Doctrine ORM for PHP

Guide to Doctrine for PHP

Doctrine 2.0

Version: manual-2.0-en-2010-02-10

Table of Contents

Chapter 1

Introduction

Welcome

Doctrine 2 is an object-relational mapper (ORM) for PHP 5.3.0+ that provides transparent persistence for PHP objects. It sits on top of a powerful database abstraction layer (DBAL). One of its key features is the option to write database queries in a proprietary object oriented SQL dialect called Doctrine Query Language (DQL), inspired by Hibernates HQL. This provides developers with a powerful alternative to SQL that maintains flexibility without requiring unnecessary code duplication.

Disclaimer

This is the Doctrine 2 reference documentation. Introductory guides and tutorials that you can follow along from start to finish, like the "Guide to Doctrine" book known from the Doctrine 1.x series, will be available at a later date.

Requirements

Doctrine 2 requires a minimum of PHP 5.3.0. For greatly improved performance it is also recommended that you use APC with PHP.

Doctrine 2 Packages

Doctrine 2 is divided into three main packages.

- Common
- DBAL (includes Common)
- ORM (includes DBAL+Common)

This manual mainly covers the ORM package, sometimes touching parts of the underlying DBAL and Common packages. The Doctrine code base is split in to these packages for a few reasons and they are to...

- ...make things more maintainable and decoupled
- ...allow you to use the code in Doctrine Common without the ORM or DBAL
- ...allow you to use the DBAL without the ORM

The Common Package

The Common package contains highly reusable components that have no dependencies beyond the package itself (and PHP, of course). The root namespace of the Common package is `Doctrine\Common`.

The DBAL Package

The DBAL package contains an enhanced database abstraction layer on top of PDO but is not strongly bound to PDO. The purpose of this layer is to provide a single API that bridges most of the differences between the different RDBMS vendors. The root namespace of the DBAL package is `Doctrine\DBAL`.

The ORM Package

The ORM package contains the object-relational mapping toolkit that provides transparent relational persistence for plain PHP objects. The root namespace of the ORM package is `Doctrine\ORM`.

Installing

Doctrine can be installed many different ways. We will describe all the different ways and you can choose which one suits you best.

PEAR

You can easily install any of the three Doctrine packages from the PEAR command line installation utility.

To install just the `Common` package you can run the following command:

```
$ sudo pear install pear.phpdoctrine.org/DoctrineCommon-2.0.0
```

Listing 1-1

If you want to use the Doctrine Database Abstraction Layer you can install it with the following command.

```
$ sudo pear install pear.phpdoctrine.org/DoctrineDBAL-2.0.0
```

Listing 1-2

Or, if you want to get the works and go for the ORM you can install it with the following command.

```
$ sudo pear install pear.phpdoctrine.org/DoctrineORM-2.0.0
```

Listing 1-3

When you have a package installed via PEAR you can required and load the `ClassLoader` with the following code.

```
<?php

require 'Doctrine/Common/ClassLoader.php';
$classLoader = new \Doctrine\Common\ClassLoader();
```

The packages are installed in to your shared PEAR PHP code folder in a folder named `Doctrine`. You also get a nice command line utility installed and made available on your system. Now when you run the `doctrine` command you will see what you can do with it.

Listing 1-4
```
$ doctrine
Doctrine Command Line Interface
Available Tasks:
core:help
dbal:run-sql (--file=<path> | --sql=<SQL>) --depth=<DEPTH>
orm:clear-cache (--query | --metadata | --result [--id=<ID>]
[--regex=<REGEX>] [--prefix=<PREFIX>] [--suffix=<SUFFIX>])
orm:convert-mapping (--from=<SOURCE> | --from-database) --to=<TYPE>
--dest=<PATH>
orm:ensure-production-settings
orm:generate-proxies --class-dir=<PATH> [--to-dir=<PATH>]
orm:run-dql --dql=<DQL> --depth=<DEPTH>
orm:schema-tool (--create | --drop | --update | --complete-update |
--re-create) [--dump-sql] [--class-dir=<PATH>]
orm:version
```

Package Download

You can also use Doctrine 2 by downloading the latest release package from the download page[1].

Subversion

Alternatively you can check out the latest version of Doctrine 2 via SVN.

Listing 1-5
```
$ svn co http://SVN.doctrine-project.org/trunk doctrine
```

Sandbox Quickstart

The sandbox is only available via SVN or soon as a separate download on the downloads page.

The sandbox is a pre-configured environment for evaluating and playing with Doctrine 2.

Overview

After navigating to the sandbox directory, you should see the following structure:

Listing 1-6
```
sandbox/
    Entities/
        Address.php
        User.php
    xml/
        Entities.Address.dcm.xml
        Entities.User.dcm.xml
    yaml/
        Entities.Address.dcm.yml
        Entities.User.dcm.yml
    cli-config.php
    doctrine
    doctrine.php
    index.php
```

1. http://www.doctrine-project.org/download

Here is a short overview of the purpose of these folders and files:

- The `Entities` folder is where any model classes are created. Two example entities are already there.
- The `xml` folder is where any XML mapping files are created (if you want to use XML mapping). Two example mapping documents for the 2 example entities are already there.
- The `yaml` folder is where any YAML mapping files are created (if you want to use YAML mapping). Two example mapping documents for the 2 example entities are already there.
- The `cli-config.php` contains bootstrap code for a configuration that is used by the CLI tool `doctrine` whenever you execute a task.
- `doctrine/doctrine.php` is a command-line tool.
- `index.php` is a basic classical bootstrap file of a php application that uses Doctrine 2.

Mini-tutorial

1) From within the tools/sandbox folder, run the following command and you should see the same output.

Listing 1-7

```
$ php doctrine orm:schema-tool --create
Creating database schema...
Database schema created successfully.
```

2) Take another look into the tools/sandbox folder. A SQLite database should have been created with the name `database.sqlite`.

3) Open `index.php` and edit it so that it looks as follows:

```
<?php

//... bootstrap stuff

## PUT YOUR TEST CODE BELOW

$user = new \Entities\User;
$user->setName('Garfield');
$em->persist($user);
$em->flush();

echo "User saved!";
```

Open index.php in your browser or execute it on the command line. You should see the output "User saved!".

5) Inspect the SQLite database. Again from within the tools/sandbox folder, execute the following command:

Listing 1-8

```
$ php doctrine dbal:run-sql --sql="select * from users"
```

You should get the following output:

Listing 1-9

```
array(1) {
  [0]=>
  array(2) {
    ["id"]=>
    string(1) "1"
    ["name"]=>
    string(8) "Garfield"
  }
}
```

You just saved your first entity with a generated ID in an SQLite database.

6) Replace the contents of index.php with the following:

```
<?php

//... bootstrap stuff

## PUT YOUR TEST CODE BELOW

$q = $em->createQuery('select u from Entities\User u where u.name = ?1');
$q->setParameter(1, 'Garfield');
$garfield = $q->getSingleResult();

echo "Hello " . $garfield->getName() . "!";
```

You just created your first DQL query to retrieve the user with the name 'Garfield' from an SQLite database (Yes, there is an easier way to do it, but we wanted to introduce you to DQL at this point. Can you **find** the easier way?).

When you create new model classes or alter existing ones you can recreate the database schema with the command `doctrine orm:schema-tool --drop` followed by `doctrine orm:schema-tool --create`.

---

7) Explore Doctrine 2!

Chapter 2

Architecture

This chapter gives an overview of the overall architecture and terminology of Doctrine 2.

Entities

An entity is a lightweight persistent domain object. An entity can be any regular php class that obeys to the following restrictions:

- An entity class must have a default no-arg constructor. That means if you want to make use of a parameterized entity constructor, all parameters must be optional.
- An entity class must not be final or contain final methods.
- Any two entity classes in a class hierarchy that inherit directly or indirectly from one another must not have a mapped property with the same name.

Entities support inheritance, polymorphic associations, and polymorphic queries. Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.

Entity states

An entity instance can be characterized as being NEW, MANAGED, DETACHED or REMOVED.

- A NEW entity instance has no persistent identity, and is not yet associated with an EntityManager and a UnitOfWork (i.e. those just created with the "new" operator).
- A MANAGED entity instance is an instance with a persistent identity that is associated with an EntityManager and whose persistence is thus managed.
- A DETACHED entity instance is an instance with a persistent identity that is not (or no longer) associated with an EntityManager and a UnitOfWork.
- A REMOVED entity instance is an instance with a persistent identity, associated with an EntityManager, that will be removed from the database upon transaction commit.

Persistent fields

The persistent state of an entity is represented by instance variables. An instance variable must be directly accessed only from within the methods of the entity by the entity instance itself. Instance variables must not be accessed by clients of the entity. The state of the entity is available to clients only through the entity’s methods, i.e. accessor methods (getter/setter methods) or other business methods.

Collection-valued persistent fields and properties must be defined in terms of the `Doctrine\Common\Collections\Collection` interface. The collection implementation type may be used by the application to initialize fields or properties before the entity is made persistent. Once the entity becomes managed (or detached), subsequent access must be through the interface type.

The EntityManager

The `EntityManager` class is a central access point to the ORM functionality provided by Doctrine 2. The `EntityManager` API is used to manage the persistence of your objects and to query for persistent objects.

Transactional write-behind

An `EntityManager` and the underlying `UnitOfWork` employ a strategy called "transactional write-behind" that delays the execution of SQL statements in order to execute them in the most efficient way and to execute them at the end of a transaction so that all write locks are quickly released. You should see Doctrine as a tool to synchronize your in-memory objects with the database in well defined units of work. Work with your objects and modify them as usual and when you're done call `EntityManager#flush()` to make your changes persistent.

The Unit of Work

Internally an `EntityManager` uses a `UnitOfWork`, which is a typical implementation of the [http://martinfowler.com/eaaCatalog/unitOfWork.html Unit of Work Pattern], to keep track of all the things that need to be done the next time `flush` is invoked. You usually do not directly interact with a `UnitOfWork` but with the `EntityManager` instead.

Chapter 3
Configuration

Bootstrapping

Bootstrapping Doctrine is a relatively simple procedure that roughly exists of just 2 steps:

- Making sure Doctrine class files can be loaded on demand.
- Obtaining an EntityManager instance.

Class loading

Lets start with the class loading setup. We need to set up some class loader (often called "autoloader") so that Doctrine class files are loaded on demand. The Doctrine\Common namespace contains two very fast and minimalistic class loaders that can be used for Doctrine and any other libraries where the coding standards ensure that a class's location in the directory tree is reflected by its name and namespace and where there is a common root namespace.

These two class loaders are `Doctrine\Common\GlobalClassLoader` and `Doctrine\Common\IsolatedClassLoader`. The former is meant to be used as the one and only class loader for all classes in an application, thus it must be the only autoloader on the spl autoload stack. The latter, `IsolatedClassLoader`, is a class loader that only loads classes of a single (root) namespace and can be put on the spl autoloader stack together with others.

You are not forced to use one of the two mentioned class loaders to load Doctrine classes. Doctrine does not care how the classes are loaded, if you want to use a different class loader or your own to load Doctrine classes, just do that. Along the same lines, the 2 class loaders in the Doctrine\Common namespace are not meant to be only used for Doctrine classes, too. They are generic class loaders that can be used for any classes that follow some basic naming standards as described above.

The following example shows the setup of a `GlobalClassLoader`

This assumes you've created some kind of script to test the following code in. Something like a `test.php` file.

```
<?php

// test.php
```

```
require '/path/to/lib/Doctrine/Common/GlobalClassLoader.php';
$classLoader = new \Doctrine\Common\GlobalClassLoader();
$classLoader->registerNamespace('Doctrine', '/path/to/Doctrine2/lib/');
//... register more namespaces
$classLoader->register(); // register on SPL autoload stack
```

Alternatively, the following is an example that shows the usage of an `IsolatedClassLoader`.

```
<?php

// test.php

require '/path/to/lib/Doctrine/Common/IsolatedClassLoader.php';
$classLoader = new \Doctrine\Common\IsolatedClassLoader('Doctrine');
$classLoader->setBasePath('/path/to/Doctrine2/lib/');
$classLoader->register(); // register on SPL autoload stack
// ... create more isolated class loaders for different namespaces if you
want to
```

For any class libraries you installed through PEAR, including Doctrine, you do not even need to call `GlocalClassLoader#registerNamespace()` or `IsolatedClassLoader#setBasePath()`, respectively. Since the path to the PEAR library is in the include_path and all PEAR libraries follow the standard that is compatible with the class loaders, it works right away.

For best class loading performance it is recommended that you keep your include_path short, ideally it should only contain the path to the PEAR libraries, and any other class libraries should be registered with their full base path either on a `GlobalClassLoader` or an `IsolatedClassLoader`.

Obtaining an EntityManager

Once you have prepared the class loading, you acquire an EntityManager instance with the following minimalist configuration:

```
<?php

use Doctrine\ORM\EntityManager,
    Doctrine\ORM\Configuration;

// ...

$config = new Configuration;
$cache = new \Doctrine\Common\Cache\ApcCache;
$config->setMetadataCacheImpl($cache);
$config->setQueryCacheImpl($cache);
$config->setProxyDir('/path/to/myproject/lib/MyProject/Proxies');
$config->setProxyNamespace('MyProject\Proxies');
$connectionOptions = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite'
);
```

```
$em = EntityManager::create($connectionOptions, $config);
```

Do not use Doctrine without a metadata and query cache! Doctrine is highly optimized for working with caches. The main parts in Doctrine that are optimized for caching are the metadata mapping information with the metadata cache and the DQL to SQL conversions with the query cache. These 2 caches require only an absolute minimum of memory yet they heavily improve the runtime performance of Doctrine. The recommended cache driver to use with Doctrine is APC[2]. APC provides you with an opcode-cache (which is highly recommended anyway) and a very fast in-memory cache storage that you can use for the metadata and query caches as seen in the previous code snippet.

An EntityManager is your central access point to ORM functionality provided by Doctrine.

Configuration Options

The following sections describe all the configuration options available on a `Doctrine\ORM\Configuration` instance.

Proxy Directory (***REQUIRED***)

```
<?php

$config->setProxyDir($dir);
$config->getProxyDir();
```

Gets or sets the directory where Doctrine generates any proxy classes. For a detailed explanation on proxy classes and how they are used in Doctrine, refer to the "Proxy Objects" section further down.

Proxy Namespace (***REQUIRED***)

```
<?php

$config->setProxyNamespace($namespace);
$config->getProxyNamespace();
```

Gets or sets the namespace to use for generated proxy classes. For a detailed explanation on proxy classes and how they are used in Doctrine, refer to the "Proxy Objects" section further down.

Metadata Driver (***REQUIRED***)

```
<?php

$config->setMetadataDriverImpl($driver);
$config->getMetadataDriverImpl();
```

2. `http://www.php.net/apc`

Gets or sets the metadata driver implementation that is used by Doctrine to acquire the object-relational metadata for your classes.

There are currently 3 available implementations:

- `Doctrine\ORM\Mapping\Driver\AnnotationDriver`
- `Doctrine\ORM\Mapping\Driver\XmlDriver`
- `Doctrine\ORM\Mapping\Driver\YamlDriver`

Throughout the most part of this manual the AnnotationDriver is used in the examples. For information on the usage of the XmlDriver or YamlDriver please refer to the dedicated chapters `XML Mapping` and `YAML Mapping`.

Metadata Cache (***RECOMMENDED***)

```php
<?php

$config->setMetadataCacheImpl($cache);
$config->getMetadataCacheImpl();
```

Gets or sets the cache implementation to use for caching metadata information, that is, all the information you supply via annotations, xml or yaml, so that they do not need to be parsed and loaded from scratch on every single request which is a waste of resources. The cache implementation must implement the `Doctrine\Common\Cache\Cache` interface.

Usage of a metadata cache is highly recommended.

The recommended implementations are:

- `Doctrine\Common\Cache\ApcCache`
- `Doctrine\Common\Cache\MemcacheCache`
- `Doctrine\Common\Cache\XcacheCache`

Query Cache (***RECOMMENDED***)

```php
<?php

$config->setQueryCacheImpl($cache);
$config->getQueryCacheImpl();
```

Gets or sets the cache implementation to use for caching DQL queries, that is, the result of a DQL parsing process that includes the final SQL as well as meta information about how to process the SQL result set of a query. Note that the query cache does not affect query results. You do not get stale data. This is a pure optimization cache without any negative side-effects (except some minimal memory usage in your cache).

Usage of a query cache is highly recommended.

The recommended implementations are:

- `Doctrine\Common\Cache\ApcCache`
- `Doctrine\Common\Cache\MemcacheCache`
- `Doctrine\Common\Cache\XcacheCache`

SQL Logger (***Optional***)

Listing 3-1

```
[php]
$config->setSqlLogger($logger);
$config->getSqlLogger();
```

Gets or sets the logger to use for logging all SQL statements executed by Doctrine. The logger class must implement the `Doctrine\DBAL\Logging\SqlLogger` interface. A simple default implementation that logs to the standard output using `echo` and `var_dump` can be found at `Doctrine\DBAL\Logging\EchoSqlLogger`.

Auto-generating Proxy Classes (***OPTIONAL***)

```
<?php

$config->setAutoGenerateProxyClasses($bool);
$config->getAutoGenerateProxyClasses();
```

Gets or sets whether proxy classes should be generated automatically at runtime by Doctrine. If set to `FALSE`, proxy classes must be generated manually through the doctrine command line task `generate-proxies`. The strongly recommended value for a production environment is `FALSE`.

Change Tracking Policies

Change tracking is the process of determining what has changed in managed entities since the last time they were synchronized with the database.

Doctrine provides 3 different change tracking policies, each having its particular advantages and disadvantages. The change tracking policy can be defined on a per-class basis (or more precisely, per-hierarchy).

Deferred Implicit

The deferred implicit policy is the default change tracking policy and the most convenient one. With this policy, Doctrine detects the changes by a property-by-property comparison at commit time and also detects changes to entities or new entities that are referenced by other managed entities ("persistence by reachability"). Although the most convenient policy, it can have negative effects on performance if you are dealing with large units of work (see "Understanding the Unit of Work"). Since Doctrine can't know what has changed, it needs to check all managed entities for changes every time you invoke EntityManager#flush(), making this operation rather costly.

Deferred Explicit

The deferred explicit policy is similar to the deferred implicit policy in that it detects changes through a property-by-property comparison at commit time. The difference is that only entities are considered that have been explicitly marked for change detection through a call to EntityManager#persist(entity) or through a save cascade. All other entities are skipped. This policy therefore gives improved performance for larger units of work while sacrificing the behavior of "automatic dirty checking".

Therefore, flush() operations are potentially cheaper with this policy. The negative aspect this has is that if you have a rather large application and you pass your objects through several layers for processing purposes and business tasks you may need to track yourself which entities have changed on the way so you can pass them to EntityManager#persist().

This policy can be configured as follows:

Listing 3-2

```
/**
 * @Entity
 * @ChangeTrackingPolicy("DEFERRED_EXPLICIT")
 */
class User
{
    // ...
}
```

Notify

This policy is based on the assumption that the entities notify interested listeners of changes to their properties. For that purpose, a class that wants to use this policy needs to implement the NotifyPropertyChanged? interface from the Doctrine\Common namespace. As a guideline, such an implementation should look as follows:

```
<?php

use Doctrine\Common\NotifyPropertyChanged,
    Doctrine\Common\PropertyChangedListener;

/**
 * @Entity
 */
class MyEntity implements NotifyPropertyChanged
{
    // ...

    private $_listeners = array();

    public function addPropertyChangedListener(PropertyChangedListener
$listener)
    {
        $this->_listeners[] = $listener;
    }

    protected function _onPropertyChanged($propName, $oldValue, $newValue)
    {
        if ($this->_listeners) {
            foreach ($this->_listeners as $listener) {
                $listener->propertyChanged($this, $propName, $oldValue,
$newValue);
            }
        }
    }
}
```

Then, in each property setter of this class or derived classes, you need to invoke `_onPropertyChanged` as follows to notify listeners:

```
<?php

<?php
// ...

class MyEntity implements NotifyPropertyChanged
{
    // ...

    public function setData($data)
    {
        if ($data != $this->data) {
            $this->_onPropertyChanged('data', $this->data, $data);
            $this->data = $data;
        }
    }
}
```

The check whether the new value is different from the old one is not mandatory but recommended. That way you also have full control over when you consider a property changed.

The negative point of this policy is obvious: You need implement an interface and write some plumbing code. But also note that we tried hard to keep this notification functionality abstract. Strictly speaking, it has nothing to do with the persistence layer and the Doctrine ORM or DBAL. You may find that property notification events come in handy in many other scenarios as well. As mentioned earlier, the `Doctrine\Common` namespace is not that evil and consists solely of very small classes and interfaces that have almost no external dependencies (none to the DBAL and none to the ORM) and that you can easily take with you should you want to swap out the persistence layer. This change tracking policy does not introduce a dependency on the Doctrine DBAL/ORM or the persistence layer.

The positive point and main advantage of this policy is its effectiveness. It has the best performance characteristics of the 3 policies with larger units of work and a flush() operation is very cheap when nothing has changed.

Partial Objects

A partial object is an object whose state is not fully initialized after being reconstituted from the database and that is disconnected from the rest of its data. The following section will describe why partial objects are problematic and what the approach of Doctrine2 to this problem is.

The partial object problem in general does not apply to methods or queries where you do not retrieve the query result as objects. Examples are: `Query#getArrayResult()`, `Query#getScalarResult()`, `Query#getSingleScalarResult()`, etc.

What is the problem?

In short, partial objects are problematic because they are usually objects with broken invariants. As such, code that uses these partial objects tends to be very fragile and either needs to "know" which fields or methods can be safely accessed or add checks around every field access or method invocation. The same holds true for the internals, i.e. the method implementations, of such objects. You usually simply assume the state you need in the method is available, after all you properly constructed this object before you pushed it into the database, right? These blind assumptions can quickly lead to null reference errors when working with such partial objects.

It gets worse with the scenario of an optional association (0..1 to 1). When the associated field is NULL, you dont know whether this object does not have an associated object or whether it was simply not loaded when the owning object was loaded from the database.

These are reasons why many ORMs do not allow partial objects at all and instead you always have to load an object with all its fields (associations being proxied). One secure way to allow partial objects is if the programming language/platform allows the ORM tool to hook deeply into the object and instrument it in such a way that individual fields (not only associations) can be loaded lazily on first access. This is possible in Java, for example, through bytecode instrumentation. In PHP though this is not possible, so there is no way to have "secure" partial objects in an ORM with transparent persistence.

Doctrine, by default, does not allow partial objects. That means, any query that only selects partial object data and wants to retrieve the result as objects (i.e. `Query#getResult()`) will raise an exception telling you that partial objects are dangerous. If you want to force a query to return you partial objects, possibly as a performance tweak, you can use the `Query#HINT_FORCE_PARTIAL_LOAD` query hint as follows:

```
<?php

$q = $em->createQuery("select u.id, u.name from MyApp\Domain\User u");
$q->setHint(Query::HINT_FORCE_PARTIAL_LOAD, true);
```

When should I force partial objects?

Mainly for optimization purposes, especially since the stateless nature of PHP applications means that any fields or objects that are loaded unnecessarily in a request are useless (though often minimal) overhead. Be careful of premature optimization. Only force partial objects if it proves to provide an improvement to a performance problem.

Proxy Objects

A proxy object is an object that is put in place or used instead of the "real" object. A proxy object can add behavior to the object being proxied without that object being aware of it. In Doctrine 2, proxy objects are used to realize several features but mainly for transparent lazy-loading.

Proxy objects with their lazy-loading facilities help to keep the subset of objects that are already in memory connected to the rest of the objects. This is an essential property as without it there would always be fragile partial objects at the outer edges of your object graph.

Doctrine 2 implements a variant of the proxy pattern where it generates classes that extend your entity classes and adds lazy-loading capabilities to them. Doctrine can then give you an instance of such a proxy class whenever you request an object of the class being proxied. This happens in two situations:

Reference Proxies

The method `EntityManager#getReference($entityName, $identifier)` lets you obtain a reference to an entity for which the identifier is known, without loading that entity from the database. This is useful, for example, as a performance enhancement, when you want to establish an association to an entity for which you have the identifier. You could simply do this:

```
<?php

// $em instanceof EntityManager, $cart instanceof MyProject\Model\Cart
// $itemId comes from somewhere, probably a request parameter
$item = $em->getReference('MyProject\Model\Item', $itemId);
$cart->addItem($item);
```

Here, we added an Item to a Cart without loading the Item from the database. If you invoke any method on the Item instance, it would fully initialize its state transparently from the database. Here $item is actually an instance of the proxy class that was generated for the Item class but your code does not need to care. In fact it **should not care**. Proxy objects should be transparent to your code.

Association proxies

The second most important situation where Doctrine uses proxy objects is when querying for objects. Whenever you query for an object that has a single-valued association to another object that is configured LAZY, without joining that association in the same query, Doctrine puts proxy objects in place where normally the associated object would be. Just like other proxies it will transparently initialize itself on first access.

Joining an association in a DQL or native query essentially means eager loading of that association in that query. This will override the 'fetch' option specified in the mapping for that association, but only for that query.

Generating Proxy classes

Proxy classes can either be generated manually through the Doctrine CLI or automatically by Doctrine. The configuration option that controls this behavior is:

```
<?php

$config->setAutoGenerateProxyClasses($bool);
$config->getAutoGenerateProxyClasses();
```

The default value is `TRUE` for convenient development. However, this setting is not optimal for performance and therefore not recommended for a production environment. To eliminate the overhead of proxy class generation during runtime, set this configuration option to `FALSE`. When you do this in a development environment, note that you may get class/file not found errors if certain proxy classes are not available or failing lazy-loads if new methods were added to the entity class that are not yet in the proxy class. In such a case, simply use the Doctrine CLI to (re)generate the proxy classes like so:

```
doctrine generate-proxies
```

Listing 3-3

Chapter 4

Basic Mapping

This chapter explains the basic mapping of objects and properties. Mapping of associations will be covered in the next chapter "Association Mapping".

Mapping Drivers

Doctrine provides several different ways for specifying object-relational mapping metadata:

- Docblock Annotations
- XML
- YAML

This manual usually uses docblock annotations in all the examples that are spread throughout all chapters. There are dedicated chapters for XML and YAML mapping, respectively.

If you're wondering which mapping driver gives the best performance, the answer is: None. 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. Therefore at the end of the day all drivers perform equally well. If you're not using a metadata cache (not recommended!) then the XML driver might have a slight edge in performance due to the powerful native XML support in PHP.

Introduction to Docblock Annotations

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 2 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 2 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 2 docblock annotations support namespaces and nested annotations among other things. The Doctrine 2 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 2 provides XML and YAML alternatives and you could easily implement your own favourite mechanism for defining ORM metadata.

Persistent classes

In order to mark a class for object-relational persistence it needs to be designated as an entity. This can be done through the `@Entity` marker annotation.

```
<?php

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

By default, the entity will be persisted to a table with the same name as the class name. In order to change that, you can use the `@Table` annotation as follows:

```
<?php

/**
 * @Entity
 * @Table(name="my_persistent_class")
 */
class MyPersistentClass
{
    //...
}
```

Now instances of MyPersistentClass will be persisted into a table named `my_persistent_class`.

Doctrine Mapping Types

A Doctrine Mapping Type defines the mapping between a PHP type and an SQL type. All Doctrine Mapping Types that ship with Doctrine are fully portable between different RDBMS. You can even write your own custom mapping types that might or might not be portable, which is explained later in this chapter.

For example, the Doctrine Mapping Type `string` defines the mapping from a PHP string to an 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 an SQL VARCHAR to a PHP string.
- `integer`: Type that maps an 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 an SQL boolean to a PHP boolean.
- `decimal`: Type that maps an SQL DECIMAL to a PHP double.

- `date`: Type that maps an SQL DATETIME to a PHP DateTime object.
- `time`: Type that maps an SQL TIME to a PHP DateTime object.
- `datetime`: Type that maps an SQL DATETIME/TIMESTAMP to a PHP DateTime object.
- `text`: Type that maps an SQL CLOB to a PHP string.

Doctrine Mapping Types are NOT SQL types and NOT PHP types! They are mapping types between 2 types.

Property Mapping

After a class has been marked as an entity it can specify mappings for its instance fields. Here we will only look at simple fields that hold scalar values like strings, numbers, etc. Associations to other objects are covered in the chapter "Association Mapping".

To mark a property for relational persistence the `@Column` docblock annotation is used. This annotation requires at least 1 attribute to be set, the `type`. The `type` attribute specifies the Doctrine Mapping Type to use for the field.

Example:

```
<?php

/** @Entity */
class MyPersistentClass
{
    /** @Column(type="integer") */
    private $id;
    /** @Column(type="string") */
    private $name;
    //...
}
```

In that example we mapped the field `id` to the column `id` using the mapping type `integer` and the field `name` is mapped to the column `name` with the mapping type `string`. As you can see, by default the column names are assumed to be the same as the field names. To specify a different name for the column, you can use the `name` attribute of the Column annotation as follows:

```
<?php

/** @Column(name="db_name", type="string") */
private $name;
```

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

- `type`: 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 if a decimal column is used.)
- `scale`: (optional, default 0) The scale for a decimal (exact numeric) column. (Applies only if a decimal column is used.)

Custom Mapping Types

Doctrine allows you to create new mapping types. This can come in handy when you're missing a specific mapping type or when you want to replace the existing implementation of a mapping type.

In order to create a new mapping type you need to subclass `Doctrine\DBAL\Types\Type` and implement/override the methods as you wish. Here is an example skeleton of such a custom type class:

```
<?php

namespace My\Project\Types;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

/**
 * My custom datatype.
 */
class MyType extends Type
{
    public function getSqlDeclaration(array $fieldDeclaration,
AbstractPlatform $platform)
    {
        // return the SQL used to create your column type. To create a
portable column type, use the $platform.
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        // This is executed when the value is read from the database. Make
your conversions here, optionally using the $platform.
    }

    public function convertToDatabaseValue($value, AbstractPlatform
$platform)
    {
        // This is executed when the value is written to the database.
Make your conversions here, optionally using the $platform.
    }
}
```

When you have implemented the type you still need to let Doctrine know about it. This can be achieved through the `Doctrine\DBAL\Configuration#setCustomTypes(array $types)` method.

`Doctrine\ORM\Configuration` is a subclass of `Doctrine\DBAL\Configuration`, so the methods are available on your ORM Configuration instance as well.

Here is an example:

```php
<?php

// in bootstrapping code

// ...

use Doctrine\DBAL\Types\Type;

// ...

// Register my type
$config->setCustomTypes(array('mytype' => 'My\Project\Types\MyType'));
```

As can be seen above, when registering the custom types in the configuration you specify a unique name for the mapping type and map that to the corresponding fully qualified class name. Now you can use your new type in your mapping like this:

```php
<?php

class MyPersistentClass
{
    /** @Column(type="mytype") */
    private $field;
}
```

Identifiers / Primary Keys

Every entity class needs an identifier/primary key. You designate the field that serves as the identifier with the `@Id` marker annotation. Here is an example:

```php
<?php

class MyPersistentClass
{
    /** @Id @Column(type="integer") */
    private $id;
    //...
}
```

Without doing anything else, the identifier is assumed to be manually assigned. That means your code would need to properly set the identifier property before passing a new entity to `EntityManager#persist($entity)`.

A common alternative strategy is to use a generated value as the identifier. To do this, you use the `@GeneratedValue` annotation like this:

```php
<?php

class MyPersistentClass
{
    /**
```

```
     * @Id @Column(type="integer")
     * @GeneratedValue(strategy="AUTO")
     */
    private $id;
}
```

This tells Doctrine to automatically generate a value for the identifier. How this value is generated is specified by the `strategy` attribute, which is mandatory. A value of `AUTO` tells Doctrine to use the generation strategy that is preferred by the currently used database platform. For MySql, for example, Doctrine would use the `IDENTITY` strategy which means a typical AUTO_INCREMENT column. For PostgreSql it would choose to use the `SEQUENCE` strategy which would result in using a database sequence.

Here is the list of possible generation strategies:

- `AUTO`: Tells Doctrine to pick the strategy that is preferred by the used database platform. This strategy provides full portability.
- `SEQUENCE`: Tells Doctrine to use a database sequence for ID generation. If the used database platform does not support sequences, these will be emulated, if possible. This strategy does currently not provide full portability.
- `IDENTITY`: Tells Doctrine to use special identity columns in the database that usually generate a value on insertion of a row (i.e. MySql AUTO_INCREMENT). If the used database platform does not support identity columns, these will be emulated, if possible. This strategy does currently not provide full portability.
- `TABLE`: Tells Doctrine to use a separate table for ID generation. This strategy provides full portability. ***This strategy is not yet implemented!***

The use of the `@GeneratedValue` annotation is only supported for simple (not composite) primary keys. To designate a composite primary key / identifier, simply put the @Id marker annotation on all fields that make up the primary key.

Quoting Reserved Words

It may sometimes be necessary to quote a column or table name because it conflicts with a reserved word of the particular RDBMS in use. This is often referred to as "Identifier Quoting". To let Doctrine know that you would like a table or column name to be quoted in all SQL statements, enclose the table or column name in backticks. Here is an example:

```
<?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 is a feature that is mainly intended to support legacy database schemas. The use of reserved words and identifier quoting is generally discouraged as it is itself not free of problems.

Chapter 5
Association Mapping

This chapter explains how associations between entities are mapped with Doctrine. We start out with an explanation of the concept of owning and inverse sides which is important to understand when working with bidirectional associations. Please read these explanations carefully.

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.
- The owning side of a relationship determines the updates to the relationship in the database.

The following rules apply to bidirectional associations:

- The inverse side of a bidirectional relationship must refer to its owning side by use of the mappedBy attribute of the OneToOne, OneToMany, or ManyToMany mapping declaration. The mappedBy attribute designates the property or field in the entity that is the owner of the relationship.
- The many side of one-to-many/many-to-one bidirectional relationships must be the owning side, hence the mappedBy element cannot be specified on the ManyToOne side.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships either side may be the owning side.

Especially important is the following statement: **The owning side of a relationship determines the updates to the relationship in the database**. 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 (thats his responsiblity). Doctrine needs to know which of these 2 in-memory references is the one that should be persisted and which not. This is what the owning/inverse concept is used for. Changes made only to the inverse side of an association are ignored.

Collections

In all the examples of many-valued associations in this manual we will make use of a `Collection` interface and a corresponding default implementation `ArrayCollection` that are defined in the `Doctrine\Common\Collections` namespace. Why do we need that? Doesn't that couple my domain model to Doctrine? Unfortunately, PHP arrays, while being great for many things, do not make up for good collections of business objects, especially not in the context of an ORM. The reason is that plain PHP arrays can not be transparently extended / instrumented in PHP code, which is necessary for a lot of advanced ORM features. The classes / interfaces that come closest to an OO collection are ArrayAccess and ArrayObject but until instances of these types can be used in all places where a plain array can be used (something that may happen in PHP6) their useability is fairly limited. You "can" type-hint on `ArrayAccess` instead of `Collection`, since the Collection interface extends `ArrayAccess`, but this will severely limit you in the way you can work with the collection, because the `ArrayAccess` API is (intentionally) very primitive and more importantly because you can not pass this collection to all the useful PHP array functions, which makes it very hard to work with.

The Collection interface and ArrayCollection class, like everything else in the Doctrine\Common 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 domain classes and elsewhere does not introduce a coupling to the persistence layer. The Collection class, like everything else in the Common namespace, is not part of the persistence layer. You could even copy that class over to your project if you want to remove Doctrine from your project and all your domain classes will work the same as before.

One-To-One, Unidirectional

Here is a one-to-one relationship between a `Product` that has one `Shipping` object associated to it. The `Shipping` side does not reference back to the `Product` so it is unidirectional.

```php
<?php

/** @Entity */
class Product
{
    // ...

    /**
     * @OneToOne(targetEntity="Shipping")
     * @JoinColumn(name="shipping_id", referencedColumnName="id")
     */
    private $shipping;

    // ...
}

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

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.

```
<?php

/** @Entity */
class Customer
{
    // ...

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

    // ...
}

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

    /**
     * @OneToOne(targetEntity="Customer")
     * @JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;

    // ...
}
```

One-To-One, Self-referencing

You can easily have self referencing one-to-one relationships like below.

```
<?php

/** @Entity */
class Customer
{
    // ...

    /**
     * @OneToOne(targetEntity="Customer")
     * @JoinColumn(name="mentor_id", referencedColumnName="id")
     */
    private $mentor;
    // ...
}
```

One-To-Many, Unidirectional with Join Table

You can easily setup a one-to-many relationship between a `User` and many `Phonenumber` objects. In this example we use a reference table and each phonenumber can only be assigned to one user. This relationship is also unidirectional.

```
<?php

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

  /**
   * @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;

  // ...
}

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

One-To-Many uni-directional relations with join-table only work using the @ManyToMany annotation and a unique-constraint.

One-To-Many, Bidirectional

Here is a traditional one-to-many relationship but without using a reference table. This relationship is also bidirectional.

```
<?php

/** @Entity */
class Product
{
    // ...
    /**
```

```
     * @OneToMany(targetEntity="Feature", mappedBy="product")
     */
    private $features;
    // ...
}

/** @Entity */
class Feature
{
    // ...
    /**
     * @ManyToOne(targetEntity="Product")
     * @JoinColumn(name="product_id", referencedColumnName="id")
     */
    private $product;
    // ...
}
```

One-To-Many, Self-referencing

You can also setup a one-to-many that is self referencing. In this example we setup a hierarchy of `Category` objects by creating a self referencing relationship.

```
<?php

/** @Entity */
class Category
{
    // ...
    /**
     * @OneToMany(targetEntity="Category", mappedBy="parent")
     */
    private $children;

    /**
     * @ManyToOne(targetEntity="Category")
     * @JoinColumn(name="parent_id", referencedColumnName="id")
     */
    private $parent;
    // ...
}
```

Many-To-Many, Unidirectional

Setting up a many-to-many relationship through a reference table is easy. Below is a simple `User` has many `Group` objects example.

```
<?php

/** @Entity */
class User
{
  // ...
  /**
   * @ManyToMany(targetEntity="Group")
   * @JoinTable(name="user_groups",
   *      joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")},
   *      inverseJoinColumns={@JoinColumn(name="group_id",
referencedColumnName="id")}
   *      )
   */
  private $groups;

  // ...
}

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

Many-To-Many, Bidirectional

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

```
<?php

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

  /**
   * @ManyToMany(targetEntity="Group")
   * @JoinTable(name="user_groups",
   *      joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")},
   *      inverseJoinColumns={@JoinColumn(name="group_id",
referencedColumnName="id")}
   *      )
   */
  private $phonenumbers;

  // ...
}

/** @Entity */
class Group
{
    // ...
    /**
     * @ManyToMany(targetEntity="User", mappedBy="phonenumbers")
     */
    private $users;
    // ...
}
```

Many-To-Many, Self-referencing

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
{
  // ...

  /**
   * @ManyToMany(targetEntity="User")
   */
  private $friendsWithMe;

  /**
   * @ManyToMany(targetEntity="User")
   * @JoinTable(name="friends",
   *      joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")},
   *      inverseJoinColumns={@JoinColumn(name="friend_user_id",
referencedColumnName="id")}
   *      )
   */
  private $myFriends;

  // ...
}
```

Chapter 6

Inheritance Mapping

Mapped Superclasses

An 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.

A mapped superclass is not queryable and persistent relationships defined by a mapped superclass must be unidirectional.

Example:

```php
<?php

/** @MappedSuperclass */
class MappedSuperclassBase
{
    /** @Column(type="integer") */
    private $mapped1;
    /** @Column(type="string") */
    private $mapped2;
    /**
     * @OneToOne(targetEntity="MappedSuperclassRelated1")
     * @JoinColumn(name="related1_id", referencedColumnName="id")
     */
    private $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):

Listing 6-1
```
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[3] 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

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, @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 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`.

3. http://martinfowler.com/eaaCatalog/singleTableInheritance.html

- 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.

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 performant.

Class Table Inheritance

Class Table Inheritance[4] 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 2 implements this strategy through the use of a discriminator column in the topmost table of the hieararchy because this is the easiest way to achieve polymorphic queries with Class Table Inheritance.

Example:

```php
<?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:

4. http://martinfowler.com/eaaCatalog/classTableInheritance.html

- 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.

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.

Chapter 7

Working with objects

Understanding

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 implicity 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.

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 flush()

How costly a flush operation is in terms of performance mainly depends on 2 factors:

- The size of your current Unit of Work
- The configured change tracking policies

You can get the size of your Unit of Work 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 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.

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.

Generated entity identifiers / primary keys are guaranteed to be available after the next invocation of `EntityManager#flush()` that involves the entity in question. YOU CAN NOT RELY ON A GENERATED IDENTIFIER TO BE AVAILABLE AFTER INVOKING `persist`!

Example:

```
<?php

$user = new User;
$user->setName('Mr.Right');
$em->persist($user);
$em->flush();
// If $user had a generated identifier, it would now be available.
```

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 at or before transaction commit or 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 "Transitive Persistence").
- If X is a removed entity, it becomes managed.
- If X is a detached entity, an InvalidArgumentException will be thrown.

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. The in-memory state of an entity is unaffected by the `remove` operation.

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.

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 "Transitive Persistence").
- 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 "Transitive Persistence").
- 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 at or before transaction commit or as a result of the flush operation.

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 "Transitive Persistence"). 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 "Transitive Persistence"). 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.
```

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 or a new managed copy X' of X is created.
- If X is a new entity instance, an InvalidArgumentException will be thrown.
- 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 "Transitive Persistence").

- 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 dont 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 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.

Associations

Associations between entities are represented just like in regular object-oriented PHP, with references to other objects or collections of objects. When it comes to persistence, it is important to understand three main things:

- The concept of owning and inverse sides in bidirectional associations as described here[5].
- A collection of entities always only represents the association to the containing entities. If an entity is removed from a collection, the association is removed, not the entity itself.
- Collection-valued persistent fields and properties must be defined in terms of the Doctrine\Common\Collections\Collection interface. See here[6] for more details.

Establishing Associations

Establishing an association between two entities is straight-forward. Here are some examples:

```
<?php

// Article <- one-to-many -> Comment
$article->getComments()->add($comment);
$comment->setArticle($article);

// User <- many-to-many -> Groups
```

5. `http://www.doctrine-project.org/documentation/manual/2_0/en/association-mapping#owning-side-and-inverse-side`
6. `http://www.doctrine-project.org/documentation/manual/2_0/en/architecture#entities:persistent-fields`

```
$user->getGroups()->add($group);
$group->getUsers()->add($user);

// User <- one-to-one -> Address
$user->setAddress($address);
$address->setUser($user);
```

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

Removing Associations

Removing an association between two entities is similarly straight-forward. Here are some examples:

```
<?php

// User <- one-to-one -> Address
$user->setAddress(null);
$address->setUser(null);

// Article <- one-to-many -> Comment
$article->getComments()->remove($comment);
$comment->setArticle(null);

// User <- many-to-many -> Group
$user->getGroups()->remove($group);
$group->getUsers()->remove($user);
```

Notice how always both sides of the bidirectional association are updated. Unidirectional associations are consequently simpler to handle. Also note that if you type-hint your methods, i.e. `setAddress(Address $address)`, then PHP does not allow null values and setAddress(null) will fail for removing the association. If you insist on type-hinting a typical 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.

Since Doctrine always only looks at the owning side of a bidirectional association, it is essentially not necessary 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.

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 a simple, idiomatic example for a bidirectional one-to-many association between an Article and its Comments.

```
<?php

// Mappings not shown.
```

```
class Article {
    // The comments of the article.
    private $comments;
    // ... constructor omitted ...
    public function addComment(Comment $comment) {
        $this->comments->add($comment);
        $comment->setArticle($this);
    }
    public function getComments() {
        return $this->comments;
    }
}
class Comment {
    // The article the comment refers to.
    private $article;
    // ... constructor omitted ...
    public function setArticle($article) {
        $this->article = $article;
    }
    public function getArticle() {
        return $this->article;
    }
}
```

With the above implementation, it is always ensured that at least the owning side from Doctrine's point of view (Comment) is properly updated. You will notice that `setArticle` does not call `addComment`, thus the bidirectional association is strictly-speaking still incomplete, if a user of the class only invokes `setArticle`. If you naively call `addComment` in `setArticle`, however, 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.

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.

Transitive persistence

Persisting, removing, detaching and merging individual entities can become pretty cumbersome, especially when a larger object graph with collections is involved. Therefore Doctrine 2 provides a mechanism for transitive persistence through cascading of these operations. Each association to another entity or a collection of entities can be configured to automatically cascade certain operations. By default, no operations are cascaded.

The following cascade options exist:

- persist : Cascades persist operations to the associated entities.
- remove : Cascades remove operations to the associated entities.
- merge : Cascades merge operations to the associated entities.
- detach : Cascades detach operations to the associated entities.
- all : Cascades persist, remove, merge and detach operations to associated entities.

The following example shows an association to a number of addresses. If persist() or remove() is invoked on any User entity, it will be cascaded to all associated Address entities in the $addresses collection.

```
<?php

class User
{
    //...
    /**
     * @OneToMany(targetEntity="Address", mappedBy="owner",
cascade={"persist", "remove"})
     */
    private $addresses;
    //...
}
```

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.

Querying

Doctrine 2 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 retreive 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'));
```

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');
```

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 familar 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\ORM\Query class can be found in the dedicated chapter[7]. For programmatically building up queries based on conditions that are only known at runtime, Doctrine provides the special `Doctrine\ORM\QueryBuilder` class. More information on constructing queries with a QueryBuilder can be found in the dedicated chapter[8].

Custom Repositories

[TBD]

7. `http://www.doctrine-project.org/documentation/manual/2_0/en/dql-doctrine-query-language`
8. `http://www.doctrine-project.org/documentation/manual/2_0/en/query-builder`

Chapter 8

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 have a negative effect on 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 that is immediately committed. Without any explicit transaction demarcation from your side, this quickly results in poor performance because transactions are not cheap and many small transactions degrade the performance of your application.

For the most part, Doctrine 2 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, small transaction. This is a strategy called "transactional write-behind" that is frequently used in ORM solutions to increase efficiency.

However, Doctrine 2 also allows you to take over and control transaction demarcation yourself, thereby "widening" the transaction boundaries. This is possible due to transparent nesting of transactions that is described in the following section.

Transaction Nesting

Each `Doctrine\DBAL\Driver\Connection` instance is wrapped in a `Doctrine\DBAL\Connection` that adds support for transparent nesting of transactions. For that purpose, the Connection class keeps an internal counter that represents the nesting level and is increased/decreased as beginTransaction(), commit() and rollback() are invoked. beginTransaction() increases the nesting level whilst commit() and rollback() decrease the nesting level. The nesting level starts at 0. Whenever the nesting level transitions from 0 to 1, beginTransaction() is invoked on the underlying driver and whenever the nesting level transitions from 1 to 0, commit() or rollback() is invoked on the underlying driver, depending on whether the transition was caused by `Connection#commit()` or `Connection#rollback()`.

Lets visualize what that means in practice. It means that the first call to `Doctrine\DBAL\Connection#beginTransaction()` will increase the nesting level from 0 to 1 and invoke beginTransaction() on the underlying driver, effectively starting a "real" transaction by suspending auto-commit mode. Any subsequent, nested calls to `Doctrine\DBAL\Connection#beginTransaction()` would only increase the nesting level.

Here is an example to help visualize how this works:

```php
<?php

// $conn instanceof Doctrine\DBAL\Connection
try {
    $conn->beginTransaction(); // 0 => 1, "real" transaction started

    ...

    try {
        $conn->beginTransaction(); // 1 => 2

        ...

        $conn->commit(); // 2 => 1
    } catch (Exception $e) {
        $conn->rollback(); // 2 => 1
        throw $e;
    }

    ...

    $conn->commit(); // 1 => 0, "real" transaction committed
} catch (Exception $e) {
    $conn->rollback(); // 1 => 0, "real" transaction rollback
    throw $e;
}
```

What is the benefit of this? It allows reliable and transparent widening of transaction boundaries. Given the following code snippet, without any explicit transaction demarcation:

```php
<?php

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

Inside `EntityManager#flush()` something like this happens:

```php
<?php

try {
    $conn->beginTransaction(); // suspend auto-commit

    ... commit all changes to the database ...

    $conn->commit();
} catch (Exception $e) {
    $conn->rollback();
    throw $e;
}
```

Since we do not do any custom transaction demarcation in the first snippet, `EntityManager#flush()` will begin and commit/rollback a "real" transaction. Now, if we want to widen the transaction boundaries, say, because we want to include some manual work with a `Doctrine\DBAL\Connection` in the same transaction, we can simply do this:

```php
<?php

// $em instanceof EntityManager
$conn = $em->getConnection();
try {
    $conn->beginTransaction(); // suspend auto-commit

    // Direct use of the Connection
    $conn->insert(...);

    $user = new User;
    $user->setName('George');
    $em->persist($user);
    $em->flush();

    $conn->commit();
} catch (Exception $e) {
    $conn->rollback();
    // handle or rethrow
}
```

Now, our own code controls the "real" transaction and the transaction demarcation that happens inside `EntityManager#flush()` will merely affect the nesting level. When flush() returns, either by throwing an exception or regularly, the nesting level is the same as before the invocation of flush(), in this case 1, and thus our own $conn->commit() / $conn->rollback() affect the "real" transaction as expected, since we were the ones who started the transaction.

Directly invoking `PDO#beginTransaction()`, `PDO#commit()` or `PDO#rollback()` or the corresponding methods on the particular `Doctrine\DBAL\Driver\Connection` instance in use bybasses the transparent transaction nesting that is provided by `Doctrine\DBAL\Connection` and can therefore corrupt the nesting level, causing errors with broken transaction boundaries that may be hard to debug.

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 dont 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;
    // ...
}
```

You could also just as easily use a datetime column and instead of incrementing an integer, a timestamp will be kept up to date.

```php
<?php

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

Chapter 9

Events

Doctrine 2 features a lightweight event system that is part of the Common package.

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
<?php

$evm = new EventManager();
```

Now we can add some event listeners to the `$evm`. Lets create a `EventTest` class to play around with.

```php
<?php

class EventTest
{
    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
$test = new EventTest($evm);
```

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

```
<?php

$evm->dispatchEvent(EventTest::preFoo);
$evm->dispatchEvent(EventTest::postFoo);
```

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

```
<?php

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

The Doctrine 2 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
{
    const preFoo = 'preFoo';

    public $preFooInvoked = false;

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

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

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

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

```
<?php

$evm->dispatchEvent(TestEventSubscriber::preFoo);
```

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

```php
<?php

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

Lifecycle Events

A lifecycle event is a regular event with the additional feature of providing a mechanism to register direct callbacks inside the corresponding entity classes that are executed when the lifecycle event occurs. An event that is also a lifecycle event is specifically designated as such in the API description.

```php
<?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:m: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 when using 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.

Listing 9-1

```
---
User:
  type: entity
  fields:
# ...
    name:
      type: string(50)
  lifecycleCallbacks:
    doStuffOnPrePersist: prePersist
    doStuffOnPostPersist: postPersist
```

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
                          /Users/robo/dev/php/Doctrine/
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>
```

You just need to make sure a public `doStuffOnPrePersist()` and `doStuffOnPostPersist()` method is defined on your `User` model.

```
<?php

// ...

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

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

The `key` of the lifecycleCallbacks is the name of the method and the value is the event type. The allowed event types are listed below.

- preRemove - The preRemove event occurs for a given entity before the respective EntityManager remove operation for that entity is executed.
- postRemove - The postRemove event occurs for an entity after the entity has been deleted. It will be invoked after the database delete operations.
- prePersist - The prePersist event occurs for a given entity before the respective EntityManager persist operation for that entity is executed.
- 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.
- postUpdate - The postUpdate event occurs after the database update operations to entity data.
- 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).

Note that the postLoad event occurs for an entity before any associations have been initialized. Therefore it is not safe to access associations in a postLoad callback or event handler.

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

```
<?php

use Doctrine\ORM\Events;
echo Events::preUpdate;
```

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 EventTest();
$metadataFactory = $em->getMetadataFactory();
$evm = $em->getEventManager();
$evm->addEventListener(Events::loadClassMetadata, $test);

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

Chapter 10
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.

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
<?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!
    }
}
```

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
<?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#iterate()` 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
<?php

$batchSize = 20;
$i = 0;
$q = $em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
foreach($iterableResult AS $row) {
    $user = $row[0];
    $user->increaseCredit();
    $user->calculateNewBonuses();
    if (($i % $batchSize) == 0) {
        $em->flush(); // Executes all updates.
        $em->clear(); // Detaches all objects from Doctrine!
    }
    ++$i;
}
```

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.

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
<?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#iterate()` 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 = 0;
$q = $em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
while (($row = $iterableResult->next()) !== false) {
    $em->remove($row[0]);
    if (($i % $batchSize) == 0) {
        $em->flush(); // Executes all deletions.
        $em->clear(); // Detaches all objects from Doctrine!
    }
    ++$i;
}
```

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.

Chapter 11

DQL (Doctrine Query Language)

DQL Explained

DQL stands for **D**octrine **Q**uery **L**anguage and is an Object Query Language derivate that is very similar to the **H**ibernate **Q**uery **L**anguage (HQL) or the **J**ava **P**ersistence **Q**uery **L**anguage (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

SELECT/UPDATE/DELETE...

SELECT queries

DQL SELECT clause

The select clause of a DQL query specifies what appears in the query result. The composition of all the expressions in the select clause also influences the nature of the query result.

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

Listing 11-1

```
SELECT
u
FROM MyProject\Model\User u
WHERE u.age > 20
```

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 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.

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: select u from User u join u.address a where a.city = 'Berlin' Fetch join of the address: select u, a from User u join u.address a where a.city = 'Berlin'

DQL SELECT Examples

Listing 11-2

```
SELECT
u
FROM MyProject\Model\User u
[MORE TO COME]
```

UPDATE queries

xxx

DELETE queries

xxx

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: usin 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 an 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 exception is thrown. The pure/mixed distinction does not apply.
- `Query#getArrayResult()`: Retrieves an array graph (a nested array) that is largely interchangeable with the object graph generated by `Query#getResultList()` for read-only purposes.

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#getResultList()` 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 accomodate for the scalar values.

A pure result usually looks like this:

Listing 11-3

```
array
    [0] => Object
    [1] => Object
    [2] => Object
    ...
```

A mixed result on the other hand has the following general structure:

Listing 11-4

```
array
    array
        [0] => Object
        [1] => "some scalar string"
        ['count'] => 42
        // ... more scalar values, either indexed numerically or with a
name
    array
        [0] => Object
        [1] => "some scalar string"
        ['count'] => 42
        // ... more scalar values, either indexed numerically or with a
name
```

To better understand mixed results, consider the following DQL query:

Listing 11-5

```
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.

Here is how the result could look like:

Listing 11-6

```
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'];
}
```

You may have observed that in a mixed result, the object always ends up on index 0 of a result row.

Functions

xxx

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, eg. zero or one time
- curly brackets {...} are used for repetion, eg. zero or more times
- double quotation marks "..." define a terminal string a vertical bar | represents an alternative

Terminals

- identifier (name, email, ...)
- 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

Listing 11-7

```
QueryLanguage ::= SelectStatement | UpdateStatement | DeleteStatement
```

Statements

Listing 11-8

```
SelectStatement ::= SelectClause FromClause [WhereClause] [GroupByClause]
[HavingClause] [OrderByClause]
UpdateStatement ::= UpdateClause [WhereClause]
DeleteStatement ::= DeleteClause [WhereClause]
```

Identifiers

Listing 11-9

```
/* 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") */
```

```
AbstractSchemaName ::= identifier

/* identifier that must be a field (the "name" of "u.name") */
/* This is responsable 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 (for the future)
*/
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

/* 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
```

Path Expressions

Listing 11-10

```
/* "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 | SingleValuedAssociationPathExpression "." StateField

/* "u.Group" */
SingleValuedAssociationPathExpression     ::= IdentificationVariable "."
{SingleValuedAssociationField "."}* SingleValuedAssociationField

/* "u.Group.Permissions" */
CollectionValuedPathExpression            ::= IdentificationVariable "."
```

```
{SingleValuedAssociationField "."}* CollectionValuedAssociationField

/* "name" */
StateField                                ::= {EmbeddedClassStateField
"."}* SimpleStateField

/* "u.name" or "u.address.zip" (address = EmbeddedClassStateField) */
SimpleStateFieldPathExpression            ::= IdentificationVariable "."
StateField
```

Clauses

Listing 11-11

```
SelectClause        ::= "SELECT" ["ALL" | "DISTINCT"] SelectExpression
{"," SelectExpression}*
SimpleSelectClause  ::= "SELECT" ["ALL" | "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

Listing 11-12

```
UpdateItem  ::= IdentificationVariable "." (StateField |
SingleValuedAssociationField) "=" NewValue
OrderByItem ::= (ResultVariable | StateFieldPathExpression) ["ASC" |
"DESC"]
GroupByItem ::= IdentificationVariable | SingleValuedPathExpression
NewValue    ::= SimpleArithmeticExpression | StringPrimary |
DatetimePrimary | BooleanPrimary |
                EnumPrimary | SimpleEntityExpression | "NULL"
```

From, Join and Index by

Listing 11-13

```
IdentificationVariableDeclaration          ::= RangeVariableDeclaration
[IndexBy] {JoinVariableDeclaration}*
SubselectIdentificationVariableDeclaration ::=
IdentificationVariableDeclaration | (AssociationPathExpression ["AS"]
AliasIdentificationVariable)
JoinVariableDeclaration                    ::= Join [IndexBy]
RangeVariableDeclaration                   ::= AbstractSchemaName ["AS"]
AliasIdentificationVariable
Join                                       ::= ["LEFT" ["OUTER"] |
"INNER"] "JOIN" JoinAssociationPathExpression
                                               ["AS"]
AliasIdentificationVariable [("ON" | "WITH") ConditionalExpression]
```

```
IndexBy                                    ::= "INDEX" "BY"
SimpleStateFieldPathExpression
```

Select Expressions

Listing 11-14

```
SelectExpression       ::= IdentificationVariable |
StateFieldPathExpression |
                           (AggregateExpression | "(" Subselect ")"  |
FunctionDeclaration) [["AS"] AliasResultVariable]
SimpleSelectExpression ::= StateFieldPathExpression |
IdentificationVariable |
                           (AggregateExpression [["AS"]
AliasResultVariable])
```

Conditional Expressions

Listing 11-15

```
ConditionalExpression       ::= ConditionalTerm {"OR" ConditionalTerm}*
ConditionalTerm             ::= ConditionalFactor {"AND"
ConditionalFactor}*
ConditionalFactor           ::= ["NOT"] ConditionalPrimary
ConditionalPrimary          ::= SimpleConditionalExpression | "("
ConditionalExpression ")"
SimpleConditionalExpression ::= ComparisonExpression | BetweenExpression |
LikeExpression |
                                InExpression | NullComparisonExpression |
ExistsExpression |
                                EmptyCollectionComparisonExpression |
CollectionMemberExpression
```

Collection Expressions

Listing 11-16

```
EmptyCollectionComparisonExpression ::= CollectionValuedPathExpression
"IS" ["NOT"] "EMPTY"
CollectionMemberExpression          ::= EntityExpression ["NOT"] "MEMBER"
["OF"] CollectionValuedPathExpression
```

Literal Values

Listing 11-17

```
Literal     ::= string | char | integer | float | boolean
InParameter ::= Literal | InputParameter
```

Input Parameter

Listing 11-18

```
InputParameter      ::= PositionalParameter | NamedParameter
PositionalParameter ::= "?" integer
NamedParameter      ::= ":" string
```

Arithmetic Expressions

Listing 11-19

```
ArithmeticExpression       ::= SimpleArithmeticExpression | "(" Subselect
")"
SimpleArithmeticExpression ::= ArithmeticTerm {("+" | "-") ArithmeticTerm}*
ArithmeticTerm             ::= ArithmeticFactor {("*" | "/")
ArithmeticFactor}*
```

```
ArithmeticFactor           ::= [("+" | "-")] ArithmeticPrimary
ArithmeticPrimary          ::= SingleValuedPathExpression | Literal | "("
SimpleArithmeticExpression ")"
                               | FunctionsReturningNumerics |
AggregateExpression | FunctionsReturningStrings
                               | FunctionsReturningDatetime |
IdentificationVariable | InputParameter
```

Data Type Expressions

Listing 11-20

```
StringExpression       ::= StringPrimary | "(" Subselect ")"
StringPrimary          ::= StateFieldPathExpression | string |
InputParameter | FunctionsReturningStrings | AggregateExpression
BooleanExpression      ::= BooleanPrimary | "(" Subselect ")"
BooleanPrimary         ::= StateFieldPathExpression | boolean |
InputParameter
EnumExpression         ::= EnumPrimary | "(" Subselect ")"
EnumPrimary            ::= StateFieldPathExpression | string |
InputParameter
EntityExpression       ::= SingleValuedAssociationPathExpression |
SimpleEntityExpression
SimpleEntityExpression ::= IdentificationVariable | InputParameter
DatetimeExpression     ::= DatetimePrimary | "(" Subselect ")"
DatetimePrimary        ::= StateFieldPathExpression | InputParameter |
FunctionsReturningDatetime | AggregateExpression
```

Aggregate Expressions

Listing 11-21

```
AggregateExpression ::= ("AVG" | "MAX" | "MIN" | "SUM") "(" ["DISTINCT"]
StateFieldPathExpression ")" |
                        "COUNT" "(" ["DISTINCT"] (IdentificationVariable |
SingleValuedPathExpression) ")"
```

Other Expressions

QUANTIFIED/BETWEEN/COMPARISON/LIKE/NULL/EXISTS

Listing 11-22

```
QuantifiedExpression     ::= ("ALL" | "ANY" | "SOME") "(" Subselect ")"
BetweenExpression        ::= ArithmeticExpression ["NOT"] "BETWEEN"
ArithmeticExpression "AND" ArithmeticExpression
ComparisonExpression     ::= ArithmeticExpression ComparisonOperator (
QuantifiedExpression | ArithmeticExpression )
InExpression             ::= StateFieldPathExpression ["NOT"] "IN" "("
(InParameter {"," InParameter}* | Subselect) ")"
LikeExpression           ::= StringExpression ["NOT"] "LIKE" string
["ESCAPE" char]
NullComparisonExpression ::= (SingleValuedPathExpression | InputParameter)
"IS" ["NOT"] "NULL"
ExistsExpression         ::= ["NOT"] "EXISTS" "(" Subselect ")"
ComparisonOperator       ::= "=" | "<" | "<=" | "<>" | ">" | ">=" | "!="
```

Functions

Listing 11-23

```
FunctionDeclaration ::= FunctionsReturningStrings |
FunctionsReturningNumerics | FunctionsReturningDateTime
FunctionsReturningNumerics ::=
        "LENGTH" "(" StringPrimary ")" |
        "LOCATE" "(" StringPrimary "," StringPrimary [","
SimpleArithmeticExpression]")" |
        "ABS" "(" SimpleArithmeticExpression ")" | "SQRT" "("
SimpleArithmeticExpression ")" |
        "MOD" "(" SimpleArithmeticExpression ","
SimpleArithmeticExpression ")" |
        "SIZE" "(" CollectionValuedPathExpression ")"

FunctionsReturningDateTime ::= "CURRENT_DATE" | "CURRENT_TIME" |
"CURRENT_TIMESTAMP"

FunctionsReturningStrings ::=
        "CONCAT" "(" StringPrimary "," StringPrimary ")" |
        "SUBSTRING" "(" StringPrimary "," SimpleArithmeticExpression ","
SimpleArithmeticExpression ")" |
        "TRIM" "(" [["LEADING" | "TRAILING" | "BOTH"] [char] "FROM"]
StringPrimary ")" |
        "LOWER" "(" StringPrimary ")" |
        "UPPER" "(" StringPrimary ")"
```

Chapter 12

Query Builder

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 programatically build you queries, and also provides a fluent API. This means that you can change between one methodology to the other as you want, and also pick one if you prefer.

Constructing a new QueryBuilder object

The same way you build a normal Query, you build a `QueryBuilder` object, just providing the correct method name. Here is an example how to build a `QueryBuilder` object:

```
<?php

// $em instanceof EntityManager

// example1: creating a QueryBuilder instance
$qb = $em->createQueryBuilder();
```

Once you created an instance of QueryBuilder, it provides a set of useful informative functions that you can use. 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, which prevents multiple processment if called multiple times. Any changes that may affect the generated DQL actually modifies the state of `QueryBuilder` to a stage we call as STATE_DIRTY. One `QueryBuilder`can be in two different state:

- `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

All helper methods in `QueryBuilder` relies actually on a single one: `add()`. This method is the responsable to build 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

```
<?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
<?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'));
```

Of course this is the hardest way to build a DQL query in Doctrine. To simplify some of these efforts, we introduce what we call as `Expr` helper class.

The Expr class

To workaround most of the issues that `add()` method may cause, Doctrine created a class that can be considered as a helper for building queries. This class is called `Expr`, which provides a set of useful static methods to help building queries:

```php
<?php

// $qb instanceof QueryBuilder

// example8: QueryBuilder port of: "SELECT u FROM User u WHERE u.id = ? OR
u.nickname LIKE ? ORDER BY u.surname DESC" using Expr class
$qb->add('select', $qb->expr()->select('u'))
   ->add('from', $qb->expr()->from('User', 'u'))
   ->add('where', $qb->expr()->orx(
       $qb->expr()->eq('u.id', '?1'),
       $qb->expr()->like('u.nickname', '?2')
   ))
   ->add('orderBy', $qb->expr()->orderBy('u.surname', 'ASC'));
```

Although it still sounds complex, the ability to programatically create conditions are the main feature of `Expr`. Here it is a complete list of supported helper methods available:

```php
<?php

class Expr
{
    /** Base objects **/

    // Example usage - $qb->expr()->select('u')
    public function select($select = null); // Returns Expr\Select instance

    // Example - $qb->expr()->from('User', 'u')
    public function from($from, $alias); // Returns Expr\From instance

    // Example - $qb->expr()->leftJoin('u.Phonenumbers', 'p',
Expr\Join::ON, 'p.user_id = u.id AND p.country_code = 55');
    // Example - $qb->expr()->leftJoin('u. Phonenumbers', 'p', 'ON',
$qb->expr()->andx($qb->expr()->eq('p.user_id', 'u.id'),
$qb->expr()->eq('p.country_code', '55'));
    public function leftJoin($join, $alias, $conditionType = null,
$condition = null); // Returns Expr\Join instance
```

```
    // Example - $qb->expr()->innerJoin('u.Group', 'g', Expr\Join::WITH,
'g.manager_level = 100');
    // Example - $qb->expr()->innerJoin('u.Group', 'g', 'WITH',
$qb->expr()->eq('g.manager_level', '100'));
    public function innerJoin($join, $alias, $conditionType = null,
$condition = null); // Returns Expr\Join instance

    // Example - $qb->expr()->orderBy('u.surname',
'ASC')->add('u.firstname', 'ASC')->...
    public function orderBy($sort = null, $order = null); // Returns
Expr\OrderBy instance

    // Example - $qb->expr()->groupBy()->add('u.id')->...
    public function groupBy($groupBy = null); // Returns Expr\GroupBy
instance


    /** 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


    /** 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 - $qb->expr()->in('u.id', array(1, 2, 3))
    public function in($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 - $qb->expr()->between('u.id', '1', '10')
    public function between($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', $qb->expr()->concat('
', 'u.lastname'))
    public function concat($x, $y); // Returns Expr\Func

    // Example - $qb->expr()->substr('u.firstname', 0, 1)
    public function substr($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
}
```

Helper methods

Until now it was described the hardcore level of creating queries. It may be useful to work that way for optimization purposes, but most of the time it is preferred to work higher level. To simplify even more the way you build a query in Doctrine, we can take advantage of what we call as helper methods. For all base code, it has a set of useful methods to simplify programmer's life. Illustrating how to work with it, here is the same example 6 written now using `QueryBuilder` helper methods:

```
<?php

// $qb instanceof QueryBuilder

// example9: how to define: "SELECT u FROM User u WHERE u.id = ?1 ORDER BY
u.name ASC" using QueryBuilder helper methods
$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name ASC');
```

`QueryBuilder` helper methods are considered the standard way to build DQL queries. Although it is supported, it should be avoided to use string based queries and greatly encouraged to use `$qb->expr()->*` methods. Here is a converted example 8 to suggested standard way to build queries:

```
<?php

// $qb instanceof QueryBuilder

// example8: QueryBuilder port of: "SELECT u FROM User u WHERE u.id = ?1
OR u.nickname LIKE ?2 ORDER BY u.surname DESC" using QueryBuilder helper
methods
$qb->select(array('u')) // string 'u' is converted to array internally
   ->from('User', 'u')
   ->where($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 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);

    // 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 - $qb->from('Phonenumber', 'p')
    public function from($from, $alias = null);

    // Example - $qb->innerJoin('u.Group', 'g', Expr\Join::ON,
$qb->expr()->and($qb->expr()->eq('u.group_id', 'g.id'), 'g.name = ?1'))
    // Example - $qb->innerJoin('u.Group', 'g', 'ON', 'u.group_id = g.id
AND g.name = ?1')
    public function innerJoin($join, $alias = null, $conditionType = null,
$condition = null);

    // Example - $qb->leftJoin('u.Phonenumbers', 'p', Expr\Join::WITH,
$qb->expr()->eq('p.area_code', 55))
    // Example - $qb->leftJoin('u.Phonenumbers', 'p', 'WITH', 'p.area_code
= 55')
    public function leftJoin($join, $alias = null, $conditionType = null,
$condition = 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);

    // 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 items
    //
    // 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 items
    //
    // 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'
}
```

Chapter 13

Native SQL

A `NativeQuery` lets you execute native SQL, mapping the results according to your specifications. Such a specification that describes how an SQL result set is mapped to a Doctrine result is represented by a `ResultSetMapping`.

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 and finally execute it.

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 becomed **mixed** (see DQL - Doctrine Query Language) if the same ResultSetMapping also contains entity results.

It might not surprise you that Doctrine uses `ResultSetMapping`s 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 an 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 an 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.

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

$rsm = new ResultSetMapping;
$rsm->addEntityResult('Doctrine\Tests\Models\CMS\CmsUser', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');

$query = $this->_em->createNativeQuery('SELECT id, name FROM cms_users
WHERE username = ?', $rsm);
$query->setParameter(1, 'romanb');

$users = $query->getResult();
```

The result would look like this:

Listing 13-1

```
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.

Chapter 14

XML Mapping

The XML mapping driver enables you to provide the ORM metadata in form of XML documents.

The XML driver is backed by an XML Schema document that describes the structure of a mapping document. The most recent version of the XML Schema document is available online at http://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd[9]. In order to point to the latest version of the document of a particular stable release branch, just append the release number, i.e.: doctrine-mapping-2.0.xsd The most convenient way to work with XML mapping files is to use an IDE/editor that can provide code-completion based on such an XML Schema document. The following is an outline of a XML mapping document with the proper xmlns/xsi setup for the latest code in trunk.

```
<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
                    http://doctrine-project.org/schemas/orm/
doctrine-mapping.xsd">

    ...

</doctrine-mapping>
```

The XML mapping document of a class is loaded on-demand the first time it is requested and subsequently stored in the metadata cache. In order to work, this requires certain conventions:

- Each entity/mapped superclass must get its own dedicated XML mapping document.
- The name of the mapping document must consist of the fully qualified name of the class, where namespace separators are replaced by dots (.).
- All mapping documents should get the extension ".dcm.xml" to identify it as a Doctrine mapping file. This is more of a convention and you are not forced to do this. You can change the file extension easily enough.

```
<?php

$driver->setFileExtension('.xml');
```

9. http://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd

It is recommended to put all XML mapping documents in a single folder but you can spread the documents over several folders if you want to. In order to tell the XmlDriver where to look for your mapping documents, supply an array of paths as the first argument of the constructor, like this:

```
<?php

// $config instanceof Doctrine\ORM\Configuration
$driver = new XmlDriver(array('/path/to/files'));
$config->setMetadataDriverImpl($driver);
```

Example

As a quick start, here is a small example document that makes use of several common elements:

```
// Doctrine.Tests.ORM.Mapping.User.dcm.xml
<?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
                          /Users/robo/dev/php/Doctrine/
doctrine-mapping.xsd">

    <entity name="Doctrine\Tests\ORM\Mapping\User" table="cms_users">

        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist" method="onPrePersist" />
        </lifecycle-callbacks>

        <id name="id" type="integer" column="id">
            <generator strategy="AUTO"/>
        </id>

        <field name="name" column="name" type="string" length="50"/>

        <one-to-one field="address" target-entity="Address">
            <join-column name="address_id" referenced-column-name="id"/>
        </one-to-one>

        <one-to-many field="phonenumbers" target-entity="Phonenumber"
mapped-by="user">
            <cascade>
                <cascade-persist/>
            </cascade>
        </one-to-many>

        <many-to-many field="groups" target-entity="Group">
            <join-table name="cms_users_groups">
                <join-columns>
                    <join-column name="user_id"
referenced-column-name="id"/>
```

```
                </join-columns>
                <inverse-join-columns>
                    <join-column name="group_id"
referenced-column-name="id"/>
                </inverse-join-columns>
            </join-table>
        </many-to-many>

    </entity>
</doctrine-mapping>
```

Be aware that class-names specified in the XML files should be fully qualified.

Chapter 15

YAML Mapping

The YAML mapping driver enables you to provide the ORM metadata in form of YAML documents.

The YAML mapping document of a class is loaded on-demand the first time it is requested and subsequently stored in the metadata cache. In order to work, this requires certain conventions:

- Each entity/mapped superclass must get its own dedicated YAML mapping document.
- The name of the mapping document must consist of the fully qualified name of the class, where namespace separators are replaced by dots (.).
- All mapping documents should get the extension ".dcm.yml" to identify it as a Doctrine mapping file. This is more of a convention and you are not forced to do this. You can change the file extension easily enough.

```
<?php

$driver->setFileExtension('.yml');
```

It is recommended to put all YAML mapping documents in a single folder but you can spread the documents over several folders if you want to. In order to tell the YamlDriver where to look for your mapping documents, supply an array of paths as the first argument of the constructor, like this:

```
<?php

// $config instanceof Doctrine\ORM\Configuration
$driver = new YamlDriver(array('/path/to/files'));
$config->setMetadataDriverImpl($driver);
```

Example

As a quick start, here is a small example document that makes use of several common elements:

Listing 15-1

```
---
# Doctrine.Tests.ORM.Mapping.User.dcm.yml
Doctrine\Tests\ORM\Mapping\User:
  type: entity
  table: cms_users
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
      length: 50
  oneToOne:
    address:
      targetEntity: Address
      joinColumn:
        name: address_id
        referencedColumnName: id
  oneToMany:
    phonenumbers:
      targetEntity: Phonenumber
      mappedBy: user
      cascade: cascadePersist
  manyToMany:
    groups:
      targetEntity: Group
      joinTable:
        name: cms_users_groups
        joinColumns:
          user_id:
            referencedColumnName: id
        inverseJoinColumns:
          group_id:
            referencedColumnName: id
  lifecycleCallbacks:
    doStuffOnPrePersist: prePersist
    doStuffOnPostPersist: postPersist
```

Be aware that class-names specified in the YAML files should be fully qualified.

Chapter 16
Annotations Reference

In this chapter a reference of every Doctrine 2 Annotation is given with short explanations on their context and usage.

Index

Reference

@Column

Marks an annotated instance variable as "persistant". 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:

- length - Used by the "string" type to determine its maximum length in the database. Doctrine does not validate the length of a string values for you.
- precision - The precision for a decimal (exact numeric) column (Applies only for decimal column)
- scale - The scale for a decimal (exact numeric) column (Applies only for decimal column)
- unique - Boolean value to determine if the value of the column should be unique accross all rows of the underlying entities table.
- nullable - Determines if NULL values allowed for this column.
- columnDefinition - DDL SQL snippet that starts after the column name and specificies the complete (non-portable!) column definition. This attribute allows to make use of advanced RMDBS features.

Examples:

Listing 16-1

```
/**
 * @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;
```

@ChangeTrackingPolicy

Optional Annotation in the Entity Class PHP DocBlock.

@DiscrimnatorColumn

@DiscriminatorMap

@Entity

Required annotation to mark a PHP class as Entity. Doctrine manages the persistence of all classes marked as entity.

Optional attributes:

- repositoryClass - Specifies the FQCN of a subclass of the Doctrine\ORM\EntityRepository. Use of repositories for entites is encouraged to keep specialized DQL and SQL operations separated from the Model/Domain Layer.

Example:

Listing 16-2

```
/**
 * @Entity(repositoryClass="MyProject\UserRepository")
 */
class User
{
    //...
}
```

@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.

Required attributes:

- strategy - Set the name of the identifier generation strategy. Valid values are AUTO, SEQUENCE, TABLE, IDENTITY and NONE.

Example:

Listing 16-3

```
/**
 * @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 life-cycle 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:

Listing 16-4

```
/**
 * @Entity
 * @HasLifecycleCallbacks
 */
class User
{
    /**
     * @PostPersist
     */
    public function sendOptinMail() {}
}
```

@Index

Annotation is used inside the @Table annotation on the entity-class level. It allows to hint 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.

Example:

Listing 16-5
```
/**
 * @Entity
 * @Table(name="ecommerce_products",indexes={@index(name="search_idx",
columns={"name", "email"})})
 */
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 entites that have multiple identifier columns each column has to be marked with @Id.

Example:

Listing 16-6
```
/**
 * @Id
 * @Column(type="integer")
 */
protected $id = null;
```

@InheritanceType

@JoinColumn

This annotation is used in the context of relations. For the owning side of @ManyToOne and @OneToOne fields this annotation is required. In the Context of @JoinTable nested inside a @ManyToMany this annotation is used to describe the owning and inverse side join columns of the join-table.

Required 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.

Optional attributes:

- unique - Determines if this relation exclusive between the affected entities and should be enforced so on the database constraint level. Defaults to false.
- nullable - Determine if the related entity is required, or if null is an allowed state for the relation. Defaults to true.
- onDelete - Cascade Action (Database-level)

- onUpdate - Cascade Action (Database-level)

Example:

Listing 16-7

```
/**
 * @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 specifiy the @JoinTable annotation which describes the details of the database join table.

Required attributes:

- name - Database name of the join-table
- joinColumns - An array of @JoinColumn annotations describing the join-relation between the owning entites table and the join table.
- inverseJoinColumns - An array of @JoinColumn annotations describing the join-relation between the inverse entities table and the join table.

Optional attributes:

- schema - Database schema name of this table.

Example:

Listing 16-8

```
/**
 * @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.

Optional attributes:

- cascade - Cascade Option
- fetch - One of LAZY or EAGER

Example:

Listing 16-9

```
/**
 * @ManyToOne(targetEntity="Cart", cascade="ALL", fetch="EAGER")
 */
private $cart;
```

@ManyToMany

Defines an instance variable holds a many-to-many relationship between two entities.

Required attributes:

- targetEntity - FQCN of the referenced target entity.

Optional attributes:

- 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.
- cascade - Cascade Option
- fetch - One of LAZY or EAGER

Example:

Listing 16-10

```
/**
 * Owning Side
 *
 * @ManyToMany(targetEntity="Group")
 * @JoinTable(name="user_groups",
 *      joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")},
 *      inverseJoinColumns={@JoinColumn(name="group_id",
referencedColumnName="id")}
 *      )
 */
private $phonenumbers;

/**
 * Inverse Side
 *
 * @ManyToMany(targetEntity="Feature", mappedBy="product")
 */
private $features;
```

@MappedSuperclass

@OnetoOne

@OneToMany

@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 the use with @generatedValue(strategy="SEQUENCE") this annotation allows to specifiy details about the sequence, such as the increment size and initial values of the sequence.

Required attributes:

- name - Name of the sequence

Optional attributes:

- allocationSize - Increment the sequence by the allocation size when its fetched. A value larger than 1 allows to optimize for scenarios where you create more than one new entity per request. Defaults to 10
- initialValue - Where does the sequence start, defaults to 1.

Example:

Listing 16-11

```
/**
 * @Id
 * @GeneratedValue(strategy="SEQUENCE")
 * @Column(type="integer")
 * @SequenceGenerator(name="tablename_seq", initialValue=1,
allocationSize=100)
 */
protected $id = null;
```

@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 entities unqualified classname.

Required attributes:

- name - Name of the table

Optional attributes:

- schema - Database schema name of this table.
- indexes - Array of @Index annotations
- uniqueConstraints - Array of @UniqueConstraint annotations.

Example:

Listing 16-12

```
/**
 * @Entity
 * @Table(name="user",
 *
uniqueConstraints={@UniqueConstraint(name="user_unique",columns={"username"})},
 *      indexes={@Index(name="user_idx", columns={"email"})}
 * )
 */
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.

Example:

Listing 16-13

```
/**
 * @Entity
 *
@Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx"
columns={"name", "email"})})
 */
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.

Example:

Listing 16-14

```
/**
 * @column(type="integer")
 * @version
 */
protected $version;
```

Chapter 17

Caching

Doctrine provides cache drivers in the `Common` package for some of the most popular caching implementations such as APC, Memcache and Xcache. We also provide an `ArrayCache` driver which stores the data in a PHP array. Obviously, the cache does not live between requests but this is useful for testing in a development environment.

Cache Drivers

The cache drivers follow a simple interface that is defined in `Doctrine\Common\Cache\Cache`. All the cache drivers extend a base class `Doctrine\Common\Cache\AbstractCache` which implements the before mentioned interface.

The interface defines the following methods for you to publicly use.

- fetch($id) - Fetches an entry from the cache.
- contains($id) - Test if an entry exists in the cache.
- save($id, $data, $lifeTime = false) - Puts data into the cache.
- delete($id) - Deletes a cache entry.

Each driver extends the `AbstractCache` class which defines a few abstract protected methods that each of the drivers must implement.

- _doFetch($id)
- _doContains($id)
- _doSave($id, $data, $lifeTime = false)
- _doDelete($id)

The public methods `fetch()`, `contains()`, etc. utilize the above protected methods that are implemented by the drivers. The code is organized this way so that the protected methods in the drivers do the raw interaction with the cache implementation and the `AbstractCache` can build custom functionality on top of these methods.

APC

In order to use the APC cache driver you must have it compiled and enabled in your php.ini. You can read about APC here[10] on the PHP website. It will give you a little background information about what it is and how you can use it as well as how to install it.

Below is a simple example of how you could use the APC cache driver by itself.

10. `http://us2.php.net/apc`

```
<?php

$cacheDriver = new \Doctrine\Common\Cache\ApcCache();
$cacheDriver->save('cache_id', 'my_data');
```

Memcache

In order to use the Memcache cache driver you must have it compiled and enabled in your php.ini. You can read about Memcache here[11] on the PHP website. It will give you a little background information about what it is and how you can use it as well as how to install it.

Below is a simple example of how you could use the Memcache cache driver by itself.

```
<?php

$memcache = new Memcache();
$memcache->connect('memcache_host', 11211);

$cacheDriver = new \Doctrine\Common\Cache\MemcacheCache();
$cacheDriver->setMemcache()
$cacheDriver->save('cache_id', 'my_data');
```

Xcache

In order to use the Xcache cache driver you must have it compiled and enabled in your php.ini. You can read about Xcache here[12]. It will give you a little background information about what it is and how you can use it as well as how to install it.

Below is a simple example of how you could use the Xcache cache driver by itself.

```
<?php

$cacheDriver = new \Doctrine\Common\Cache\XcacheCache();
$cacheDriver->save('cache_id', 'my_data');
```

Using Cache Drivers

In this section we'll describe how you can fully utilize the API of the cache drivers to save cache, check if some cache exists, fetch the cached data and delete the cached data. We'll use the `ArrayCache` implementation as our example here.

```
<?php

$cacheDriver = new \Doctrine\Common\Cache\ArrayCache();
```

Saving

To save some data to the cache driver it is as simple as using the `save()` method.

11. http://us2.php.net/memcache
12. http://xcache.lighttpd.net/

```php
<?php

$cacheDriver->save('cache_id', 'my_data');
```

The `save()` method accepts three arguments which are described below.

- `$id` - The cache id
- `$data` - The cache entry/data.
- `$lifeTime` - The lifetime. If != false, sets a specific lifetime for this cache entry (null => infinite lifeTime).

You can save any type of data whether it be a string, array, object, etc.

```php
<?php

$array = array(
    'key1' => 'value1',
    'key2' => 'value2'
);
$cacheDriver->save('my_array', $array);
```

Checking

Checking whether some cache exists is very simple, just use the `contains()` method. It accepts a single argument which is the ID of the cache entry.

```php
<?php

if ($cacheDriver->contains('cache_id')) {
    echo 'cache exists';
} else {
    echo 'cache does not exist';
}
```

Fetching

Now if you want to retrieve some cache entry you can use the `fetch()` method. It also accepts a single argument just like `contains()` which is the ID of the cache entry.

```php
<?php

$array = $cacheDriver->fetch('my_array');
```

Deleting

As you might guess, deleting is just as easy as saving, checking and fetching. We have a few ways to delete cache entries. You can delete by an individual ID, regular expression, prefix, suffix or you can delete all entries.

By Cache ID

```php
<?php
```

```
$cacheDriver->delete('my_array');
```

You can also pass wild cards to the `delete()` method and it will return an array of IDs that were matched and deleted.

```
<?php

$deleted = $cacheDriver->delete('users_*');
```

By Regular Expression

If you need a little more control than wild cards you can use a PHP regular expression to delete cache entries.

```
<?php

$deleted = $cacheDriver->deleteByRegex('/users_.*/');
```

By Prefix

Because regular expressions are kind of slow, if simply deleting by a prefix or suffix is sufficient, it is recommended that you do that instead of using a regular expression because it will be much faster if you have many cache entries.

```
<?php

$deleted = $cacheDriver->deleteByPrefix('users_');
```

By Suffix

Just like we did above with the prefix you can do the same with a suffix.

```
<?php

$deleted = $cacheDriver->deleteBySuffix('_my_account');
```

All

If you simply want to delete all cache entries you can do so with the `deleteAll()` method.

```
<?php

$deleted = $cacheDriver->deleteAll();
```

Counting

If you want to count how many entries are stored in the cache driver instance you can use the `count()` method.

```
<?php
```

```
echo $cacheDriver->count();
```

In order to use `deleteByRegex()`, `deleteByPrefix()`, `deleteBySuffix()`, `deleteAll()`, `count()` or `getIds()` you must enable an option for the cache driver to manage your cache IDs internally. This is necessary because APC, Memcache, etc. don't have any advanced functionality for fetching and deleting. We add some functionality on top of the cache drivers to maintain an index of all the IDs stored in the cache driver so that we can allow more granular deleting operations.

```
<?php

$cacheDriver->setManageCacheIds(true);
```

Namespaces

If you heavily use caching in your application and utilize it in multiple parts of your application, or use it in different applications on the same server you may have issues with cache naming collisions. This can be worked around by using namespaces. You can set the namespace a cache driver should use by using the `setNamespace()` method.

```
<?php

$cacheDriver->setNamespace('my_namespace_');
```

Integrating with the ORM

The Doctrine ORM package is tightly integrated with the cache drivers to allow you to improve performance of various aspects of Doctrine by just simply making some additional configurations and method calls.

Query Cache

It is highly recommended that in a production environment you cache the transformation of a DQL query to its SQL counterpart. It doesn't make sense to do this parsing multiple times as it doesn't change unless you alter the DQL query.

This can be done by configuring the query cache implementation to use on your ORM configuration.

```
<?php

$config = new \Doctrine\ORM\Configuration();
$config->setQueryCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Result Cache

The result cache can be used to cache the results of your queries so that we don't have to query the database or hydrate the data again after the first time. You just need to configure the result cache implementation.

```php
<?php

$config->setResultCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Now when you're executing DQL queries you can configure them to use the result cache.

```php
<?php

$query = $em->createQuery('select u from \Entities\User u');
$query->useResultCache(true);
```

You can also configure an individual query to use a different result cache driver.

```php
<?php

$query->setResultCacheDriver(new \Doctrine\Common\Cache\ApcCache());
```

Setting the result cache driver on the query will automatically enable the result cache for the query. If you want to disable it pass false to `useResultCache()`.

```php
<?php

$query->useResultCache(false);
```

If you want to set the time the cache has to live you can use the `setResultCacheLifetime()` method.

```php
<?php

$query->setResultCacheLifetime(3600);
```

The ID used to store the result set cache is a hash which is automatically generated for you if you don't set a custom ID yourself with the `setResultCacheId()` method.

```php
<?php

$query->setResultCacheId('my_custom_id');
```

You can also set the lifetime and cache ID by passing the values as the second and third argument to `useResultCache()`.

```php
<?php

$query->useResultCache(true, 3600, 'my_custom_id');
```

Metadata Cache

Your class metadata can be parsed from a few different sources like YAML, XML, Annotations, etc. Instead of parsing this information on each request we should cache it using one of the cache drivers.

Just like the query and result cache we need to configure it first.

```
<?php

$config->setMetadataCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

Now the metadata information will only be parsed once and stored in the cache driver.

Clearing the Cache

We've already shown you previously how you can use the API of the cache drivers to manually delete cache entries. For your convenience we offer a command line task for you to help you with clearing the query, result and metadata cache.

From the Doctrine command line you can run the following command.

Listing 17-1
```
$ ./doctrine clear-cache
```

Running this task with no arguments will clear all the cache for all the configured drivers. If you want to be more specific about what you clear you can use the following options.

To clear the query cache use the `--query` option.

Listing 17-2
```
$ ./doctrine clear-cache --query
```

To clear the metadata cache use the `--metadata` option.

Listing 17-3
```
$ ./doctrine clear-cache --metadata
```

To clear the result cache use the `--result` option.

Listing 17-4
```
$ ./doctrine clear-cache --result
```

When you use the `--result` option you can use some other options to be more specific about what queries result sets you want to clear.

Just like the API of the cache drivers you can clear based on an ID, regular expression, prefix or suffix.

Listing 17-5
```
$ ./doctrine clear-cache --result --id=cache_id
```

Or if you want to clear based on a regular expressions.

Listing 17-6
```
$ ./doctrine clear-cache --result --regex=users_.*
```

Or with a prefix.

Listing 17-7
```
$ ./doctrine clear-cache --result --prefix=users_
```

And finally with a suffix.

```
$ ./doctrine clear-cache --result --suffix=_my_account
```

Listing 17-8

Using the `--id`, `--regex`, etc. options with the `--query` and `--metadata` are not allowed as it is not necessary to be specific about what you clear. You only ever need to completely clear the cache to remove stale entries.

Cache Slams

Something to be careful of when utilizing the cache drivers is cache slams. If you have a heavily trafficked website with some code that checks for the existence of a cache record and if it does not exist it generates the information and saves it to the cache. Now if 100 requests were issued all at the same time and each one sees the cache does not exist and they all try and insert the same cache entry it could lock up APC, Xcache, etc. and cause problems. Ways exist to work around this, like pre-populating your cache and not letting your users requests populate the cache.

You can read more about cache slams here[13].

13. `http://t3.dotgnu.info/blog/php/user-cache-timebomb`

Chapter 18
Improving Performance

Bytecode Cache

It is highly recommended to make use of a bytecode cache like APC. A bytecode cache removes the need for parsing PHP code on every request and can greatly improve performance.

"If you care about performance and don't use a bytecode cache then you don't really care about performance. Please get one and start using it." (Stas Malyshev, Core Contributor to PHP and Zend Employee).

Metadata and Query caches

As already mentioned earlier in the chapter about configuring Doctrine, it is strongly discouraged to use Doctrine without a Metadata and Query cache (preferrably with APC or Memcache as the cache driver). Operating Doctrine without these caches means Doctrine will need to load your mapping information on every single request and has to parse each DQL query on every single request. This is a waste of resources.

Alternative Query Result Formats

Make effective use of the available alternative query result formats like nested array graphs or pure scalar results, especially in scenarios where data is loaded for read-only purposes.

Apply Best Practices

A lot of the points mentioned in the Best Practices chapter will also positively affect the performance of Doctrine.

Chapter 19
Tools

The Doctrine CLI

The Doctrine CLI (Command Line Interface) is a tool for simplifying many common tasks during the development of a project that uses Doctrine.

Installation

If you installed Doctrine 2 through PEAR, the `doctrine` command line tool should already be available to you.

If you use Doctrine through SVN or a release package you need to copy the `doctrine` and `doctrine.php` files from the `tools/sandbox` or `bin` folder, respectively, to a location of your choice, for example a `tools` folder of your project. In addition you may need to edit `doctrine.php` and adjust some paths to the new environment.

Getting Help

Type `doctrine` on the command line and you should see an overview of the available tasks or use the --help flag to get information on the available tasks. If you want to know more about the use of the schema tool for example you can call:

```
doctrine orm:schema-tool --help
```

Listing 19-1

Configuration

Whenever the `doctrine` command line tool is invoked it looks for a file named `cli-config.php` in the current working directory. This config file is a simple php script that usually defines a CLI Configuration object instance. When using ORM package, it is required to define an attribute inside Configuration: `em`. `em` must be an `EntityManager` instance that is used by ORM command-line tasks.

Many tasks of the Doctrine CLI require the `em` attribute to be an `EntityManager` in order to execute. The `EntityManager` instance implicitly defines a database connection. If you invoke a task that requires an EntityManager (and/or a database connection) and the `em` attribute is not defined in your `cli-config.php`, the task invoking will report an error for you.

A `cli-config.php` contains typical Doctrine bootstrap code and predefines the needed attributes mentioned above. A typical `cli-config.php` file looks as follows:

```
<?php
```

```
require_once '/path/to/lib/Doctrine/Common/ClassLoader.php';

$classLoader = new \Doctrine\Common\ClassLoader('MyProject', '/path/to/
myproject/lib');
$classLoader->register();

$ormConfig = new \Doctrine\ORM\Configuration();
$ormConfig->setMetadataCacheImpl(new \Doctrine\Common\Cache\ArrayCache);
$ormConfig->setProxyDir('/path/to/myproject/lib/MyProject/Proxies');
$ormConfig->setProxyNamespace('MyProject\Proxies');

$connectionOptions = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite'
);

$em = \Doctrine\ORM\EntityManager::create($connectionOptions, $config);

$cliConfig = new \Doctrine\Common\Cli\Configuration();
$cliConfig->setAttribute('em', $em);
```

Task Overview

The following tasks are currently available:

- `dbal:run-sql`: Used to run arbitrary SQL on the command-line.
- `orm:convert-mapping`: Used to convert between annotations/xml/yaml mapping informations as well as for class generating from xml/yaml mapping documents or for reverse engineering.
- `orm:generate-proxies`: Used to generate proxy classes used by Doctrine.
- `orm:run-dql`: Used to run arbitrary DQL on the command-line.
- `orm:schema-tool`: Used to forward-engineer the relational database schema from existing classes and mappings.
- `orm:version`: Used to show the current version of the CLI and Doctrine.

Database Schema Generation

To generate your database schema from your Doctrine mapping files you can use the `SchemaTool` class or the `schema-tool` CLI task.

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 alot durving 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 `ClassMetdataInfo` instances.

```
<?php

$tool->updateSchema($classes);
```

If you want to use this functionality from the command line you can use the `schema-tool` task.

To create the schema use the `--create` option:

```
$ php doctrine orm:schema-tool --create
```

Listing 19-2

To drop the scheme use the `--drop` option:

```
$ php doctrine orm:schema-tool --drop
```

Listing 19-3

If you want to drop and then recreate the schema then use both options:

```
$ php doctrine orm:schema-tool --drop --create
```

Listing 19-4

As you would think, if you want to update your schema use the `--update` option:

```
$ php doctrine orm:schema-tool --update
```

Listing 19-5

All of the above tasks also accept a `--dump-sql` option that will output the SQL for the ran operation.

```
$ php doctrine orm:schema-tool --create --dump-sql
```

Listing 19-6

Convert Mapping Information

Doctrine comes with some special tools for working with the various supported formats for specifying mapping information.

You have the ability to convert from a few different sources.

- An existing database

- A directory of YAML schema files
- A directory of XML schema files
- A directory of PHP scripts which populate `ClassMetadataInfo` instances
- A directory of PHP classes defining Doctrine entities with annotations

To convert a mapping source you can do everything you need with the `ClassMetadataExporter`.

```
<?php

$cme = new \Doctrine\ORM\Tools\Export\ClassMetadataExporter();
```

Once you have an instance you can start adding mapping sources to convert.

```
<?php

$cme->addMappingSource('/path/to/yml', 'yml');
$cme->addMappingSource('/path/to/xml', 'xml');
$cme->addMappingSource('/path/to/php', 'php');
$cme->addMappingSource('/path/to/annotations', 'annotation');
```

Now to convert the added mapping sources you can do so by using the exporter drivers.

```
<?php

$metadatas = $cme->getMetadatasForMappingSources();

$exporter = $cme->getExporter('yml', '/path/to/export/yml');
$exporter->setMetadatas($metadatas);
$exporter->export();
```

This functionality functionality is also available from the command line to for example convert some YAML mapping files to XML.

Listing 19-7

```
$ php doctrine orm:convert-mapping --from=/path/to/yml --to=xml --dest=/
path/to/xml
```

Reverse Engineering

You can use the same `ClassMetadataExporter` to reverse engineer a database and generate YAML, XML, etc. from your existing databases.

```
<?php

$sm = $em->getConnection()->getSchemaManager();

$cme->addMappingSource($sm, 'database');
$metadatas = $cme->getMetadatasForMappingSources();

$exporter = $cme->getExporter('yml', '/path/to/export/yml');
$exporter->setMetadatas($metadatas);
$exporter->export();
```

From the command line it is very simple to do something like reverse engineer your existing database to set of YAML mapping files.

Listing 19-8

```
$ php doctrine orm:convert-mapping --from=database --to=yml --dest=/path/
to/yml
```

Chapter 20
DBAL

Database Access Layer Introduction

The Doctrine 2 database layer can be used independently of the object-relational mapping. It offers a leightweight abstraction layer around a PDO like API and allows optional access to lots of convenience functionality aswell as the ability to generate platform independent DQL and DDL statements. You can create a Doctrine Connection by using the `Doctrine\DBAL\DriverManager` class.

```
<?php

$config = new \Doctrine\DBAL\Configuration();
//..

$connectionParams = array(
    'dbname' => 'mydb',
    'user' => 'user',
    'password' => 'secret',
    'host' => 'locahlost',
    'driver' => 'pdo_mysql',
);
$conn = DriverManager::getConnection($connectionParams);
```

The `DriverManager` returns an instance of `Doctrine\DBAL\Connection` which is a wrapper around any configured database driver, for example the PDO Mysql driver in the previous example.

DBAL Architecture

The DBAL is seperated into several different packages that perfectly seperate responsibilities of the different RDBMS layers.

- **Drivers** abstract a PHP specific database API by enforcing two interfaces `\Doctrine\DBAL\Driver\Driver` and `\Doctrine\DBAL\Driver\Statement` which require exactly the same methods as PDO.
- **Platforms** abstract the generation of queries and which database features a platform supports. The `\Doctrine\DBAL\Platforms\AbstractPlatform` defines the common denominator of what a database platform has to publish to the userland, to be fully supportable by Doctrine. This includes the SchemaTool,

Transaction Isolation and many other features. The Database platform for MySQL for example can be used by all 3 mysql extensions, PDO, Mysqli and ext/mysql.
- **Logging** holds the interface and some implementations for debugging of Doctrine SQL query execution during a request.
- **Schema** offers an API for each database platform to execute DDL statements against your platform or retrieve metadata about it. It also holds the Schema Abstraction Layer which is used by the different Schema Management facilities of Doctrine DBAL and ORM.
- **Types** offers an abstraction layer for the converting and generation of types between Databases and PHP.

Data Retrieval and Manipulation

The following methods exist for executing queries against your configured database, three very generic methods and some advanced retrievial methods:

- `prepare($sql)` - Prepare a given sql statement and return the `\Doctrine\DBAL\Driver\Statement` instance.
- `executeUpdate($sql, array $params)` - Executes a prepared statement with the given sql and parameters and returns the affected rows count.
- `execute($sql, array $params)` - Creates a prepared statement for the given sql and passes the parameters to the execute method, then returning the statement.
- `fetchAll($sql, array $params)` - Execute the query and fetch all results into an array.
- `fetchArray($sql, array $params)` - Numeric index retrieval of first result row of the given query.
- `fetchBoth($sql, array $params)` - Both numeric and assoc column name retrieval of the first result row.
- `fetchColumn($sql, array $params, $colnum)` - Retrieve only the given column of the first result row.
- `fetchRow($sql, array $params)` - Retrieve assoc row of the first result row.
- `select($sql, $limit, $offset)` - Modify the given query with a limit clause.

There are also convenience methods for data manipulation queries:

- `delete($tableName, array $identifier)` - Delete all rows of a table matching the given identifier, where keys are column names.
- `insert($tableName, array $data)` - Insert a row into the given table name using the key value pairs of data.
- `update($tableName, array $data, array $identifier)` - Update all rows for the matching key value identifiers with the given data.

By default the Doctrine DBAL does no escaping. Escaping is a very tricky business to do automagically, therefore there is none by default. The ORM internally escapes all your values, because it has lots of metadata available about the current context. When you use the Doctrine DBAL as standalone, you have to take care of this yourself. The following methods help you with it:

- `quote($input, $type=null)` - Quote a value
- `quoteIdentifier($identifier)`- Quote an identifier according to the platform details.

Transactions

Doctrine handles transactions with a PDO like API, having methods for `beginTransaction()`, `commit()` and `rollBack()`. For consistency across different drivers Doctrine also handles the nesting of transactions internally. You can call `beginTransaction()` more than once, and only a matching amount of calls to `commit()` triggers the commit to the database. The Doctrine connectionalso has a method to set the transaction isolation level of the connection as supported by the underlying database.

```php
<?php

class Connection
{
    /**
     * Constant for transaction isolation level READ UNCOMMITTED.
     */
    const TRANSACTION_READ_UNCOMMITTED = 1;

    /**
     * Constant for transaction isolation level READ COMMITTED.
     */
    const TRANSACTION_READ_COMMITTED = 2;

    /**
     * Constant for transaction isolation level REPEATABLE READ.
     */
    const TRANSACTION_REPEATABLE_READ = 3;

    /**
     * Constant for transaction isolation level SERIALIZABLE.
     */
    const TRANSACTION_SERIALIZABLE = 4;
}
```

A transaction with Doctrine DBAL might then look like:

```php
<?php

$conn->setTransactionIsolationLevel(Connection::TRANSACTION_SERIALIZABLE);

try{
    $conn->beginTransaction();
    // do stuff
    $conn->commit();
} catch(\Exception $e) {
    $conn->rollback();
}
```

Schema Representation

Doctrine has a very powerful abstraction of database schemas. It offers an object-oriented representation of a database schema with support for all the details of Tables, Sequences, Indexes and Foreign Keys. These Schema instances generate a representation that is equal for all the supported platforms. Internally this functionality is used by the ORM Schema Tool to offer you create, drop and update database schema methods from your Doctrine ORM Metadata model. Up to very specific functionality of your database system this allows you to generate SQL code that makes your Domain model work.

You will be pleased to hear, that Schema representation is completly decoupled from the Doctrine ORM though, that is you can also use it in any other project to implement database migrations or for SQL schema generation for any metadata model that your application has. You can easily generate a Schema, as a simple example shows:

```php
<?php

$schema = new \Doctrine\DBAL\Schema\Schema();
$myTable = $schema->createTable("my_table");
$myTable->createColumn("id", "integer", array("unsigned" => true));
$myTable->createColumn("username", "string", array("length" => 32));
$myTable->setPrimaryKey(array("id"));
$myTable->addUniqueIndex(array("username"));
$schema->createSequence("my_table_seq");

$myForeign = $schema->createTable("my_foreign");
$myForeign->createColumn("id", "integer");
$myForeign->createColumn("user_id", "integer");
$myForeign->addForeignKeyConstraint($myTable, array("user_id"),
array("id"), array("onUpdate" => "CASCADE"));

$queries = $schema->toSql($myPlatform); // get queries to create this
schema.
$dropSchema = $schema->toDropSql($myPlatform); // get queries to safely
delete this schema.
```

Now if you want to compare this schema with another schema, you can use the `Comparator` class to get instances of `SchemaDiff`, `TableDiff` and `ColumnDiff`, aswell as information about other foreign key, sequence and index changes.

```php
<?php

$comparator = new \Doctrine\DBAL\Schema\Comparator();
$schemaDiff = $comparator->compare($fromSchema, $toSchema);

$queries = $schemaDiff->toSql($myPlatform); // queries to get from one to
another schema.
$saveQueries = $schemaDiff->toSaveSql($myPlatform);
```

The Save Diff mode is a specific mode that prevents the deletion of tables and sequences that might occour when making a diff of your schema. This is often necessary when your target schema is not complete but only describes a subset of your application.

All methods that generate SQL queries for you make much effort to get the order of generation correct, so that no problems will ever occour with missing links of foreign keys.

Platforms

Platforms abstract query generation and specifics of the RDBMS featuresets. In most cases you don't need to interact with this package alot, but there might be certain cases when you are programming database independent where you want to access the platform to generate queries for you.

The platform can be accessed from any `Doctrine\DBAL\Connection` instance by calling the `getDatabasePlatform()` method.

You can use your own platform by specifying the 'platform' key with an instance of your own platform:

```php
<?php

$myPlatform = new MyPlatform();
$options = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite',
    'platform' => $myPlatform
);
```

This way you can optimize your schema or generated SQL code with features that might not be portable for instance, however are required for your special needs.

Schema Manager

A Schema Manager instance helps you with the abstraction of the generation of SQL assets such as Tables, Sequences, Foreign Keys and Indexes. You can use any of the Schema Asset classes `Table`, `Sequence`, `ForeignKeyConstraint` and `Index` for use with the methods of the style `dropAndCreate(AssetName)($asset)`, `drop(AssetName)($asset)` and `create(AssetName)($asset)`.

You also have methods to retrieve instances of those types from the current database you are connected to. These methods are:

- `listDatabases()`
- `listFunctions()`
- `listSequences()`
- `listTableColumns($tableName)`
- `listTableConstraints($tableName)`
- `listTableDetails($tableName)`
- `listTableForeignKeys($tableName)`
- `listTableIndexes($tableName)`
- `listTables()`
- `listTriggers()`
- `listUsers()`

For a complete representation of the current database you can use the `createSchema()` method which returns an instance of Schema, which you can use in conjunction with the SchemaTool or Schema Comparator.

Supporting other Databases

To support a database which is not currently shipped with Doctrine you have to implement the following interfaces and abstract classes:

- `\Doctrine\DBAL\Driver\Driver`
- `\Doctrine\DBAL\Driver\Statement`
- `\Doctrine\DBAL\Platforms\AbstractPlatform`
- `\Doctrine\DBAL\Schema\AbstractSchemaManager`

For an already supported platform but unsupported driver you only need to implement the first two interfaces, since the SQL Generation and Schema Management is already supported by the respective platform and schema instances. You can also make use of several Abstract Unittests in the `\Doctrine\Tests\DBAL` package to check if your platform behaves like all the others which is necessary for SchemaTool support, namely:

- `\Doctrine\Tests\DBAL\Platforms\AbstractPlatformTestCase`
- `\Doctrine\Tests\DBAL\Functional\Schema\AbstractSchemaManagerTestCase`

We would be very happy if any support for new databases would be contributed back to Doctrine to make it an even better product.

Chapter 21

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.

Don't use public properties on entities

It is very important that you don't map public properties on entities, but only protected or private ones. The reason for this is simple, whenever you access a public property of a proxy object that hasn't been initialized yet the return value will be null. Doctrine cannot hook into this process and magically make the entity lazy load.

This can create situations where it is very hard to debug the current failure. We therefore urge you to map only private and protected properties on entities and use getter methods or magic __get() to access them.

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 (PHP6).

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
<?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.