $y); // Returns Expr\Math instance

    // Example - $qb->expr()->sum('u.id', '2') => u.id + 2
    public function sum($x, $y); // Returns Expr\Math instance

    // Example - $qb->expr()->quot('u.id', '2') => u.id / 2
    public function quot($x, $y); // Returns Expr\Math instance


    /** Pseudo-function objects **/

    // Example - $qb->expr()->exists($qb2->getDql())
    public function exists($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->all($qb2->getDql())
    public function all($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->some($qb2->getDql())
    public function some($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->any($qb2->getDql())
    public function any($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->not($qb->expr()->eq('u.id', '?1'))
    public function not($restriction); // Returns Expr\Func instance

    // Example - $qb->expr()->in('u.id', array(1, 2, 3))
    // Make sure that you do NOT use something similar to $qb->expr()-
>in('value', array('stringvalue')) as this will cause Doctrine to throw an
Exception.
    // Instead, use $qb->expr()->in('value', array('?1')) and bind your
parameter to ?1 (see section above)
    public function in($x, $y); // Returns Expr\Func instance

    // Example - $qb->expr()->notIn('u.id', '2')
    public function notIn($x, $y); // Returns Expr\Func instance

    // Example - $qb->expr()->like('u.firstname', $qb->expr()->literal('Gui%'))
    public function like($x, $y); // Returns Expr\Comparison instance
    // Example - $qb->expr()->notLike('u.firstname', $qb->expr()->literal('Gui
%'))
    public function notLike($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->between('u.id', '1', '10')
    public function between($val, $x, $y); // Returns Expr\Func


    /** Function objects **/

    // Example - $qb->expr()->trim('u.firstname')
    public function trim($x); // Returns Expr\Func

    // Example - $qb->expr()->concat('u.firstname', $qb->expr()->concat($qb-
>expr()->literal(' '), 'u.lastname'))
    public function concat($x, $y); // Returns Expr\Func

    // Example - $qb->expr()->substring('u.firstname', 0, 1)
    public function substring($x, $from, $len); // Returns Expr\Func

    // Example - $qb->expr()->lower('u.firstname')
    public function lower($x); // Returns Expr\Func

    // Example - $qb->expr()->upper('u.firstname')
    public function upper($x); // Returns Expr\Func

    // Example - $qb->expr()->length('u.firstname')
    public function length($x); // Returns Expr\Func

    // Example - $qb->expr()->avg('u.age')
    public function avg($x); // Returns Expr\Func

    // Example - $qb->expr()->max('u.age')
    public function max($x); // Returns Expr\Func

    // Example - $qb->expr()->min('u.age')
    public function min($x); // Returns Expr\Func

    // Example - $qb->expr()->abs('u.currentBalance')
    public function abs($x); // Returns Expr\Func

    // Example - $qb->expr()->sqrt('u.currentBalance')
    public function sqrt($x); // Returns Expr\Func

    // Example - $qb->expr()->count('u.firstname')
    public function count($x); // Returns Expr\Func

    // Example - $qb->expr()->countDistinct('u.surname')
    public function countDistinct($x); // Returns Expr\Func
}
```

Adding a Criteria to a Query

You can also add a Working with Associations to a QueryBuilder by using `addCriteria`:

```
<?php
use Doctrine\Common\Collections\Criteria;
// ...

$criteria = Criteria::create()
    ->orderBy(['firstName', 'ASC']);

// $qb instanceof QueryBuilder
```

```
$qb->addCriteria($criteria);
// then execute your query like normal
```

Low Level API

Now we will describe the low level method of creating queries. It may be useful to work at this level for optimization purposes, but most of the time it is preferred to work at a higher level of abstraction.

All helper methods in `QueryBuilder` actually rely on a single one: `add()`. This method is responsible of building every piece of DQL. It takes 3 parameters: `$dqlPartName`, `$dqlPart` and `$append` (default=false)

- `$dqlPartName`: Where the `$dqlPart` should be placed. Possible values: select, from, where, groupBy, having, orderBy
- `$dqlPart`: What should be placed in `$dqlPartName`. Accepts a string or any instance of `Doctrine\ORM\Query\Expr\*`
- `$append`: Optional flag (default=false) if the `$dqlPart` should override all previously defined items in `$dqlPartName` or not (no effect on the `where` and `having` DQL query parts, which always override all previously defined items)
-

```
<?php
// $qb instanceof QueryBuilder

// example6: how to define:
// "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name ASC"
// using QueryBuilder string support
$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('where', 'u.id = ?1')
   ->add('orderBy', 'u.name ASC');
```

Expr* classes

When you call `add()` with string, it internally evaluates to an instance of `Doctrine\ORM\Query\Expr\Expr\*` class. Here is the same query of example 6 written using `Doctrine\ORM\Query\Expr\Expr\*` classes:

```
<?php
// $qb instanceof QueryBuilder

// example7: how to define:
// "SELECT u FROM User u WHERE u.id = ? ORDER BY u.name ASC"
// using QueryBuilder using Expr\* instances
$qb->add('select', new Expr\Select(array('u')))
   ->add('from', new Expr\From('User', 'u'))
   ->add('where', new Expr\Comparison('u.id', '=', '?1'))
   ->add('orderBy', new Expr\OrderBy('u.name', 'ASC'));
```

Native SQL

With `NativeQuery` you can execute native SELECT SQL statements and map the results to Doctrine entities or any other result format supported by Doctrine.

In order to make this mapping possible, you need to describe to Doctrine what columns in the result map to which entity property. This description is represented by a `ResultSetMapping` object.

With this feature you can map arbitrary SQL code to objects, such as highly vendor-optimized SQL or stored-procedures.

Writing `ResultSetMapping` from scratch is complex, but there is a convenience wrapper around it called a `ResultSetMappingBuilder`. It can generate the mappings for you based on Entities and even generates the `SELECT` clause based on this information for you.

> If you want to execute DELETE, UPDATE or INSERT statements the Native SQL API cannot be used and will probably throw errors. Use `EntityManager#getConnection()` to access the native database connection and call the `executeUpdate()` method for these queries.

The NativeQuery class

To create a `NativeQuery` you use the method `EntityManager#createNativeQuery($sql, $resultSetMapping)`. As you can see in the signature of this method, it expects 2 ingredients: The SQL you want to execute and the `ResultSetMapping` that describes how the results will be mapped.

Once you obtained an instance of a `NativeQuery`, you can bind parameters to it with the same API that `Query` has and execute it.

```
<?php
use Doctrine\ORM\Query\ResultSetMapping;

$rsm = new ResultSetMapping();
// build rsm here

$query = $entityManager->createNativeQuery('SELECT id, name, discr FROM users
WHERE name = ?', $rsm);
$query->setParameter(1, 'romanb');

$users = $query->getResult();
```

ResultSetMappingBuilder

An easy start into ResultSet mapping is the `ResultSetMappingBuilder` object. This has several benefits:

- The builder takes care of automatically updating your `ResultSetMapping` when the fields or associations change on the metadata of an entity.
- You can generate the required `SELECT` expression for a builder by converting it to a string.
- The API is much simpler than the usual `ResultSetMapping` API.

One downside is that the builder API does not yet support entities with inheritance hierarchies.

```
<?php

use Doctrine\ORM\Query\ResultSetMappingBuilder;

$sql = "SELECT u.id, u.name, a.id AS address_id, a.street, a.city " .
       "FROM users u INNER JOIN address a ON u.address_id = a.id";

$rsm = new ResultSetMappingBuilder($entityManager);
$rsm->addRootEntityFromClassMetadata('MyProject\User', 'u');
$rsm->addJoinedEntityFromClassMetadata('MyProject\Address', 'a', 'u', 'address',
array('id' => 'address_id'));
```

The builder extends the `ResultSetMapping` class and as such has all the functionality of it as well.

The `SELECT` clause can be generated from a `ResultSetMappingBuilder`. You can either cast the builder object to `(string)` and the DQL aliases are used as SQL table aliases or use the `generateSelectClause($tableAliases)` method and pass a mapping from DQL alias (key) to SQL alias (value)

```
<?php

$selectClause = $rsm->generateSelectClause(array(
    'u' => 't1',
    'g' => 't2'
));
$sql = "SELECT " . $selectClause . " FROM users t1 JOIN groups t2 ON t1.group_id
= t2.id";
```

The ResultSetMapping

Understanding the `ResultSetMapping` is the key to using a `NativeQuery`. A Doctrine result can contain the following components:

- Entity results. These represent root result elements.
- Joined entity results. These represent joined entities in associations of root entity results.
- Field results. These represent a column in the result set that maps to a field of an entity. A field result always belongs to an entity result or joined entity result.
- Scalar results. These represent scalar values in the result set that will appear in each result row. Adding scalar results to a ResultSetMapping can also cause the overall result to become **mixed** (see DQL - Doctrine Query Language) if the same ResultSetMapping also contains entity results.
- Meta results. These represent columns that contain meta-information, such as foreign keys and discriminator columns. When querying for objects (`getResult()`), all meta columns of root entities or joined entities must be present in the SQL query and mapped accordingly using `ResultSetMapping#addMetaResult`.

It might not surprise you that Doctrine uses `ResultSetMapping` internally when you create DQL queries. As the query gets parsed and transformed to SQL, Doctrine fills a `ResultSetMapping` that describes how the results should be processed by the hydration routines.

We will now look at each of the result types that can appear in a ResultSetMapping in detail.

Entity results

An entity result describes an entity type that appears as a root element in the transformed result. You add an entity result through `ResultSetMapping#addEntityResult()`. Let's take a look at the method signature in detail:

```
<?php
/**
 * Adds an entity result to this ResultSetMapping.
 *
 * @param string $class The class name of the entity.
 * @param string $alias The alias for the class. The alias must be unique among
all entity
 *                      results or joined entity results within this
ResultSetMapping.
 */
public function addEntityResult($class, $alias)
```

The first parameter is the fully qualified name of the entity class. The second parameter is some arbitrary alias for this entity result that must be unique within a `ResultSetMapping`. You use this alias to attach field results to the entity result. It is very similar to an identification variable that you use in DQL to alias classes or relationships.

An entity result alone is not enough to form a valid `ResultSetMapping`. An entity result or joined entity result always needs a set of field results, which we will look at soon.

Joined entity results

A joined entity result describes an entity type that appears as a joined relationship element in the transformed result, attached to a (root) entity result. You add a joined entity result through `ResultSetMapping#addJoinedEntityResult()`. Let's take a look at the method signature in detail:

```
<?php
/**
 * Adds a joined entity result.
 *
 * @param string $class The class name of the joined entity.
 * @param string $alias The unique alias to use for the joined entity.
 * @param string $parentAlias The alias of the entity result that is the parent
of this joined result.
 * @param object $relation The association field that connects the parent entity
result with the joined entity result.
 */
public function addJoinedEntityResult($class, $alias, $parentAlias, $relation)
```

The first parameter is the class name of the joined entity. The second parameter is an arbitrary alias for the joined entity that must be unique within the `ResultSetMapping`. You use this alias to attach field results to the entity result. The third parameter is the alias of the entity result that is the parent type of the joined relationship. The fourth and last parameter is the name of the field on the parent entity result that should contain the joined entity result.

Field results

A field result describes the mapping of a single column in a SQL result set to a field in an entity. As such, field results are inherently bound to entity results. You add a field result through `ResultSetMapping#addFieldResult()`. Again, let's examine the method signature in detail:

```
<?php
/**
 * Adds a field result that is part of an entity result or joined entity result.
 *
 * @param string $alias The alias of the entity result or joined entity result.
 * @param string $columnName The name of the column in the SQL result set.
 * @param string $fieldName The name of the field on the (joined) entity.
 */
public function addFieldResult($alias, $columnName, $fieldName)
```

The first parameter is the alias of the entity result to which the field result will belong. The second parameter is the name of the column in the SQL result set. Note that this name is case sensitive, i.e. if you use a native query against Oracle it must be all uppercase. The third parameter is the name of the field on the entity result identified by `$alias` into which the value of the column should be set.

Scalar results

A scalar result describes the mapping of a single column in a SQL result set to a scalar value in the Doctrine result. Scalar results are typically used for aggregate values but any column in the SQL result set can be mapped as a scalar value. To add a scalar result use `ResultSetMapping#addScalarResult()`. The method signature in detail:

```
<?php
/**
 * Adds a scalar result mapping.
 *
 * @param string $columnName The name of the column in the SQL result set.
 * @param string $alias The result alias with which the scalar result should be
placed in the result structure.
 */
public function addScalarResult($columnName, $alias)
```

The first parameter is the name of the column in the SQL result set and the second parameter is the result alias under which the value of the column will be placed in the transformed Doctrine result.

Meta results

A meta result describes a single column in a SQL result set that is either a foreign key or a discriminator column. These columns are essential for Doctrine to properly construct objects out of SQL result sets. To add a column as a meta result use `ResultSetMapping#addMetaResult()`. The method signature in detail:

```
<?php
/**
 * Adds a meta column (foreign key or discriminator column) to the result set.
 *
 * @param string  $alias
 * @param string  $columnAlias
 * @param string  $columnName
 * @param boolean $isIdentifierColumn
 */
public function addMetaResult($alias, $columnAlias, $columnName,
$isIdentifierColumn = false)
```

The first parameter is the alias of the entity result to which the meta column belongs. A meta result column (foreign key or discriminator column) always belongs to an entity result. The second parameter is the column alias/name of the column in the SQL result set and the third parameter is the column name used in the mapping. The fourth parameter should be set to true in case the primary key of the entity is the foreign key you're adding.

Discriminator Column

When joining an inheritance tree you have to give Doctrine a hint which meta-column is the discriminator column of this tree.

```
<?php
/**
 * Sets a discriminator column for an entity result or joined entity result.
 * The discriminator column will be used to determine the concrete class name to
 * instantiate.
 *
 * @param string $alias The alias of the entity result or joined entity result
the discriminator
 *                        column should be used for.
 * @param string $discrColumn The name of the discriminator column in the SQL
result set.
 */
public function setDiscriminatorColumn($alias, $discrColumn)
```

Examples

Understanding a ResultSetMapping is probably easiest through looking at some examples.

First a basic example that describes the mapping of a single entity.

```
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User owns no associations.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');

$query = $this->_em->createNativeQuery('SELECT id, name FROM users WHERE name
= ?', $rsm);
$query->setParameter(1, 'romanb');

$users = $query->getResult();
```

The result would look like this:

```
array(
    [0] => User (Object)
)
```

Note that this would be a partial object if the entity has more fields than just id and name. In the example above the column and field names are identical but that is not necessary, of course. Also note that the query string passed to createNativeQuery is **real native SQL**. Doctrine does not touch this SQL in any way.

In the previous basic example, a User had no relations and the table the class is mapped to owns no foreign keys. The next example assumes User has a unidirectional or bidirectional one-to-one association to a CmsAddress, where the User is the owning side and thus owns the foreign key.

```
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User owns an association to an Address but the Address is not loaded in the
query.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addMetaResult('u', 'address_id', 'address_id');

$query = $this->_em->createNativeQuery('SELECT id, name, address_id FROM users
WHERE name = ?', $rsm);
$query->setParameter(1, 'romanb');

$users = $query->getResult();
```

Foreign keys are used by Doctrine for lazy-loading purposes when querying for objects. In the previous example, each user object in the result will have a proxy (a ghost) in place of the address that contains the address_id. When the ghost proxy is accessed, it loads itself based on this key.

Consequently, associations that are *fetch-joined* do not require the foreign keys to be present in the SQL result set, only associations that are lazy.

```
<?php
// Equivalent DQL query: "select u from User u join u.address a WHERE u.name = ?
1"
// User owns association to an Address and the Address is loaded in the query.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addJoinedEntityResult('Address' , 'a', 'u', 'address');
$rsm->addFieldResult('a', 'address_id', 'id');
$rsm->addFieldResult('a', 'street', 'street');
$rsm->addFieldResult('a', 'city', 'city');

$sql = 'SELECT u.id, u.name, a.id AS address_id, a.street, a.city FROM users u '
.
       'INNER JOIN address a ON u.address_id = a.id WHERE u.name = ?';
$query = $this->_em->createNativeQuery($sql, $rsm);
$query->setParameter(1, 'romanb');

$users = $query->getResult();
```

In this case the nested entity `Address` is registered with the `ResultSetMapping#addJoinedEntityResult` method, which notifies Doctrine that this entity is not hydrated at the root level, but as a joined entity somewhere inside the object graph. In this case we specify the alias 'u' as third parameter and `address` as fourth parameter, which means the `Address` is hydrated into the `User::$address` property.

If a fetched entity is part of a mapped hierarchy that requires a discriminator column, this column must be present in the result set as a meta column so that Doctrine can create the appropriate concrete type. This is shown in the following example where we assume that there are one or more subclasses that extend User and either Class Table Inheritance or Single Table Inheritance is used to map the hierarchy (both use a discriminator column).

```
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User is a mapped base class for other classes. User owns no associations.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addMetaResult('u', 'discr', 'discr'); // discriminator column
$rsm->setDiscriminatorColumn('u', 'discr');

$query = $this->_em->createNativeQuery('SELECT id, name, discr FROM users WHERE
name = ?', $rsm);
$query->setParameter(1, 'romanb');

$users = $query->getResult();
```

Note that in the case of Class Table Inheritance, an example as above would result in partial objects if any objects in the result are actually a subtype of User. When using DQL, Doctrine automatically includes the necessary joins for this mapping strategy but with native SQL it is your responsibility.

Named Native Query

You can also map a native query using a named native query mapping.

To achieve that, you must describe the SQL resultset structure using named native query (and sql resultset mappings if is a several resultset mappings).

Like named query, a named native query can be defined at class level or in a XML or YAML file.

A resultSetMapping parameter is defined in @NamedNativeQuery, it represents the name of a defined @SqlResultSetMapping.

- *PHP*

```
<?php
namespace MyProject\Model;
/**
 * @NamedNativeQueries({
 *      @NamedNativeQuery(
 *          name            = "fetchMultipleJoinsEntityResults",
 *          resultSetMapping= "mappingMultipleJoinsEntityResults",
 *          query           = "SELECT u.id AS u_id, u.name AS u_name, u.status
AS u_status, a.id AS a_id, a.zip AS a_zip, a.country AS a_country,
COUNT(p.phonenumber) AS numphones FROM users u INNER JOIN addresses a ON u.id =
a.user_id INNER JOIN phonenumbers p ON u.id = p.user_id GROUP BY u.id, u.name,
u.status, u.username, a.id, a.zip, a.country ORDER BY u.username"
 *      ),
 * })
 * @SqlResultSetMappings({
 *      @SqlResultSetMapping(
 *          name    = "mappingMultipleJoinsEntityResults",
 *          entities= {
 *              @EntityResult(
 *                  entityClass = "__CLASS__",
 *                  fields      = {
 *                      @FieldResult(name = "id",       column="u_id"),
 *                      @FieldResult(name = "name",     column="u_name"),
 *                      @FieldResult(name = "status",   column="u_status"),
 *                  }
 *              ),
 *              @EntityResult(
 *                  entityClass = "Address",
 *                  fields      = {
 *                      @FieldResult(name = "id",       column="a_id"),
 *                      @FieldResult(name = "zip",      column="a_zip"),
 *                      @FieldResult(name = "country",  column="a_country"),
 *                  }
 *              )
 *          },
 *          columns = {
 *              @ColumnResult("numphones")
 *          }
 *      )
 *})
 */
 class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;

    /** @Column(type="string", length=50, nullable=true) */
    public $status;

    /** @Column(type="string", length=255, unique=true) */
    public $username;

    /** @Column(type="string", length=255) */
    public $name;

    /** @OneToMany(targetEntity="Phonenumber") */
    public $phonenumbers;

    /** @OneToOne(targetEntity="Address") */
    public $address;

    // ....
}
```

Things to note :

- The resultset mapping declares the entities retrieved by this native query. - Each field of the entity is bound to a SQL alias (or column name). - All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. - Field definitions are optional provided that they map to the same column name as the one declared on the class property. - `__CLASS__` is an alias for the mapped class

In the above example, the `fetchJoinedAddress` named query use the joinMapping result set mapping. This mapping returns 2 entities, User and Address, each property is declared and associated to a column name, actually the column name retrieved by the query.

Let's now see an implicit declaration of the property / column.

- *PHP*

```
<?php
```

```
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name                = "findAll",
     *          resultSetMapping    = "mappingFindAll",
     *          query               = "SELECT * FROM addresses"
     *      ),
     * })
     * @SqlResultSetMappings({
     *      @SqlResultSetMapping(
     *          name    = "mappingFindAll",
     *          entities= {
     *              @EntityResult(
     *                  entityClass = "Address"
     *              )
     *          }
     *      )
     * })
     */
   class Address
   {
        /**  @Id @Column(type="integer") @GeneratedValue */
        public $id;

        /** @Column() */
        public $country;

        /** @Column() */
        public $zip;

        /** @Column()*/
        public $city;

        // ....
    }
```

In this example, we only describe the entity member of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the model property is bound to the model_txt column. If the association to a related entity involve a composite primary key, a @FieldResult element should be used for each foreign key column. The @FieldResult name is composed of the property name for the relationship, followed by a dot (.), followed by the name or the field or property of the primary key.

- *PHP*

```
<?php
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name            = "fetchJoinedAddress",
     *          resultSetMapping= "mappingJoinedAddress",
     *          query           = "SELECT u.id, u.name, u.status, a.id AS a_id,
a.country AS a_country, a.zip AS a_zip, a.city AS a_city FROM users u INNER JOIN
addresses a ON u.id = a.user_id WHERE u.username = ?"
     *      ),
     * })
     * @SqlResultSetMappings({
     *      @SqlResultSetMapping(
     *          name    = "mappingJoinedAddress",
     *          entities= {
     *              @EntityResult(
     *                  entityClass = "__CLASS__",
     *                  fields      = {
     *                      @FieldResult(name = "id"),
     *                      @FieldResult(name = "name"),
     *                      @FieldResult(name = "status"),
     *                      @FieldResult(name = "address.id", column = "a_id"),
     *                      @FieldResult(name = "address.zip", column =
"a_zip"),
     *                      @FieldResult(name = "address.city", column =
"a_city"),
     *                      @FieldResult(name = "address.country", column =
"a_country"),
     *                  }
     *              )
     *          }
     *      )
     * })
     */
    class User
    {
        /** @Id @Column(type="integer") @GeneratedValue */
        public $id;

        /** @Column(type="string", length=50, nullable=true) */
        public $status;

        /** @Column(type="string", length=255, unique=true) */
        public $username;

        /** @Column(type="string", length=255) */
        public $name;

        /** @OneToOne(targetEntity="Address") */
        public $address;

        // ....
    }
```

If you retrieve a single entity and if you use the default mapping, you can use the resultClass attribute instead of resultSetMapping:

- *PHP*

```
<?php
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name            = "find-by-id",
     *          resultClass     = "Address",
     *          query           = "SELECT * FROM addresses"
     *      ),
     * })
     */
   class Address
   {
        // ....
   }
```

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the @SqlResultsetMapping through @ColumnResult. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

- *PHP*

```
<?php
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name            = "count",
     *          resultSetMapping= "mappingCount",
     *          query           = "SELECT COUNT(*) AS count FROM addresses"
     *      )
     * })
     * @SqlResultSetMappings({
     *      @SqlResultSetMapping(
     *          name    = "mappingCount",
     *          columns = {
     *              @ColumnResult(
     *                  name = "count"
     *              )
     *          }
     *      )
     * })
     */
   class Address
   {
       // ....
   }
```

Validation of Entities

Written by Benjamin Eberlei

> You don't need to validate your entities in the lifecycle events. Its only one of many options. Of course you can also perform validations in value setters or any other method of your entities that are used in your code.

Entities can register lifecycle event methods with Doctrine that are called on different occasions. For validation we would need to hook into the events called before persisting and updating. Even though we don't support validation out of the box, the implementation is even simpler than in Doctrine 1 and you will get the additional benefit of being able to re-use your validation in any other part of your domain.

Say we have an `Order` with several `OrderLine` instances. We never want to allow any customer to order for a larger sum than they are allowed to:

```
<?php
class Order
{
    public function assertCustomerAllowedBuying()
    {
        $orderLimit = $this->customer->getOrderLimit();
        $amount = 0;
        foreach ($this->orderLines as $line) {
            $amount += $line->getAmount();
        }

        if ($amount > $orderLimit) {
            throw new CustomerOrderLimitExceededException();
        }
    }
}
```

Now this is some pretty important piece of business logic in your code, enforcing it at any time is important so that customers with a unknown reputation don't owe your business too much money.

We can enforce this constraint in any of the metadata drivers. First Annotations:

```
<?php
/**
 * @Entity
 * @HasLifecycleCallbacks
 */
class Order
{
    /**
     * @PrePersist @PreUpdate
     */
    public function assertCustomerAllowedBuying() {}
}
```

In XML Mappings:

```
<doctrine-mapping>
    <entity name="Order">
        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist"
method="assertCustomerallowedBuying" />
            <lifecycle-callback type="preUpdate"
method="assertCustomerallowedBuying" />
        </lifecycle-callbacks>
    </entity>
</doctrine-mapping>
```

YAML needs some little change yet, to allow multiple lifecycle events for one method, this will happen before Beta 1 though.

Now validation is performed whenever you call `EntityManager#persist($order)` or when you call `EntityManager#flush()` and an order is about to be updated. Any Exception that happens in the lifecycle callbacks will be caught by the EntityManager and the current transaction is rolled back.

Of course you can do any type of primitive checks, not null, email-validation, string size, integer and date ranges in your validation callbacks.

```
<?php
class Order
{
    /**
     * @PrePersist @PreUpdate
     */
    public function validate()
    {
        if (!($this->plannedShipDate instanceof DateTime)) {
            throw new ValidateException();
        }

        if ($this->plannedShipDate->format('U') < time()) {
            throw new ValidateException();
        }

        if ($this->customer == null) {
            throw new OrderRequiresCustomerException();
        }
    }
}
```

What is nice about lifecycle events is, you can also re-use the methods at other places in your domain, for example in combination with your form library. Additionally there is no limitation in the number of methods you register on one particular event, i.e. you can register multiple methods for validation in PrePersist or PreUpdate or mix and share them in any combinations between those two events.

There is no limit to what you can and can't validate in PrePersist and PreUpdate as long as you don't create new entity instances. This was already discussed in the previous blog post on the Versionable extension, which requires another type of event called onFlush.

Further readings: Events

Working with DateTime Instances

There are many nitty gritty details when working with PHPs DateTime instances. You have to know their inner workings pretty well not to make mistakes with date handling. This cookbook entry holds several interesting pieces of information on how to work with PHP DateTime instances in ORM.

DateTime changes are detected by Reference

When calling `EntityManager#flush()` Doctrine computes the changesets of all the currently managed entities and saves the differences to the database. In case of object properties (@Column(type=datetime) or @Column(type=object)) these comparisons are always made **BY REFERENCE**. That means the following change will **NOT** be saved into the database:

```
<?php
/** @Entity */
class Article
{
    /** @Column(type="datetime") */
    private $updated;

    public function setUpdated()
    {
        // will NOT be saved in the database
        $this->updated->modify("now");
    }
}
```

The way to go would be:

```
<?php
class Article
{
    public function setUpdated()
    {
        // WILL be saved in the database
        $this->updated = new \DateTime("now");
    }
}
```

Default Timezone Gotcha

By default Doctrine assumes that you are working with a default timezone. Each DateTime instance that is created by Doctrine will be assigned the timezone that is currently the default, either through the `date.timezone` ini setting or by calling `date_default_timezone_set()`.

This is very important to handle correctly if your application runs on different servers or is moved from one to another server (with different timezone settings). You have to make sure that the timezone is the correct one on all this systems.

Handling different Timezones with the DateTime Type

If you first come across the requirement to save different timezones you may be still optimistic about how to manage this mess, however let me crush your expectations fast. There is not a single database out there (supported by Doctrine ORM) that supports timezones correctly. Correctly here means that you can cover all the use-cases that can come up with timezones. If you don't believe me you should read up on Storing DateTime in Databases.

The problem is simple. Not a single database vendor saves the timezone, only the differences to UTC. However with frequent daylight saving and political timezone changes you can have a UTC offset that moves in different offset directions depending on the real location.

The solution for this dilemma is simple. Don't use timezones with DateTime and Doctrine ORM. However there is a workaround that even allows correct date-time handling with timezones:

1. Always convert any DateTime instance to UTC.
2. Only set Timezones for displaying purposes
3. Save the Timezone in the Entity for persistence.

Say we have an application for an international postal company and employees insert events regarding postal-package around the world, in their current timezones. To determine the exact time an event occurred means to save both the UTC time at the time of the booking and the timezone the event happened in.

```
<?php

namespace DoctrineExtensions\DBAL\Types;

use Doctrine\DBAL\Platforms\AbstractPlatform;
use Doctrine\DBAL\Types\ConversionException;
use Doctrine\DBAL\Types\DateTimeType;

class UTCDateTimeType extends DateTimeType
{
    /**
     * @var \DateTimeZone
     */
    private static $utc;

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if ($value instanceof \DateTime) {
            $value->setTimezone(self::getUtc());
        }

        return parent::convertToDatabaseValue($value, $platform);
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        if (null === $value || $value instanceof \DateTime) {
            return $value;
        }

        $converted = \DateTime::createFromFormat(
            $platform->getDateTimeFormatString(),
            $value,
            self::getUtc()
        );

        if (! $converted) {
            throw ConversionException::conversionFailedFormat(
                $value,
                $this->getName(),
                $platform->getDateTimeFormatString()
            );
        }

        return $converted;
    }

    private static function getUtc(): \DateTimeZone
    {
        return self::$utc ?: self::$utc = new \DateTimeZone('UTC');
    }
}
```

This database type makes sure that every DateTime instance is always saved in UTC, relative to the current timezone that the passed DateTime instance has.

To actually use this new type instead of the default `datetime` type, you need to run following code before bootstrapping the ORM:

```
<?php

use Doctrine\DBAL\Types\Type;
use DoctrineExtensions\DBAL\Types\UTCDateTimeType;

Type::overrideType('datetime', UTCDateTimeType::class);
Type::overrideType('datetimetz', UTCDateTimeType::class);
```

To be able to transform these values back into their real timezone you have to save the timezone in a separate field of the entity requiring timezoned datetimes:

```php
<?php
namespace Shipping;

/**
 * @Entity
 */
class Event
{
    /** @Column(type="datetime") */
    private $created;

    /** @Column(type="string") */
    private $timezone;

    /**
     * @var bool
     */
    private $localized = false;

    public function __construct(\DateTime $createDate)
    {
        $this->localized = true;
        $this->created = $createDate;
        $this->timezone = $createDate->getTimeZone()->getName();
    }

    public function getCreated()
    {
        if (!$this->localized) {
            $this->created->setTimeZone(new \DateTimeZone($this->timezone));
        }
        return $this->created;
    }
}
```

This snippet makes use of the previously discussed changeset by reference only property of objects. That means a new DateTime will only be used during updating if the reference changes between retrieval and flush operation. This means we can easily go and modify the instance by setting the previous local timezone.

Mysql Enums

The type system of Doctrine ORM consists of flyweights, which means there is only one instance of any given type. Additionally types do not contain state. Both assumptions make it rather complicated to work with the Enum Type of MySQL that is used quite a lot by developers.

When using Enums with a non-tweaked Doctrine ORM application you will get errors from the Schema-Tool commands due to the unknown database type enum. By default Doctrine does not map the MySQL enum type to a Doctrine type. This is because Enums contain state (their allowed values) and Doctrine types don't.

This cookbook entry shows two possible solutions to work with MySQL enums. But first a word of warning. The MySQL Enum type has considerable downsides:

- Adding new values requires to rebuild the whole table, which can take hours depending on the size.
- Enums are ordered in the way the values are specified, not in their "natural" order.
- Enums validation mechanism for allowed values is not necessarily good, specifying invalid values leads to an empty enum for the default MySQL error settings. You can easily replicate the "allow only some values" requirement in your Doctrine entities.

Solution 1: Mapping to Varchars

You can map ENUMs to varchars. You can register MySQL ENUMs to map to Doctrine varchars. This way Doctrine always resolves ENUMs to Doctrine varchars. It will even detect this match correctly when using SchemaTool update commands.

```
<?php
$conn = $em->getConnection();
$conn->getDatabasePlatform()->registerDoctrineTypeMapping('enum', 'string');
```

In this case you have to ensure that each varchar field that is an enum in the database only gets passed the allowed values. You can easily enforce this in your entities:

```
<?php
/** @Entity */
class Article
{
    const STATUS_VISIBLE = 'visible';
    const STATUS_INVISIBLE = 'invisible';

    /** @Column(type="string") */
    private $status;

    public function setStatus($status)
    {
        if (!in_array($status, array(self::STATUS_VISIBLE,
self::STATUS_INVISIBLE))) {
            throw new \InvalidArgumentException("Invalid status");
        }
        $this->status = $status;
    }
}
```

If you want to actively create enums through the Doctrine Schema-Tool by using the **columnDefinition** attribute.

```
<?php
/** @Entity */
class Article
{
    /** @Column(type="string", columnDefinition="ENUM('visible', 'invisible')")
*/
    private $status;
}
```

In this case however Schema-Tool update will have a hard time not to request changes for this column on each call.

Solution 2: Defining a Type

You can make a stateless ENUM type by creating a type class for each unique set of ENUM values. For example for the previous enum type:

```
<?php
namespace MyProject\DBAL;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

class EnumVisibilityType extends Type
{
    const ENUM_VISIBILITY = 'enumvisibility';
    const STATUS_VISIBLE = 'visible';
    const STATUS_INVISIBLE = 'invisible';

    public function getSQLDeclaration(array $fieldDeclaration, AbstractPlatform
$platform)
    {
        return "ENUM('visible', 'invisible')";
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return $value;
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if (!in_array($value, array(self::STATUS_VISIBLE,
self::STATUS_INVISIBLE))) {
            throw new \InvalidArgumentException("Invalid status");
        }
        return $value;
    }

    public function getName()
    {
        return self::ENUM_VISIBILITY;
    }

    public function requiresSQLCommentHint(AbstractPlatform $platform)
    {
        return true;
    }
}
```

You can register this type with `Type::addType('enumvisibility', 'MyProject\DBAL\EnumVisibilityType');`. Then in your entity you can just use this type:

```
<?php
/** @Entity */
class Article
{
    /** @Column(type="enumvisibility") */
    private $status;
}
```

You can generalize this approach easily to create a base class for enums:

```
<?php
```

```php
namespace MyProject\DBAL;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

abstract class EnumType extends Type
{
    protected $name;
    protected $values = array();

    public function getSQLDeclaration(array $fieldDeclaration, AbstractPlatform
$platform)
    {
        $values = array_map(function($val) { return "'".$val."'"; }, $this-
>values);

        return "ENUM(".implode(", ", $values).")";
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return $value;
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if (!in_array($value, $this->values)) {
            throw new \InvalidArgumentException("Invalid '".$this->name."'
value.");
        }
        return $value;
    }

    public function getName()
    {
        return $this->name;
    }

    public function requiresSQLCommentHint(AbstractPlatform $platform)
    {
        return true;
    }
}
```

With this base class you can define an enum as easily as:

```php
<?php
namespace MyProject\DBAL;

class EnumVisibilityType extends EnumType
{
    protected $name = 'enumvisibility';
    protected $values = array('visible', 'invisible');
}
```

Advanced field value conversion using custom mapping types

Written by Jan Sorgalla

There are several ways to achieve this: converting the value inside the Type class, converting the value on the database-level or a combination of both.

This article describes the third way by implementing the MySQL specific column type Point.

The `Point` type is part of the Spatial extension of MySQL and enables you to store a single location in a coordinate space by using x and y coordinates. You can use the Point type to store a longitude/latitude pair to represent a geographic location.

The entity

We create a simple entity with a field `$point` which holds a value object `Point` representing the latitude and longitude of the position.

The entity class:

```
<?php

namespace Geo\Entity;

/**
 * @Entity
 */
class Location
{
    /**
     * @Column(type="point")
     *
     * @var \Geo\ValueObject\Point
     */
    private $point;

    /**
     * @Column(type="string")
     *
     * @var string
     */
    private $address;

    /**
     * @param \Geo\ValueObject\Point $point
     */
    public function setPoint(\Geo\ValueObject\Point $point)
    {
        $this->point = $point;
    }

    /**
     * @return \Geo\ValueObject\Point
     */
    public function getPoint()
    {
        return $this->point;
    }

    /**
     * @param string $address
     */
    public function setAddress($address)
    {
        $this->address = $address;
    }

    /**
     * @return string
     */
    public function getAddress()
    {
        return $this->address;
    }
}
```

We use the custom type `point` in the `@Column` docblock annotation of the `$point` field. We will create this custom mapping type in the next chapter.

The point class:

```
<?php

namespace Geo\ValueObject;

class Point
{

    /**
     * @param float $latitude
     * @param float $longitude
     */
    public function __construct($latitude, $longitude)
    {
        $this->latitude  = $latitude;
        $this->longitude = $longitude;
    }

    /**
     * @return float
     */
    public function getLatitude()
    {
        return $this->latitude;
    }

    /**
     * @return float
     */
    public function getLongitude()
    {
        return $this->longitude;
    }
}
```

The mapping type

Now we're going to create the `point` type and implement all required methods.

```php
<?php

namespace Geo\Types;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

use Geo\ValueObject\Point;

class PointType extends Type
{
    const POINT = 'point';

    public function getName()
    {
        return self::POINT;
    }

    public function getSQLDeclaration(array $fieldDeclaration, AbstractPlatform
$platform)
    {
        return 'POINT';
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        list($longitude, $latitude) = sscanf($value, 'POINT(%f %f)');

        return new Point($latitude, $longitude);
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if ($value instanceof Point) {
            $value = sprintf('POINT(%F %F)', $value->getLongitude(), $value-
>getLatitude());
        }

        return $value;
    }

    public function canRequireSQLConversion()
    {
        return true;
    }

    public function convertToPHPValueSQL($sqlExpr, AbstractPlatform $platform)
    {
        return sprintf('AsText(%s)', $sqlExpr);
    }

    public function convertToDatabaseValueSQL($sqlExpr, AbstractPlatform
$platform)
    {
        return sprintf('PointFromText(%s)', $sqlExpr);
    }
}
```

We do a 2-step conversion here. In the first step, we convert the `Point` object into a string representation before saving to the database (in the `convertToDatabaseValue` method) and back into an object after fetching the value from the database (in the `convertToPHPValue` method).

The format of the string representation format is called Well-known text (WKT). The advantage of this format is, that it is both human readable and parsable by MySQL.

Internally, MySQL stores geometry values in a binary format that is not
identical to the WKT format. So, we need to let MySQL transform the WKT representation into its internal format.

This is where the `convertToPHPValueSQL` and `convertToDatabaseValueSQL` methods come into play.

This methods wrap a sql expression (the WKT representation of the Point) into MySQL functions ST_PointFromText and ST_AsText which convert WKT strings to and from the internal format of MySQL.

> When using DQL queries, the `convertToPHPValueSQL` and `convertToDatabaseValueSQL` methods only apply to identification variables and path expressions in SELECT clauses. Expressions in WHERE clauses are
> **not** wrapped!
>
> If you want to use Point values in WHERE clauses, you have to implement a user defined function for
> `PointFromText`.

Example usage

```
<?php

// Bootstrapping stuff...
// $em = \Doctrine\ORM\EntityManager::create($connectionOptions, $config);

// Setup custom mapping type
use Doctrine\DBAL\Types\Type;

Type::addType('point', 'Geo\Types\PointType');
$em->getConnection()->getDatabasePlatform()-
>registerDoctrineTypeMapping('point', 'point');

// Store a Location object
use Geo\Entity\Location;
use Geo\ValueObject\Point;

$location = new Location();

$location->setAddress('1600 Amphitheatre Parkway, Mountain View, CA');
$location->setPoint(new Point(37.4220761, -122.0845187));

$em->persist($location);
$em->flush();
$em->clear();

// Fetch the Location object
$query = $em->createQuery("SELECT l FROM Geo\Entity\Location l WHERE l.address =
'1600 Amphitheatre Parkway, Mountain View, CA'");
$location = $query->getSingleResult();

/* @var Geo\ValueObject\Point */
$point = $location->getPoint();
```

Entities in the Session

There are several use-cases to save entities in the session, for example:

1. User object
2. Multi-step forms

To achieve this with Doctrine you have to pay attention to some details to get this working.

Merging entity into an EntityManager

In Doctrine an entity objects has to be managed by an EntityManager to be updateable. Entities saved into the session are not managed in the next request anymore. This means that you have to register these entities with an EntityManager again if you want to change them or use them as part of references between other entities. You can achieve this by calling `EntityManager#merge()`.

For a representative User object the code to get turn an instance from the session into a managed Doctrine object looks like this:

```
<?php
require_once 'bootstrap.php';
$em = GetEntityManager(); // creates an EntityManager

session_start();
if (isset($_SESSION['user']) && $_SESSION['user'] instanceof User) {
    $user = $_SESSION['user'];
    $user = $em->merge($user);
}
```

A frequent mistake is not to get the merged user object from the return value of `EntityManager#merge()`. The entity object passed to merge is not necessarily the same object that is returned from the method.

Serializing entity into the session

Entities that are serialized into the session normally contain references to other entities as well. Think of the user entity has a reference to their articles, groups, photos or many other different entities. If you serialize this object into the session then you don't want to serialize the related entities as well. This is why you should call `EntityManager#detach()` on this object or implement the __sleep() magic method on your entity.

```
<?php
require_once 'bootstrap.php';
$em = GetEntityManager(); // creates an EntityManager

$user = $em->find("User", 1);
$em->detach($user);
$_SESSION['user'] = $user;
```

When you called detach on your objects they get unmanaged with that entity manager. This means you cannot use them as part of write operations during `EntityManager#flush()` anymore in this request.

Metadata Drivers

The heart of an object relational mapper is the mapping information that glues everything together. It instructs the EntityManager how it should behave when dealing with the different entities.

Core Metadata Drivers

Doctrine provides a few different ways for you to specify your metadata:

- **XML files** (XmlDriver)
- **Class DocBlock Annotations** (AnnotationDriver)
- **YAML files** (YamlDriver)
- **PHP Code in files or static functions** (PhpDriver)

Something important to note about the above drivers is they are all an intermediate step to the same end result. The mapping information is populated to `Doctrine\ORM\Mapping\ClassMetadata` instances. So in the end, Doctrine only ever has to work with the API of the `ClassMetadata` class to get mapping information for an entity.

> The populated `ClassMetadata` instances are also cached so in a production environment the parsing and populating only ever happens once. You can configure the metadata cache implementation using the `setMetadataCacheImpl()` method on the `Doctrine\ORM\Configuration` class:

```
<?php
$em->getConfiguration()->setMetadataCacheImpl(new ApcuCache());
```

If you want to use one of the included core metadata drivers you just need to configure it. All the drivers are in the `Doctrine\ORM\Mapping\Driver` namespace:

```
<?php
$driver = new \Doctrine\ORM\Mapping\Driver\XmlDriver('/path/to/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

Implementing Metadata Drivers

In addition to the included metadata drivers you can very easily implement your own. All you need to do is define a class which implements the `Driver` interface:

```
<?php
namespace Doctrine\ORM\Mapping\Driver;

use Doctrine\ORM\Mapping\ClassMetadataInfo;

interface Driver
```

```
{
    /**
     * Loads the metadata for the specified class into the provided container.
     *
     * @param string $className
     * @param ClassMetadataInfo $metadata
     */
    function loadMetadataForClass($className, ClassMetadataInfo $metadata);

    /**
     * Gets the names of all mapped classes known to this driver.
     *
     * @return array The names of all mapped classes known to this driver.
     */
    function getAllClassNames();

    /**
     * Whether the class with the specified name should have its metadata
loaded.
     * This is only the case if it is either mapped as an Entity or a
     * MappedSuperclass.
     *
     * @param string $className
     * @return boolean
     */
    function isTransient($className);
}
```

If you want to write a metadata driver to parse information from some file format we've made your life a little easier by providing the `AbstractFileDriver` implementation for you to extend from:

```
<?php
class MyMetadataDriver extends AbstractFileDriver
{
    /**
     * {@inheritdoc}
     */
    protected $_fileExtension = '.dcm.ext';

    /**
     * {@inheritdoc}
     */
    public function loadMetadataForClass($className, ClassMetadataInfo
$metadata)
    {
        $data = $this->_loadMappingFile($file);

        // populate ClassMetadataInfo instance from $data
    }

    /**
     * {@inheritdoc}
     */
    protected function _loadMappingFile($file)
    {
        // parse contents of $file and return php data structure
    }
}
```

When using the `AbstractFileDriver` it requires that you only have one entity defined per file and the file named after the class described inside where namespace separators are replaced by periods. So if you have an entity named `Entities\User` and you wanted to write a mapping file for your driver above you would need to name the file `Entities.User.dcm.ext` for it to be recognized.

Now you can use your `MyMetadataDriver` implementation by setting it with the `setMetadataDriverImpl()` method:

```
<?php
$driver = new MyMetadataDriver('/path/to/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

ClassMetadata

The last piece you need to know and understand about metadata in Doctrine ORM is the API of the `ClassMetadata` classes. You need to be familiar with them in order to implement your own drivers but more importantly to retrieve mapping information for a certain entity when needed.

You have all the methods you need to manually specify the mapping information instead of using some mapping file to populate it from. The base `ClassMetadataInfo` class is responsible for only data storage and is not meant for runtime use. It does not require that the class actually exists yet so it is useful for describing some entity before it exists and using that information to generate for example the entities themselves. The class `ClassMetadata` extends `ClassMetadataInfo` and adds some functionality required for runtime usage and requires that the PHP class is present and can be autoloaded.

You can read more about the API of the `ClassMetadata` classes in the PHP Mapping chapter.

Getting ClassMetadata Instances

If you want to get the `ClassMetadata` instance for an entity in your project to programmatically use some mapping information to generate some HTML or something similar you can retrieve it through the `ClassMetadataFactory`:

```
<?php
$cmf = $em->getMetadataFactory();
$class = $cmf->getMetadataFor('MyEntityName');
```

Now you can learn about the entity and use the data stored in the `ClassMetadata` instance to get all mapped fields for example and iterate over them:

```
<?php
foreach ($class->fieldMappings as $fieldMapping) {
    echo $fieldMapping['fieldName'] . "\n";
}
```

Best Practices

The best practices mentioned here that affect database design generally refer to best practices when working with Doctrine and do not necessarily reflect best practices for database design in general.

Constrain relationships as much as possible

It is important to constrain relationships as much as possible. This means:

- Impose a traversal direction (avoid bidirectional associations if possible)
- Eliminate nonessential associations

This has several benefits:

- Reduced coupling in your domain model
- Simpler code in your domain model (no need to maintain bidirectionality properly)
- Less work for Doctrine

Avoid composite keys

Even though Doctrine fully supports composite keys it is best not to use them if possible. Composite keys require additional work by Doctrine and thus have a higher probability of errors.

Use events judiciously

The event system of Doctrine is great and fast. Even though making heavy use of events, especially lifecycle events, can have a negative impact on the performance of your application. Thus you should use events judiciously.

Use cascades judiciously

Automatic cascades of the persist/remove/merge/etc. operations are very handy but should be used wisely. Do NOT simply add all cascades to all associations. Think about which cascades actually do make sense for you for a particular association, given the scenarios it is most likely used in.

Don't use special characters

Avoid using any non-ASCII characters in class, field, table or column names. Doctrine itself is not unicode-safe in many places and will not be until PHP itself is fully unicode-aware.

Don't use identifier quoting

Identifier quoting is a workaround for using reserved words that often causes problems in edge cases. Do not use identifier quoting and avoid using reserved words as table or column names.

Initialize collections in the constructor

It is recommended best practice to initialize any business collections in entities in the constructor. Example:

```
<?php
namespace MyProject\Model;
use Doctrine\Common\Collections\ArrayCollection;

class User {
    private $addresses;
    private $articles;

    public function __construct() {
        $this->addresses = new ArrayCollection;
        $this->articles = new ArrayCollection;
    }
}
```

Don't map foreign keys to fields in an entity

Foreign keys have no meaning whatsoever in an object model. Foreign keys are how a relational database establishes relationships. Your object model establishes relationships through object references. Thus mapping foreign keys to object fields heavily leaks details of the relational model into the object model, something you really should not do.

Use explicit transaction demarcation

While Doctrine will automatically wrap all DML operations in a transaction on flush(), it is considered best practice to explicitly set the transaction boundaries yourself. Otherwise every single query is wrapped in a small transaction (Yes, SELECT queries, too) since you can not talk to your database outside of a transaction. While such short transactions for read-only (SELECT) queries generally don't have any noticeable performance impact, it is still preferable to use fewer, well-defined transactions that are established through explicit transaction boundaries.

Pagination

Doctrine ORM ships with a Paginator for DQL queries. It has a very simple API and implements the SPL interfaces `Countable` and `IteratorAggregate`.

<?php use Doctrine\ORM\Tools\Pagination\Paginator; $dql = "SELECT p, c FROM BlogPost p JOIN p.comments c"; $query = $entityManager->createQuery($dql) ->setFirstResult(0) ->setMaxResults(100); $paginator = new Paginator($query, $fetchJoinCollection = true); $c = count($paginator); foreach ($paginator as $post) { echo $post->getHeadline() . "\n"; }
Paginating Doctrine queries is not as simple as you might think in the beginning. If you have complex fetch-join scenarios with one-to-many or many-to-many associations using the default LIMIT functionality of database vendors is not sufficient to get the correct results.

By default the pagination extension does the following steps to compute the correct result:

- Perform a Count query using `DISTINCT` keyword.
- Perform a Limit Subquery with `DISTINCT` to find all ids of the entity in from on the current page.
- Perform a WHERE IN query to get all results for the current page.

This behavior is only necessary if you actually fetch join a to-many collection. You can disable this behavior by setting the `$fetchJoinCollection` flag to `false`; in that case only 2 instead of the 3 queries described are executed. We hope to automate the detection for this in the future.

Tools

Doctrine Console

The Doctrine Console is a Command Line Interface tool for simplifying common administration tasks during the development of a project that uses ORM.

Take a look at the Installation and Configuration chapter for more information how to setup the console command.

Display Help Information

Type `php vendor/bin/doctrine` on the command line and you should see an overview of the available commands or use the --help flag to get information on the available commands. If you want to know more about the use of generate entities for example, you can call:

```
$> php vendor/bin/doctrine orm:generate-entities --help
```

Configuration

Whenever the `doctrine` command line tool is invoked, it can access all Commands that were registered by a developer. There is no auto-detection mechanism at work. The Doctrine binary already registers all the commands that currently ship with Doctrine DBAL and ORM. If you want to use additional commands you have to register them yourself.

All the commands of the Doctrine Console require access to the `EntityManager`. You have to inject it into the console application with `ConsoleRunner::createHelperSet`. Whenever you invoke the Doctrine binary, it searches the current directory for the file `cli-config.php`. This file contains the project-specific configuration.

Here is an example of a the project-specific `cli-config.php`:

```
<?php
use Doctrine\ORM\Tools\Console\ConsoleRunner;

// replace this with the path to your own project bootstrap file.
require_once 'bootstrap.php';

// replace with mechanism to retrieve EntityManager in your app
$entityManager = GetEntityManager();

return ConsoleRunner::createHelperSet($entityManager);
```

You have to adjust this snippet for your specific application or framework and use their facilities to access the Doctrine EntityManager and Connection Resources.

Command Overview

The following Commands are currently available:

- `help` Displays help for a command (?)
- `list` Lists commands
- `dbal:import` Import SQL file(s) directly to Database.
- `dbal:run-sql` Executes arbitrary SQL directly from the command line.
- `orm:clear-cache:metadata` Clear all metadata cache of the various cache drivers.
- `orm:clear-cache:query` Clear all query cache of the various cache drivers.
- `orm:clear-cache:result` Clear result cache of the various cache drivers.
- `orm:convert-d1-schema` Converts Doctrine 1.X schema into a Doctrine 2.X schema.
- `orm:convert-mapping` Convert mapping information between supported formats.
- `orm:ensure-production-settings` Verify that Doctrine is properly configured for a production environment.
- `orm:generate-entities` Generate entity classes and method stubs from your mapping information.
- `orm:generate-proxies` Generates proxy classes for entity classes.
- `orm:generate-repositories` Generate repository classes from your mapping information.
- `orm:run-dql` Executes arbitrary DQL directly from the command line.
- `orm:schema-tool:create` Processes the schema and either create it directly on EntityManager Storage Connection or generate the SQL output.
- `orm:schema-tool:drop` Processes the schema and either drop the database schema of EntityManager Storage Connection or generate the SQL output.
- `orm:schema-tool:update` Processes the schema and either update the database schema of EntityManager Storage Connection or generate the SQL output.

For these commands are also available aliases:

- `orm:convert:d1-schema` is alias for `orm:convert-d1-schema`.
- `orm:convert:mapping` is alias for `orm:convert-mapping`.
- `orm:generate:entities` is alias for `orm:generate-entities`.
- `orm:generate:proxies` is alias for `orm:generate-proxies`.
- `orm:generate:repositories` is alias for `orm:generate-repositories`.

Console also supports auto completion, for example, instead of `orm:clear-cache:query` you can use just `o:c:q`.

Database Schema Generation

SchemaTool can do harm to your database. It will drop or alter tables, indexes, sequences and such. Please use this tool with caution in development and not on a production server. It is meant for helping you develop your Database Schema, but NOT with migrating schema from A to B in production. A safe approach would be generating the SQL on development server and saving it into SQL Migration files that are executed manually on the production server.

SchemaTool assumes your Doctrine Project uses the given database on its own. Update and Drop commands will mess with other tables if they are not related to the current project that is using Doctrine. Please be careful!

To generate your database schema from your Doctrine mapping files you can use the `SchemaTool` class or the `schema-tool` Console Command.

When using the SchemaTool class directly, create your schema using the `createSchema()` method. First create an instance of the `SchemaTool` and pass it an instance of the `EntityManager` that you want to use to create the schema. This method receives an array of `ClassMetadataInfo` instances.

```
<?php
$tool = new \Doctrine\ORM\Tools\SchemaTool($em);
$classes = array(
  $em->getClassMetadata('Entities\User'),
  $em->getClassMetadata('Entities\Profile')
);
$tool->createSchema($classes);
```

To drop the schema you can use the `dropSchema()` method.

```
<?php
$tool->dropSchema($classes);
```

This drops all the tables that are currently used by your metadata model. When you are changing your metadata a lot during development you might want to drop the complete database instead of only the tables of the current model to clean up with orphaned tables.

```
<?php
$tool->dropSchema($classes, \Doctrine\ORM\Tools\SchemaTool::DROP_DATABASE);
```

You can also use database introspection to update your schema easily with the `updateSchema()` method. It will compare your existing database schema to the passed array of `ClassMetadataInfo` instances.

```
<?php
$tool->updateSchema($classes);
```

If you want to use this functionality from the command line you can use the `schema-tool` command.

To create the schema use the `create` command:

```
$ php doctrine orm:schema-tool:create
```

To drop the schema use the `drop` command:

```
$ php doctrine orm:schema-tool:drop
```

If you want to drop and then recreate the schema then use both options:

$ php doctrine orm:schema-tool:drop $ php doctrine orm:schema-tool:create

As you would think, if you want to update your schema use the `update` command:

```
$ php doctrine orm:schema-tool:update
```

All of the above commands also accept a `--dump-sql` option that will output the SQL for the ran operation.

```
$ php doctrine orm:schema-tool:create --dump-sql
```

Before using the orm:schema-tool commands, remember to configure your cli-config.php properly.

Entity Generation

Generate entity classes and method stubs from your mapping information

$ php doctrine orm:generate-entities

$ php doctrine orm:generate-entities --update-entities

$ php doctrine orm:generate-entities –regenerate-entities

This command is not suited for constant usage. It is a little helper and does not support all the mapping edge cases very well. You still have to put work in your entities after using this command.

It is possible to use the EntityGenerator on code that you have already written. It will not be lost. The EntityGenerator will only append new code to your file and will not delete the old code. However this approach may still be prone to error and we suggest you use code repositories such as GIT or SVN to make backups of your code.

It makes sense to generate the entity code if you are using entities as Data Access Objects only and don't put much additional logic on them. If you are however putting much more logic on the entities you should refrain from using the entity-generator and code your entities manually.

> Even if you specified Inheritance options in your XML or YAML Mapping files the generator cannot generate the base and child classes for you correctly, because it doesn't know which class is supposed to extend which. You have to adjust the entity code manually for inheritance to work!

Convert Mapping Information

Convert mapping information between supported formats.

This is an **execute one-time** command. It should not be necessary for you to call this method multiple times, especially when using the `--from-database` flag.

Converting an existing database schema into mapping files only solves about 70-80% of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

There is no need to convert YAML or XML mapping files to annotations every time you make changes. All mapping drivers are first class citizens in Doctrine 2 and can be used as runtime mapping for the ORM. See the docs on XML and YAML Mapping for an example how to register this metadata drivers as primary mapping source.

To convert some mapping information between the various supported formats you can use the `ClassMetadataExporter` to get exporter instances for the different formats:

```
<?php
$cme = new \Doctrine\ORM\Tools\Export\ClassMetadataExporter();
```

Once you have a instance you can use it to get an exporter. For example, the yml exporter:

```
<?php
$exporter = $cme->getExporter('yml', '/path/to/export/yml');
```

Now you can export some `ClassMetadata` instances:

```
<?php
$classes = array(
  $em->getClassMetadata('Entities\User'),
  $em->getClassMetadata('Entities\Profile')
);
$exporter->setMetadata($classes);
$exporter->export();
```

This functionality is also available from the command line to convert your loaded mapping information to another format. The `orm:convert-mapping` command accepts two arguments, the type to convert to and the path to generate it:

```
$ php doctrine orm:convert-mapping xml /path/to/mapping-path-converted-to-xml
```

Reverse Engineering

You can use the `DatabaseDriver` to reverse engineer a database to an array of `ClassMetadataInfo` instances and generate YAML, XML, etc. from them.

Reverse Engineering is a **one-time** process that can get you started with a project. Converting an existing database schema into mapping files only detects about 70-80% of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

First you need to retrieve the metadata instances with the `DatabaseDriver`:

```
<?php
$em->getConfiguration()->setMetadataDriverImpl(
    new \Doctrine\ORM\Mapping\Driver\DatabaseDriver(
        $em->getConnection()->getSchemaManager()
    )
);

$cmf = new \Doctrine\ORM\Tools\DisconnectedClassMetadataFactory();
```

```
$cmf->setEntityManager($em);
$metadata = $cmf->getAllMetadata();
```

Now you can get an exporter instance and export the loaded metadata to yml:

```
<?php
$cme = new \Doctrine\ORM\Tools\Export\ClassMetadataExporter();
$exporter = $cme->getExporter('yml', '/path/to/export/yml');
$exporter->setMetadata($metadata);
$exporter->export();
```

You can also reverse engineer a database using the `orm:convert-mapping` command:

```
$ php doctrine orm:convert-mapping --from-database yml /path/to/mapping-path-
converted-to-yml
```

Reverse Engineering is not always working perfectly depending on special cases. It will only detect Many-To-One relations (even if they are One-To-One) and will try to create entities from Many-To-Many tables. It also has problems with naming of foreign keys that have multiple column names. Any Reverse Engineered Database-Schema needs considerable manual work to become a useful domain model.

Runtime vs Development Mapping Validation

For performance reasons Doctrine ORM has to skip some of the necessary validation of metadata mappings. You have to execute this validation in your development workflow to verify the associations are correctly defined.

You can either use the Doctrine Command Line Tool:

```
doctrine orm:validate-schema
```

Or you can trigger the validation manually:

```
<?php
use Doctrine\ORM\Tools\SchemaValidator;

$validator = new SchemaValidator($entityManager);
$errors = $validator->validateMapping();

if (count($errors) > 0) {
    // Lots of errors!
    echo implode("\n\n", $errors);
}
```

If the mapping is invalid the errors array contains a positive number of elements with error messages.

One mapping option that is not validated is the use of the referenced column name. It has to point to the equivalent primary key otherwise Doctrine will not work.

One common error is to use a backlash in front of the fully-qualified class-name. Whenever a FQCN is represented inside a string (such as in your mapping definitions) you have to drop the prefix backslash. PHP does this with `get_class()` or Reflection methods for backwards compatibility reasons.

Adding own commands

You can also add your own commands on-top of the Doctrine supported tools if you are using a manually built console script.

To include a new command on Doctrine Console, you need to do modify the `doctrine.php` file a little:

```
<?php
// doctrine.php
use Symfony\Component\Console\Application;

// as before ...

// replace the ConsoleRunner::run() statement with:
$cli = new Application('Doctrine Command Line Interface', \Doctrine\ORM\
Version::VERSION);
$cli->setCatchExceptions(true);
$cli->setHelperSet($helperSet);

// Register All Doctrine Commands
ConsoleRunner::addCommands($cli);

// Register your own command
$cli->addCommand(new \MyProject\Tools\Console\Commands\MyCustomCommand);

// Runs console application
$cli->run();
```

Additionally, include multiple commands (and overriding previously defined ones) is possible through the command:

```
<?php

$cli->addCommands(array(
    new \MyProject\Tools\Console\Commands\MyCustomCommand(),
    new \MyProject\Tools\Console\Commands\SomethingCommand(),
    new \MyProject\Tools\Console\Commands\AnotherCommand(),
    new \MyProject\Tools\Console\Commands\OneMoreCommand(),
));
```

Re-use console application

You are also able to retrieve and re-use the default console application. Just call `ConsoleRunner::createApplication(...)` with an appropriate HelperSet, like it is described in the configuration section.

```
<?php

// Retrieve default console application
$cli = ConsoleRunner::createApplication($helperSet);

// Runs console application
$cli->run();
```

https://www.doctrine-project.org/projects/doctrine-orm/en/2.8/index.html

Annotations Reference

You've probably used docblock annotations in some form already, most likely to provide documentation metadata for a tool like `PHPDocumentor` (@author, @link, ...). Docblock annotations are a tool to embed metadata inside the documentation section which can then be processed by some tool. Doctrine ORM generalizes the concept of docblock annotations so that they can be used for any kind of metadata and so that it is easy to define new docblock annotations. In order to allow more involved annotation values and to reduce the chances of clashes with other docblock annotations, the Doctrine ORM docblock annotations feature an alternative syntax that is heavily inspired by the Annotation syntax introduced in Java 5.

The implementation of these enhanced docblock annotations is located in the `Doctrine\Common\Annotations` namespace and therefore part of the Common package. Doctrine ORM docblock annotations support namespaces and nested annotations among other things. The Doctrine ORM ORM defines its own set of docblock annotations for supplying object-relational mapping metadata.

> If you're not comfortable with the concept of docblock annotations, don't worry, as mentioned earlier Doctrine ORM provides XML and YAML alternatives and you could easily implement your own favourite mechanism for defining ORM metadata.

In this chapter a reference of every Doctrine ORM Annotation is given with short explanations on their context and usage.

Index

Reference

@Column

Marks an annotated instance variable as persistent. It has to be inside the instance variables PHP DocBlock comment. Any value hold inside this variable will be saved to and loaded from the database as part of the lifecycle of the instance variables entity-class.

Required attributes:

- **type**: Name of the Doctrine Type which is converted between PHP and Database representation.

Optional attributes:

- **name**: By default the property name is used for the database column name also, however the 'name' attribute allows you to determine the column name.
- **length**: Used by the "string" type to determine its maximum length in the database. Doctrine does not validate the length of a string value for you.
- **precision**: The precision for a decimal (exact numeric) column (applies only for decimal column), which is the maximum number of digits that are stored for the values.
- **scale**: The scale for a decimal (exact numeric) column (applies only for decimal column), which represents the number of digits to the right of the decimal point and must not be greater than *precision*.
- **unique**: Boolean value to determine if the value of the column should be unique across all rows of the underlying entities table.

- **nullable**: Determines if NULL values allowed for this column. If not specified, default value is false.
- **options**: Array of additional options:
    - `default`: The default value to set for the column if no value is supplied.
    - `unsigned`: Boolean value to determine if the column should be capable of representing only non-negative integers (applies only for integer column and might not be supported by all vendors).
    - `fixed`: Boolean value to determine if the specified length of a string column should be fixed or varying (applies only for string/binary column and might not be supported by all vendors).
    - `comment`: The comment of the column in the schema (might not be supported by all vendors).
    - `collation`: The collation of the column (only supported by Drizzle, Mysql, PostgreSQL>=9.1, Sqlite and SQLServer).
    - `check`: Adds a check constraint type to the column (might not be supported by all vendors).
- **columnDefinition**: DDL SQL snippet that starts after the column name and specifies the complete (non-portable!) column definition. This attribute allows to make use of advanced RMDBS features. However you should make careful use of this feature and the consequences. SchemaTool will not detect changes on the column correctly anymore if you use "columnDefinition". Additionally you should remember that the "type" attribute still handles the conversion between PHP and Database values. If you use this attribute on a column that is used for joins between tables you should also take a look at @JoinColumn.

For more detailed information on each attribute, please refer to the DBAL `Schema-Representation` documentation.

Examples:

<?php /** * @Column(type="string", length=32, unique=true, nullable=false) */ protected $username; /** * @Column(type="string", columnDefinition="CHAR(2) NOT NULL") */ protected $country; /** * @Column(type="decimal", precision=2, scale=1) */ protected $height; /** * @Column(type="string", length=2, options={"fixed":true, "comment":"Initial letters of first and last name"}) */ protected $initials; /** * @Column(type="integer", name="login_count", nullable=false, options={"unsigned":true, "default":0}) */ protected $loginCount;

@ColumnResult

References name of a column in the SELECT clause of a SQL query. Scalar result types can be included in the query result by specifying this annotation in the metadata.

Required attributes:

- **name**: The name of a column in the SELECT clause of a SQL query

@Cache

Add caching strategy to a root entity or a collection.

Optional attributes:

- **usage**: One of `READ_ONLY`, `READ_WRITE` or `NONSTRICT_READ_WRITE`, By default this is `READ_ONLY`.
- **region**: An specific region name

@ChangeTrackingPolicy

The Change Tracking Policy annotation allows to specify how the Doctrine ORM UnitOfWork should detect changes in properties of entities during flush. By default each entity is checked according to a deferred implicit strategy, which means upon flush UnitOfWork compares all the properties of an entity to a previously stored snapshot. This works out of the box, however you might want to tweak the flush performance where using another change tracking policy is an interesting option.

The details on all the available change tracking policies can be found in the configuration section.

Example:

```
<?php /** * @Entity * @ChangeTrackingPolicy("DEFERRED_IMPLICIT") *
@ChangeTrackingPolicy("DEFERRED_EXPLICIT") * @ChangeTrackingPolicy("NOTIFY") */
class User {}
```

@CustomIdGenerator

This annotations allows you to specify a user-provided class to generate identifiers. This annotation only works when both @Id and @GeneratedValue(strategy="CUSTOM") are specified.

Required attributes:

- **class**: name of the class which should extend Doctrine\ORM\Id\AbstractIdGenerator

Example:

```
<?php /** * @Id * @Column(type="integer") * @GeneratedValue(strategy="CUSTOM") *
@CustomIdGenerator(class="My\Namespace\MyIdGenerator") */ public $id;
```

@DiscriminatorColumn

This annotation is an optional annotation for the topmost/super class of an inheritance hierarchy. It specifies the details of the column which saves the name of the class, which the entity is actually instantiated as.

If this annotation is not specified, the discriminator column defaults to a string column of length 255 called `dtype`.

Required attributes:

- **name**: The column name of the discriminator. This name is also used during Array hydration as key to specify the class-name.

Optional attributes:

- **type**: By default this is string.
- **length**: By default this is 255.

@DiscriminatorMap

The discriminator map is a required annotation on the topmost/super class in an inheritance hierarchy. Its only argument is an array which defines which class should be saved under which name in the database. Keys are the database value and values are the classes, either as fully- or as unqualified class names depending on whether the classes are in the namespace or not.

```
<?php /** * @Entity * @InheritanceType("JOINED") * @DiscriminatorColumn(name="discr",
type="string") * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"}) */ class
Person { // ... }
```

@Embeddable

The embeddable annotation is required on a class, in order to make it embeddable inside an entity. It works together with the @Embedded annotation to establish the relationship between the two classes.

```
<?php /** * @Embeddable */ class Address { // ... class User { /** * @Embedded(class =
"Address") */ private $address;
```

@Embedded

The embedded annotation is required on an entity's member variable, in order to specify that it is an embedded class.

Required attributes:

- **class**: The embeddable class

```
<?php // ... class User { /** * @Embedded(class = "Address") */ private $address; /** *
@Embeddable */ class Address { // ...
```

@Entity

Required annotation to mark a PHP class as an entity. Doctrine manages the persistence of all classes marked as entities.

Optional attributes:

- **repositoryClass**: Specifies the FQCN of a subclass of the EntityRepository. Use of repositories for entities is encouraged to keep specialized DQL and SQL operations separated from the Model/Domain Layer.
- **readOnly**: Specifies that this entity is marked as read only and not considered for change-tracking. Entities of this type can be persisted and removed though.

Example:

```
<?php /** * @Entity(repositoryClass="MyProject\UserRepository") */ class User { //... }
```

@EntityResult

References an entity in the SELECT clause of a SQL query. If this annotation is used, the SQL statement should select all of the columns that are mapped to the entity object. This should include foreign key columns to related entities. The results obtained when insufficient data is available are undefined.

Required attributes:

- **entityClass**: The class of the result.

Optional attributes:

- **fields**: Array of @FieldResult, Maps the columns specified in the SELECT list of the query to the properties or fields of the entity class.
- **discriminatorColumn**: Specifies the column name of the column in the SELECT list that is used to determine the type of the entity instance.

@FieldResult

Is used to map the columns specified in the SELECT list of the query to the properties or fields of the entity class.

Required attributes:

- **name**: Name of the persistent field or property of the class.

Optional attributes:

- **column**: Name of the column in the SELECT clause.

@GeneratedValue

Specifies which strategy is used for identifier generation for an instance variable which is annotated by @Id. This annotation is optional and only has meaning when used in conjunction with @Id.

If this annotation is not specified with @Id the NONE strategy is used as default.

Optional attributes:

- **strategy**: Set the name of the identifier generation strategy. Valid values are AUTO, SEQUENCE, TABLE, IDENTITY, UUID, CUSTOM and NONE. If not specified, default value is AUTO.

Example:

<?php /** * @Id * @Column(type="integer") * @GeneratedValue(strategy="IDENTITY") */ protected $id = null;

@HasLifecycleCallbacks

Annotation which has to be set on the entity-class PHP DocBlock to notify Doctrine that this entity has entity lifecycle callback annotations set on at least one of its methods. Using @PostLoad, @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate or @PostUpdate without this marker annotation will make Doctrine ignore the callbacks.

Example:

<?php /** * @Entity * @HasLifecycleCallbacks */ class User { /** * @PostPersist */ public function sendOptinMail() {} }

@Index

Annotation is used inside the @Table annotation on the entity-class level. It provides a hint to the SchemaTool to generate a database index on the specified table columns. It only has meaning in the SchemaTool schema generation context.

Required attributes:

- **name**: Name of the Index
- **columns**: Array of columns.

Optional attributes:

- **options**: Array of platform specific options:
  - `where`: SQL WHERE condition to be used for partial indexes. It will only have effect on supported platforms.

Basic example:

<?php /** * @Entity * @Table(name="ecommerce_products",indexes={@Index(name="search_idx", columns={"name", "email"})}) */ class ECommerceProduct { }

Example with partial indexes:

<?php /** * @Entity * @Table(name="ecommerce_products",indexes={@Index(name="search_idx", columns={"name", "email"}, options={"where": "(((id IS NOT NULL) AND (name IS NULL)) AND (email IS NULL))"})}) */ class ECommerceProduct { }

@Id

The annotated instance variable will be marked as entity identifier, the primary key in the database. This annotation is a marker only and has no required or optional attributes. For entities that have multiple identifier columns each column has to be marked with @Id.

Example:

<?php /** * @Id * @Column(type="integer") */ protected $id = null;

@InheritanceType

In an inheritance hierarchy you have to use this annotation on the topmost/super class to define which strategy should be used for inheritance. Currently Single Table and Class Table Inheritance are supported.

This annotation has always been used in conjunction with the @DiscriminatorMap and @DiscriminatorColumn annotations.

Examples:

<?php /** * @Entity * @InheritanceType("SINGLE_TABLE") * @DiscriminatorColumn(name="discr", type="string") * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"}) */ class Person { // ... } /** * @Entity * @InheritanceType("JOINED") * @DiscriminatorColumn(name="discr", type="string") * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"}) */ class Person { // ... }

@JoinColumn

This annotation is used in the context of relations in @ManyToOne, @OneToOne fields and in the Context of @JoinTable nested inside a @ManyToMany. If this annotation or both *name* and *referencedColumnName* are missing they will be computed considering the field's name and the current naming strategy.

Optional attributes:

- **name**: Column name that holds the foreign key identifier for this relation. In the context of @JoinTable it specifies the column name in the join table.
- **referencedColumnName**: Name of the primary key identifier that is used for joining of this relation. Defaults to *id*.
- **unique**: Determines whether this relation is exclusive between the affected entities and should be enforced as such on the database constraint level. Defaults to false.
- **nullable**: Determine whether the related entity is required, or if null is an allowed state for the relation. Defaults to true.
- **onDelete**: Cascade Action (Database-level)
- **columnDefinition**: DDL SQL snippet that starts after the column name and specifies the complete (non-portable!) column definition. This attribute enables the use of advanced RMDBS features. Using this attribute on @JoinColumn is necessary if you need slightly different column definitions for joining columns, for example regarding NULL/NOT NULL defaults. However by default a "columnDefinition" attribute on @Column also sets the related @JoinColumn's columnDefinition. This is necessary to make foreign keys work.

Example:

```
<?php /** * @OneToOne(targetEntity="Customer") * @JoinColumn(name="customer_id",
referencedColumnName="id") */ private $customer;
```

@JoinColumns

An array of @JoinColumn annotations for a @ManyToOne or @OneToOne relation with an entity that has multiple identifiers.

@JoinTable

Using @OneToMany or @ManyToMany on the owning side of the relation requires to specify the @JoinTable annotation which describes the details of the database join table. If you do not specify @JoinTable on these relations reasonable mapping defaults apply using the affected table and the column names.

Optional attributes:

- **name**: Database name of the join-table
- **joinColumns**: An array of @JoinColumn annotations describing the join-relation between the owning entities table and the join table.
- **inverseJoinColumns**: An array of @JoinColumn annotations describing the join-relation between the inverse entities table and the join table.

Example:

```
<?php /** * @ManyToMany(targetEntity="Phonenumber") *
@JoinTable(name="users_phonenumbers", * joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")}, *
inverseJoinColumns={@JoinColumn(name="phonenumber_id", referencedColumnName="id",
unique=true)} * ) */ public $phonenumbers;
```

@ManyToOne

Defines that the annotated instance variable holds a reference that describes a many-to-one relationship between two entities.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **cascade**: Cascade Option
- **fetch**: One of LAZY or EAGER
- inversedBy - The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.

Example:

```
<?php /** * @ManyToOne(targetEntity="Cart", cascade={"all"}, fetch="EAGER") */ private
$cart;
```

@ManyToMany

Defines that the annotated instance variable holds a many-to-many relationship between two entities. @JoinTable is an additional, optional annotation that has reasonable default configuration values using the table and names of the two related entities.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **mappedBy**: This option specifies the property name on the targetEntity that is the owning side of this relation. It is a required attribute for the inverse side of a relationship.
- **inversedBy**: The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.
- **cascade**: Cascade Option
- **fetch**: One of LAZY, EXTRA_LAZY or EAGER
- **indexBy**: Index the collection by a field on the target entity.

For ManyToMany bidirectional relationships either side may be the owning side (the side that defines the @JoinTable and/or does not make use of the mappedBy attribute, thus using a default join table).

Example:

```
<?php /** * Owning Side * * @ManyToMany(targetEntity="Group", inversedBy="features") *
@JoinTable(name="user_groups", * joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")}, * inverseJoinColumns={@JoinColumn(name="group_id",
referencedColumnName="id")} * ) */ private $groups; /** * Inverse Side * *
@ManyToMany(targetEntity="User", mappedBy="groups") */ private $features;
```

@MappedSuperclass

A mapped superclass is an abstract or concrete class that provides persistent entity state and mapping information for its subclasses, but which is not itself an entity. This annotation is specified on the Class docblock and has no additional attributes.

The @MappedSuperclass annotation cannot be used in conjunction with @Entity. See the Inheritance Mapping section for more details on the restrictions of mapped superclasses.

Optional attributes:

- **repositoryClass**: Specifies the FQCN of a subclass of the EntityRepository. That will be inherited for all subclasses of that Mapped Superclass.

Example:

```
<?php /** * @MappedSuperclass */ class MappedSuperclassBase { // ... fields and methods } /** *
@Entity */ class EntitySubClassFoo extends MappedSuperclassBase { // ... fields and methods }
```

@NamedNativeQuery

Is used to specify a native SQL named query. The NamedNativeQuery annotation can be applied to an entity or mapped superclass.

Required attributes:

- **name**: The name used to refer to the query with the EntityManager methods that create query objects.
- **query**: The SQL query string.

Optional attributes:

- **resultClass**: The class of the result.
- **resultSetMapping**: The name of a SqlResultSetMapping, as defined in metadata.

Example:

```
<?php /** * @NamedNativeQueries({ * @NamedNativeQuery( * name = "fetchJoinedAddress", *
resultSetMapping= "mappingJoinedAddress", * query = "SELECT u.id, u.name, u.status, a.id AS
a_id, a.country, a.zip, a.city FROM cms_users u INNER JOIN cms_addresses a ON u.id =
a.user_id WHERE u.username = ?" * ), * }) * @SqlResultSetMappings({ *
@SqlResultSetMapping( * name = "mappingJoinedAddress", * entities= { * @EntityResult( *
entityClass = "__CLASS__", * fields = { * @FieldResult(name = "id"), * @FieldResult(name =
"name"), * @FieldResult(name = "status"), * @FieldResult(name = "address.zip"), *
@FieldResult(name = "address.city"), * @FieldResult(name = "address.country"), *
@FieldResult(name = "address.id", column = "a_id"), * } * ) * } * ) * }) */ class User { /** @Id
@Column(type="integer") @GeneratedValue */ public $id; /** @Column(type="string",
length=50, nullable=true) */ public $status; /** @Column(type="string", length=255, unique=true)
*/ public $username; /** @Column(type="string", length=255) */ public $name; /**
@OneToOne(targetEntity="Address") */ public $address; // .... }
```

@OneToOne

The @OneToOne annotation works almost exactly as the @ManyToOne with one additional option which can be specified. The configuration defaults for @JoinColumn using the target entity table and primary key column names apply here too.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **cascade**: Cascade Option
- **fetch**: One of LAZY or EAGER
- **orphanRemoval**: Boolean that specifies if orphans, inverse OneToOne entities that are not connected to any owning instance, should be removed by Doctrine. Defaults to false.
- **inversedBy**: The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.

Example:

```
<?php /** * @OneToOne(targetEntity="Customer") * @JoinColumn(name="customer_id",
referencedColumnName="id") */ private $customer;
```

@OneToMany

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **cascade**: Cascade Option
- **orphanRemoval**: Boolean that specifies if orphans, inverse OneToOne entities that are not connected to any owning instance, should be removed by Doctrine. Defaults to false.
- **mappedBy**: This option specifies the property name on the targetEntity that is the owning side of this relation. Its a required attribute for the inverse side of a relationship.
- **fetch**: One of LAZY, EXTRA_LAZY or EAGER.
- **indexBy**: Index the collection by a field on the target entity.

Example:

```
<?php /** * @OneToMany(targetEntity="Phonenumber", mappedBy="user", cascade={"persist",
"remove", "merge"}, orphanRemoval=true) */ public $phonenumbers;
```

@OrderBy

Optional annotation that can be specified with a @ManyToMany or @OneToMany annotation to specify by which criteria the collection should be retrieved from the database by using an ORDER BY clause.

This annotation requires a single non-attributed value with an DQL snippet:

Example:

```
<?php /** * @ManyToMany(targetEntity="Group") * @OrderBy({"name" = "ASC"}) */ private
$groups;
```

The DQL Snippet in OrderBy is only allowed to consist of unqualified, unquoted field names and of an optional ASC/DESC positional statement. Multiple Fields are separated by a comma (,). The referenced field names have to exist on the `targetEntity` class of the `@ManyToMany` or `@OneToMany` annotation.

@PostLoad

Marks a method on the entity to be called as a @PostLoad event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostPersist

Marks a method on the entity to be called as a @PostPersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostRemove

Marks a method on the entity to be called as a @PostRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostUpdate

Marks a method on the entity to be called as a @PostUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PrePersist

Marks a method on the entity to be called as a @PrePersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PreRemove

Marks a method on the entity to be called as a @PreRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PreUpdate

Marks a method on the entity to be called as a @PreUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@SequenceGenerator

For use with @GeneratedValue(strategy=SEQUENCE) this annotation allows to specify details about the sequence, such as the increment size and initial values of the sequence.

Required attributes:

- **sequenceName**: Name of the sequence

Optional attributes:

- **allocationSize**: Increment the sequence by the allocation size when its fetched. A value larger than 1 allows optimization for scenarios where you create more than one new entity per request. Defaults to 10
- **initialValue**: Where the sequence starts, defaults to 1.

Example:

```
<?php /** * @Id * @GeneratedValue(strategy="SEQUENCE") * @Column(type="integer") *
@SequenceGenerator(sequenceName="tablename_seq", initialValue=1, allocationSize=100) */
protected $id = null;
```

@SqlResultSetMapping

The SqlResultSetMapping annotation is used to specify the mapping of the result of a native SQL query. The SqlResultSetMapping annotation can be applied to an entity or mapped superclass.

Required attributes:

- **name**: The name given to the result set mapping, and used to refer to it in the methods of the Query API.

Optional attributes:

- **entities**: Array of @EntityResult, Specifies the result set mapping to entities.
- **columns**: Array of @ColumnResult, Specifies the result set mapping to scalar values.

Example:

```
<?php /** * @NamedNativeQueries({ * @NamedNativeQuery( * name =
"fetchUserPhonenumberCount", * resultSetMapping= "mappingUserPhonenumberCount", * query
= "SELECT id, name, status, COUNT(phonenumber) AS numphones FROM cms_users INNER
JOIN cms_phonenumbers ON id = user_id WHERE username IN (?) GROUP BY id, name, status,
username ORDER BY username" * ), * @NamedNativeQuery( * name =
"fetchMultipleJoinsEntityResults", * resultSetMapping= "mappingMultipleJoinsEntityResults", *
query = "SELECT u.id AS u_id, u.name AS u_name, u.status AS u_status, a.id AS a_id, a.zip AS
a_zip, a.country AS a_country, COUNT(p.phonenumber) AS numphones FROM cms_users u
INNER JOIN cms_addresses a ON u.id = a.user_id INNER JOIN cms_phonenumbers p ON u.id =
p.user_id GROUP BY u.id, u.name, u.status, u.username, a.id, a.zip, a.country ORDER BY
u.username" * ), * }) * @SqlResultSetMappings({ * @SqlResultSetMapping( * name =
"mappingUserPhonenumberCount", * entities= { * @EntityResult( * entityClass = "User", * fields
= { * @FieldResult(name = "id"), * @FieldResult(name = "name"), * @FieldResult(name =
"status"), * } * ) * }, * columns = { * @ColumnResult("numphones") * } * ), *
@SqlResultSetMapping( * name = "mappingMultipleJoinsEntityResults", * entities= { *
@EntityResult( * entityClass = "__CLASS__", * fields = { * @FieldResult(name = "id",
column="u_id"), * @FieldResult(name = "name", column="u_name"), * @FieldResult(name =
"status", column="u_status"), * } * ), * @EntityResult( * entityClass = "Address", * fields = { *
@FieldResult(name = "id", column="a_id"), * @FieldResult(name = "zip", column="a_zip"), *
@FieldResult(name = "country", column="a_country"), * } * ) * }, * columns = { *
@ColumnResult("numphones") * } * ) *}) */ class User { /** @Id @Column(type="integer")
@GeneratedValue */ public $id; /** @Column(type="string", length=50, nullable=true) */ public
$status; /** @Column(type="string", length=255, unique=true) */ public $username; /**
@Column(type="string", length=255) */ public $name; /**
@OneToMany(targetEntity="Phonenumber") */ public $phonenumbers; /**
@OneToOne(targetEntity="Address") */ public $address; // .... }
```

@Table

Annotation describes the table an entity is persisted in. It is placed on the entity-class PHP DocBlock and is optional. If it is not specified the table name will default to the entity's unqualified classname.

Required attributes:

- **name**: Name of the table

Optional attributes:

- **indexes**: Array of @Index annotations
- **uniqueConstraints**: Array of @UniqueConstraint annotations.
- **schema**: Name of the schema the table lies in.

Example:

<?php /** * @Entity * @Table(name="user", * uniqueConstraints={@UniqueConstraint(name="user_unique",columns={"username"})}, * indexes={@Index(name="user_idx", columns={"email"})} * schema="schema_name" *) */ class User { }

@UniqueConstraint

Annotation is used inside the @Table annotation on the entity-class level. It allows to hint the SchemaTool to generate a database unique constraint on the specified table columns. It only has meaning in the SchemaTool schema generation context.

Required attributes:

- **name**: Name of the Index
- **columns**: Array of columns.

Optional attributes:

- **options**: Array of platform specific options:
  - `where`: SQL WHERE condition to be used for partial indexes. It will only have effect on supported platforms.

Basic example:

<?php /** * @Entity * @Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx", columns={"name", "email"})}) */ class ECommerceProduct { }
Example with partial indexes:

<?php /** * @Entity * @Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx", columns={"name", "email"}, options={"where": "(((id IS NOT NULL) AND (name IS NULL)) AND (email IS NULL))"})}) */ class ECommerceProduct { }

@Version

Marker annotation that defines a specified column as version attribute used in an optimistic locking scenario. It only works on @Column annotations that have the type `integer` or `datetime`. Combining `@Version` with @Id is not supported.

Example:

<?php /** * @Column(type="integer") * @Version */ protected $version;
https://www.doctrine-project.org/projects/doctrine-orm/en/2.8/reference/annotations-reference.html