Doctrine 2 ORM Documentation

Release 2

Doctrine Project Team

February 12, 2015

Contents

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

Doctrine DBAL and Doctrine Common both have their own documentation.

Getting Help

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

- There is a *FAQ* with answers to frequent questions.
- The Doctrine Mailing List
- Internet Relay Chat (IRC) in #doctrine on Freenode
- Report a bug on JIRA.
- On Twitter with `#doctrine2`
- On StackOverflow

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

CHAPTER 2

Getting Started

- **Tutorial**: *Getting Started with Doctrine*
- **Setup**: *Installation & Configuration*

CHAPTER 3

Mapping Objects onto a Database

- **Mapping**: *Objects* | *Associations* | *Inheritance*
- **Drivers**: *Docblock Annotations* | *XML* | *YAML* | *PHP*

CHAPTER 4

Working with Objects

- **Basic Reference**: *Entities* | *Associations* | *Events*
- **Query Reference**: *DQL* | *QueryBuilder* | *Native SQL*
- **Internals**: *Internals explained* | *Associations*

CHAPTER 5

Advanced Topics

CHAPTER 6

Tutorials

CHAPTER 7

Changelogs

- `Migration to 2.5`

CHAPTER 8

Cookbook

- **Patterns**: *Aggregate Fields* | *Decorator Pattern* | *Strategy Pattern*
- **DQL Extension Points**: *DQL Custom Walkers* | *DQL User-Defined-Functions*
- **Implementation**: *Array Access* | *Notify ChangeTracking Example* | *Using Wakeup Or Clone* | *Working with DateTime* | *Validation* | *Entities in the Session* | *Keeping your Modules independent*
- **Integration into Frameworks/Libraries** *CodeIgniter*
- **Hidden Gems** *Prefixing Table Name*
- **Custom Datatypes** *MySQL Enums Advanced Field Value Conversion*

CHAPTER 9

Welcome to Doctrine 2 ORM's documentation!

9.1 Tutorials

9.1.1 Getting Started with Doctrine

This guide covers getting started with the Doctrine ORM. After working through the guide you should know:

- How to install and configure Doctrine by connecting it to a database
- Mapping PHP objects to database tables
- Generating a database schema from PHP objects
- Using the `EntityManager` to insert, update, delete and find objects in the database.

Guide Assumptions

This guide is designed for beginners that haven't worked with Doctrine ORM before. There are some prerequesites for the tutorial that have to be installed:

- PHP 5.4 or above
- Composer Package Manager (Install Composer)

The code of this tutorial is available on Github.

Note: This tutorial assumes you work with **Doctrine 2.4** and above. Some of the code will not work with lower versions.

What is Doctrine?

Doctrine 2 is an object-relational mapper (ORM) for PHP 5.4+ that provides transparent persistence for PHP objects. It uses the Data Mapper pattern at the heart, aiming for a complete separation of your domain/business logic from the persistence in a relational database management system.

The benefit of Doctrine for the programmer is the ability to focus on the object-oriented business logic and worry about persistence only as a secondary problem. This doesn't mean persistence is downplayed by Doctrine 2, however it is our belief that there are considerable benefits for object-oriented programming if persistence and entities are kept separated.

What are Entities?

Entities are PHP Objects that can be identified over many requests by a unique identifier or primary key. These classes don't need to extend any abstract base class or interface. An entity class must not be final or contain final methods. Additionally it must not implement **clone** nor **wakeup** or *do so safely*.

An entity contains persistable properties. A persistable property is an instance variable of the entity that is saved into and retrieved from the database by Doctrine's data mapping capabilities.

An Example Model: Bug Tracker

For this Getting Started Guide for Doctrine we will implement the Bug Tracker domain model from the Zend_Db_Table documentation. Reading their documentation we can extract the requirements:

- A Bug has a description, creation date, status, reporter and engineer
- A Bug can occur on different Products (platforms)
- A Product has a name.
- Bug reporters and engineers are both Users of the system.
- A User can create new Bugs.
- The assigned engineer can close a Bug.
- A User can see all his reported or assigned Bugs.
- Bugs can be paginated through a list-view.

Project Setup

Create a new empty folder for this tutorial project, for example `doctrine2-tutorial` and create a new file `composer.json` with the following contents:

```
{
    "require": {
        "doctrine/orm": "2.4.*",
        "symfony/yaml": "2.*"
    },
    "autoload": {
        "psr-0": {"": "src/"}
    }
}
```

Install Doctrine using the Composer Dependency Management tool, by calling:

```
$ composer install
```

This will install the packages Doctrine Common, Doctrine DBAL, Doctrine ORM, Symfony YAML and Symfony Console into the *vendor* directory. The Symfony dependencies are not required by Doctrine but will be used in this tutorial.

Add the following directories:

```
doctrine2-tutorial
|-- config
|   |-- xml
|   `-- yaml
`-- src
```

Obtaining the EntityManager

Doctrine's public interface is the EntityManager, it provides the access point to the complete lifecycle management of your entities and transforms entities from and back to persistence. You have to configure and create it to use your entities with Doctrine 2. I will show the configuration steps and then discuss them step by step:

```php
<?php
// bootstrap.php
use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

require_once "vendor/autoload.php";

// Create a simple "default" Doctrine ORM configuration for Annotations
$isDevMode = true;
$config = Setup::createAnnotationMetadataConfiguration(array(__DIR__."/src"), $isDevMode);
// or if you prefer yaml or XML
//$config = Setup::createXMLMetadataConfiguration(array(__DIR__."/config/xml"), $isDevMode);
//$config = Setup::createYAMLMetadataConfiguration(array(__DIR__."/config/yaml"), $isDevMode);

// database configuration parameters
$conn = array(
    'driver' => 'pdo_sqlite',
    'path' => __DIR__ . '/db.sqlite',
);

// obtaining the entity manager
$entityManager = EntityManager::create($conn, $config);
```

The first require statement sets up the autoloading capabilities of Doctrine using the Composer autoload.

The second block consists of the instantiation of the ORM `Configuration` object using the Setup helper. It assumes a bunch of defaults that you don't have to bother about for now. You can read up on the configuration details in the *reference chapter on configuration*.

The third block shows the configuration options required to connect to a database, in my case a file-based sqlite database. All the configuration options for all the shipped drivers are given in the DBAL Configuration section of the manual.

The last block shows how the `EntityManager` is obtained from a factory method.

Generating the Database Schema

Now that we have defined the Metadata mappings and bootstrapped the EntityManager we want to generate the relational database schema from it. Doctrine has a Command-Line Interface that allows you to access the SchemaTool, a component that generates the required tables to work with the metadata.

For the command-line tool to work a cli-config.php file has to be present in the project root directory, where you will execute the doctrine command. Its a fairly simple file:

```php
<?php
// cli-config.php
require_once "bootstrap.php";

return \Doctrine\ORM\Tools\Console\ConsoleRunner::createHelperSet($entityManager);
```

You can then change into your project directory and call the Doctrine command-line tool:

```
$ cd project/
$ vendor/bin/doctrine orm:schema-tool:create
```

At this point no entitiy metadata exists in *src* so you will see a message like “No Metadata Classes to process.” Don’t worry, we’ll create a Product entity and corresponding metadata in the next section.

You should be aware that during the development process you’ll periodically need to update your database schema to be in sync with your Entities metadata.

You can easily recreate the database:

```
$ vendor/bin/doctrine orm:schema-tool:drop --force
$ vendor/bin/doctrine orm:schema-tool:create
```

Or use the update functionality:

```
$ vendor/bin/doctrine orm:schema-tool:update --force
```

The updating of databases uses a Diff Algorithm for a given Database Schema, a cornerstone of the `Doctrine\DBAL` package, which can even be used without the Doctrine ORM package.

Starting with the Product

We start with the simplest entity, the Product. Create a `src/Product.php` file to contain the `Product` entity definition:

```php
<?php
// src/Product.php
class Product
{
    /**
     * @var int
     */
    protected $id;
    /**
     * @var string
     */
    protected $name;

    public function getId()
    {
        return $this->id;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
        $this->name = $name;
    }
}
```

Note that all fields are set to protected (not public) with a mutator (getter and setter) defined for every field except $id. The use of mutators allows Doctrine to hook into calls which manipulate the entities in ways that it could not if you just directly set the values with `entity#field = foo;`

The id field has no setter since, generally speaking, your code should not set this value since it represents a database id value. (Note that Doctrine itself can still set the value using the Reflection API instead of a defined setter function)

The next step for persistence with Doctrine is to describe the structure of the `Product` entity to Doctrine using a metadata language. The metadata language describes how entities, their properties and references should be persisted and what constraints should be applied to them.

Metadata for entities are configured using a XML, YAML or Docblock Annotations. This Getting Started Guide will show the mappings for all Mapping Drivers. References in the text will be made to the XML mapping.

- *PHP*

  ```php
  <?php
  // src/Product.php
  /**
   * @Entity @Table(name="products")
   **/
  class Product
  {
      /** @Id @Column(type="integer") @GeneratedValue **/
      protected $id;
      /** @Column(type="string") **/
      protected $name;

      // .. (other code)
  }
  ```

- *XML*

  ```xml
  <!-- config/xml/Product.dcm.xml -->
  <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://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

        <entity name="Product" table="products">
            <id name="id" type="integer">
                <generator strategy="AUTO" />
            </id>

            <field name="name" type="string" />
        </entity>
  </doctrine-mapping>
  ```

- *YAML*

  ```yaml
  # config/yaml/Product.dcm.yml
  Product:
    type: entity
    table: products
    id:
      id:
        type: integer
        generator:
          strategy: AUTO
    fields:
      name:
        type: string
  ```

The top-level `entity` definition tag specifies information about the class and table-name. The primitive type `Product#name` is defined as a `field` attribute. The `id` property is defined with the `id` tag, this has a `generator` tag nested inside which defines that the primary key generation mechanism automatically uses the database platforms native id generation strategy (for example AUTO INCREMENT in the case of MySql or Sequences in the case of PostgreSql and Oracle).

Now that we have defined our first entity, lets update the database:

```
$ vendor/bin/doctrine orm:schema-tool:update --force --dump-sql
```

Specifying both flags `--force` and `-dump-sql` prints and executes the DDL statements.

Now create a new script that will insert products into the database:

```
<?php
// create_product.php
require_once "bootstrap.php";

$newProductName = $argv[1];

$product = new Product();
$product->setName($newProductName);

$entityManager->persist($product);
$entityManager->flush();

echo "Created Product with ID " . $product->getId() . "\n";
```

Call this script from the command-line to see how new products are created:

```
$ php create_product.php ORM
$ php create_product.php DBAL
```

What is happening here? Using the `Product` is pretty standard OOP. The interesting bits are the use of the `EntityManager` service. To notify the EntityManager that a new entity should be inserted into the database you have to call `persist()`. To intiate a transaction to actually perform the insertion, You have to explicitly call `flush()` on the `EntityManager`.

This distinction between persist and flush is allows to aggregate all writes (INSERT, UPDATE, DELETE) into one single transaction, which is executed when flush is called. Using this approach the write-performance is significantly better than in a scenario where updates are done for each entity in isolation.

Doctrine follows the UnitOfWork pattern which additionally detects all entities that were fetched and have changed during the request. You don't have to keep track of entities yourself, when Doctrine already knows about them.

As a next step we want to fetch a list of all the Products. Let's create a new script for this:

```
<?php
// list_products.php
require_once "bootstrap.php";

$productRepository = $entityManager->getRepository('Product');
$products = $productRepository->findAll();

foreach ($products as $product) {
    echo sprintf("-%s\n", $product->getName());
}
```

The `EntityManager#getRepository()` method can create a finder object (called a repository) for every entity. It is provided by Doctrine and contains some finder methods such as `findAll()`.

Let's continue with displaying the name of a product based on its ID:

```php
<?php
// show_product.php <id>
require_once "bootstrap.php";

$id = $argv[1];
$product = $entityManager->find('Product', $id);

if ($product === null) {
    echo "No product found.\n";
    exit(1);
}

echo sprintf("-%s\n", $product->getName());
```

Updating a product name demonstrates the functionality UnitOfWork of pattern discussed before. We only need to find a product entity and all changes to its properties are written to the database:

```php
<?php
// update_product.php <id> <new-name>
require_once "bootstrap.php";

$id = $argv[1];
$newName = $argv[2];

$product = $entityManager->find('Product', $id);

if ($product === null) {
    echo "Product $id does not exist.\n";
    exit(1);
}

$product->setName($newName);

$entityManager->flush();
```

After calling this script on one of the existing products, you can verify the product name changed by calling the `show_product.php` script.

Adding Bug and User Entities

We continue with the bug tracker domain, by creating the missing classes `Bug` and `User` and putting them into `src/Bug.php` and `src/User.php` respectively.

```php
<?php
// src/Bug.php
/**
 * @Entity(repositoryClass="BugRepository") @Table(name="bugs")
 */
class Bug
{
    /**
     * @Id @Column(type="integer") @GeneratedValue
     * @var int
     */
    protected $id;
    /**
     * @Column(type="string")
     * @var string
     */
    protected $description;
    /**
     * @Column(type="datetime")
     * @var DateTime
     */
    protected $created;
    /**
     * @Column(type="string")
     * @var string
     */
    protected $status;

    public function getId()
    {
        return $this->id;
    }

    public function getDescription()
    {
        return $this->description;
    }

    public function setDescription($description)
    {
        $this->description = $description;
    }

    public function setCreated(DateTime $created)
    {
        $this->created = $created;
    }

    public function getCreated()
    {
        return $this->created;
    }

    public function setStatus($status)
    {
        $this->status = $status;
    }

    public function getStatus()
    {
        return $this->status;
    }
}

<?php
// src/User.php
/**
 * @Entity @Table(name="users")
 */
class User
{
    /**
```

```
     * @Id @GeneratedValue @Column(type="integer")
     * @var int
     */
    protected $id;
    /**
     * @Column(type="string")
     * @var string
     */
    protected $name;

    public function getId()
    {
        return $this->id;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
        $this->name = $name;
    }
}
```

All of the properties discussed so far are simple string and integer values, for example the id fields of the entities, their names, description, status and change dates. Next we will model the dynamic relationships between the entities by defining the references between entities.

References between objects are foreign keys in the database. You never have to (and never should) work with the foreign keys directly, only with the objects that represent the foreign key through their own identity.

For every foreign key you either have a Doctrine ManyToOne or OneToOne association. On the inverse sides of these foreign keys you can have OneToMany associations. Obviously you can have ManyToMany associations that connect two tables with each other through a join table with two foreign keys.

Now that you know the basics about references in Doctrine, we can extend the domain model to match the requirements:

```
<?php
// src/Bug.php
use Doctrine\Common\Collections\ArrayCollection;

class Bug
{
    // ... (previous code)

    protected $products;

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

```
<?php
// src/User.php
use Doctrine\Common\Collections\ArrayCollection;
```

```php
class User
{
    // ... (previous code)

    protected $reportedBugs;
    protected $assignedBugs;

    public function __construct()
    {
        $this->reportedBugs = new ArrayCollection();
        $this->assignedBugs = new ArrayCollection();
    }
}
```

Whenever an entity is recreated from the database, an Collection implementation of the type Doctrine is injected into your entity instead of an array. Compared to the ArrayCollection this implementation helps the Doctrine ORM understand the changes that have happened to the collection which are noteworthy for persistence.

Warning: Lazy load proxies always contain an instance of Doctrine's EntityManager and all its dependencies. Therefore a var_dump() will possibly dump a very large recursive structure which is impossible to render and read. You have to use `Doctrine\Common\Util\Debug::dump()` to restrict the dumping to a human readable level. Additionally you should be aware that dumping the EntityManager to a Browser may take several minutes, and the Debug::dump() method just ignores any occurrences of it in Proxy instances.

Because we only work with collections for the references we must be careful to implement a bidirectional reference in the domain model. The concept of owning or inverse side of a relation is central to this notion and should always be kept in mind. The following assumptions are made about relations and have to be followed to be able to work with Doctrine 2. These assumptions are not unique to Doctrine 2 but are best practices in handling database relations and Object-Relational Mapping.

- Changes to Collections are saved or updated, when the entity on the *owning* side of the collection is saved or updated.
- Saving an Entity at the inverse side of a relation never triggers a persist operation to changes to the collection.
- In a one-to-one relation the entity holding the foreign key of the related entity on its own database table is *always* the owning side of the relation.
- In a many-to-many relation, both sides can be the owning side of the relation. However in a bi-directional many-to-many relation only one is allowed to be.
- In a many-to-one relation the Many-side is the owning side by default, because it holds the foreign key.
- The OneToMany side of a relation is inverse by default, since the foreign key is saved on the Many side. A OneToMany relation can only be the owning side, if its implemented using a ManyToMany relation with join table and restricting the one side to allow only UNIQUE values per database constraint.

Note: Consistency of bi-directional references on the inverse side of a relation have to be managed in userland application code. Doctrine cannot magically update your collections to be consistent.

In the case of Users and Bugs we have references back and forth to the assigned and reported bugs from a user, making this relation bi-directional. We have to change the code to ensure consistency of the bi-directional reference:

```php
<?php
// src/Bug.php
class Bug
{
    // ... (previous code)
    protected $engineer;
    protected $reporter;

    public function setEngineer($engineer)
    {
        $engineer->assignedToBug($this);
        $this->engineer = $engineer;
    }

    public function setReporter($reporter)
    {
        $reporter->addReportedBug($this);
        $this->reporter = $reporter;
    }

    public function getEngineer()
    {
        return $this->engineer;
    }

    public function getReporter()
    {
        return $this->reporter;
    }
}

<?php
// src/User.php
class User
{
    // ... (previous code)

    private $reportedBugs = null;
    private $assignedBugs = null;

    public function addReportedBug($bug)
    {
        $this->reportedBugs[] = $bug;
    }

    public function assignedToBug($bug)
    {
        $this->assignedBugs[] = $bug;
    }
}
```

I chose to name the inverse methods in past-tense, which should indicate that the actual assigning has already taken place and the methods are only used for ensuring consistency of the references. This approach is my personal preference, you can choose whatever method to make this work.

You can see from `User#addReportedBug()` and `User#assignedToBug()` that using this method in userland alone would not add the Bug to the collection of the owning side in `Bug#reporter` or `Bug#engineer`. Using these methods and calling Doctrine for persistence would not update the collections representation in the database.

Only using `Bug#setEngineer()` or `Bug#setReporter()` correctly saves the relation information.

The `Bug#reporter` and `Bug#engineer` properties are Many-To-One relations, which point to a User. In a normalized relational model the foreign key is saved on the Bug's table, hence in our object-relation model the Bug is at the owning side of the relation. You should always make sure that the use-cases of your domain model should drive which side is an inverse or owning one in your Doctrine mapping. In our example, whenever a new bug is saved or an engineer is assigned to the bug, we don't want to update the User to persist the reference, but the Bug. This is the case with the Bug being at the owning side of the relation.

Bugs reference Products by an uni-directional ManyToMany relation in the database that points from Bugs to Products.

```php
<?php
// src/Bug.php
class Bug
{
    // ... (previous code)

    protected $products = null;

    public function assignToProduct($product)
    {
        $this->products[] = $product;
    }

    public function getProducts()
    {
        return $this->products;
    }
}
```

We are now finished with the domain model given the requirements. Lets add metadata mappings for the `User` and `Bug` as we did for the `Product` before:

- *PHP*

```php
<?php
// src/Bug.php
/**
 * @Entity @Table(name="bugs")
 **/
class Bug
{
    /**
     * @Id @Column(type="integer") @GeneratedValue
     **/
    protected $id;
    /**
     * @Column(type="string")
     **/
    protected $description;
    /**
     * @Column(type="datetime")
     **/
    protected $created;
    /**
     * @Column(type="string")
     **/
    protected $status;

    /**
     * @ManyToOne(targetEntity="User", inversedBy="assignedBugs")
     **/
    protected $engineer;

    /**
```

```
     * @ManyToOne(targetEntity="User", inversedBy="reportedBugs")
     **/
    protected $reporter;

    /**
     * @ManyToMany(targetEntity="Product")
     **/
    protected $products;

    // ... (other code)
}
```

- *XML*

```
<!-- config/xml/Bug.dcm.xml -->
<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://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

    <entity name="Bug" table="bugs">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>

        <field name="description" type="text" />
        <field name="created" type="datetime" />
        <field name="status" type="string" />

        <many-to-one target-entity="User" field="reporter" inversed-by="reportedBugs" />
        <many-to-one target-entity="User" field="engineer" inversed-by="assignedBugs" />

        <many-to-many target-entity="Product" field="products" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
# config/yaml/Bug.dcm.yml
Bug:
  type: entity
  table: bugs
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    description:
      type: text
    created:
      type: datetime
    status:
      type: string
  manyToOne:
    reporter:
      targetEntity: User
      inversedBy: reportedBugs
    engineer:
      targetEntity: User
      inversedBy: assignedBugs
  manyToMany:
    products:
      targetEntity: Product
```

Here we have the entity, id and primitive type definitions. For the "created" field we have used the `datetime` type, which translates the YYYY-mm-dd HH:mm:ss database format into a PHP DateTime instance and back.

After the field definitions the two qualified references to the user entity are defined. They are created by the `many-to-one` tag. The class name of the related entity has to be specified with the `target-entity` attribute, which is enough information for the database mapper to access the foreign-table. Since `reporter` and `engineer` are on the owning side of a bi-directional relation we also have to specify the `inversed-by` attribute. They have to point to the field names on the inverse side of the relationship. We will see in the next example that the `inversed-by` attribute has a counterpart `mapped-by` which makes that the inverse side.

The last definition is for the `Bug#products` collection. It holds all products where the specific bug occurs. Again you have to define the `target-entity` and `field` attributes on the `many-to-many` tag.

The last missing definition is that of the User entity:

- *PHP*

```php
<?php
// src/User.php
/**
 * @Entity @Table(name="users")
 **/
class User
{
    /**
     * @Id @GeneratedValue @Column(type="integer")
     * @var int
     **/
    protected $id;

    /**
     * @Column(type="string")
     * @var string
     **/
    protected $name;

    /**
     * @OneToMany(targetEntity="Bug", mappedBy="reporter")
     * @var Bug[]
     **/
    protected $reportedBugs = null;

    /**
     * @OneToMany(targetEntity="Bug", mappedBy="engineer")
     * @var Bug[]
     **/
    protected $assignedBugs = null;

    // .. (other code)
}
```

- *XML*

```xml
<!-- config/xml/User.dcm.xml -->
<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://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

     <entity name="User" table="users">
         <id name="id" type="integer">
             <generator strategy="AUTO" />
         </id>

         <field name="name" type="string" />

         <one-to-many target-entity="Bug" field="reportedBugs" mapped-by="reporter" />
         <one-to-many target-entity="Bug" field="assignedBugs" mapped-by="engineer" />
     </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
# config/xml/User.dcm.yml
User:
  type: entity
  table: users
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
  oneToMany:
    reportedBugs:
      targetEntity: Bug
      mappedBy: reporter
    assignedBugs:
      targetEntity: Bug
      mappedBy: engineer
```

Here are some new things to mention about the `one-to-many` tags. Remember that we discussed about the inverse and owning side. Now both reportedBugs and assignedBugs are inverse relations, which means the join details have already been defined on the owning side. Therefore we only have to specify the property on the Bug class that holds the owning sides.

This example has a fair overview of the most basic features of the metadata definition language.

Update your database running:

```
$ vendor/bin/doctrine orm:schema-tool:update --force
```

Implementing more Requirements

For starters we need a create user entities:

```php
<?php
// create_user.php
require_once "bootstrap.php";
```

```
$newUsername = $argv[1];

$user = new User();
$user->setName($newUsername);

$entityManager->persist($user);
$entityManager->flush();

echo "Created User with ID " . $user->getId() . "\n";
```

Now call:

```
$ php create_user.php beberlei
```

We now have the data to create a bug and the code for this scenario may look like this:

```
<?php
// create_bug.php
require_once "bootstrap.php";

$theReporterId = $argv[1];
$theDefaultEngineerId = $argv[2];
$productIds = explode(",", $argv[3]);

$reporter = $entityManager->find("User", $theReporterId);
$engineer = $entityManager->find("User", $theDefaultEngineerId);
if (!$reporter || !$engineer) {
    echo "No reporter and/or engineer found for the input.\n";
    exit(1);
}

$bug = new Bug();
$bug->setDescription("Something does not work!");
$bug->setCreated(new DateTime("now"));
$bug->setStatus("OPEN");

foreach ($productIds as $productId) {
    $product = $entityManager->find("Product", $productId);
    $bug->assignToProduct($product);
}

$bug->setReporter($reporter);
$bug->setEngineer($engineer);

$entityManager->persist($bug);
$entityManager->flush();

echo "Your new Bug Id: ".$bug->getId()."\n";
```

Since we only have one user and product, probably with the ID of 1, we can call this script with:

```
php create_bug.php 1 1 1
```

This is the first contact with the read API of the EntityManager, showing that a call to `EntityManager#find($name, $id)` returns a single instance of an entity queried by primary key. Besides this we see the persist + flush pattern again to save the Bug into the database.

See how simple relating Bug, Reporter, Engineer and Products is done by using the discussed methods in the "A first prototype" section. The UnitOfWork will detect this relations when flush is called and relate them in the database appropriately.

Queries for Application Use-Cases

List of Bugs

Using the previous examples we can fill up the database quite a bit, however we now need to discuss how to query the underlying mapper for the required view representations. When opening the application, bugs can be paginated through a list-view, which is the first read-only use-case:

```php
<?php
// list_bugs.php
require_once "bootstrap.php";

$dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ORDER BY b.created DESC";

$query = $entityManager->createQuery($dql);
$query->setMaxResults(30);
$bugs = $query->getResult();

foreach ($bugs as $bug) {
    echo $bug->getDescription()." - ".$bug->getCreated()->format('d.m.Y')."\n";
    echo "    Reported by: ".$bug->getReporter()->getName()."\n";
    echo "    Assigned to: ".$bug->getEngineer()->getName()."\n";
    foreach ($bug->getProducts() as $product) {
        echo "    Platform: ".$product->getName()."\n";
    }
    echo "\n";
}
```

The DQL Query in this example fetches the 30 most recent bugs with their respective engineer and reporter in one single SQL statement. The console output of this script is then:

```
Something does not work! - 02.04.2010
    Reported by: beberlei
    Assigned to: beberlei
    Platform: My Product
```

Note: DQL is not SQL

You may wonder why we start writing SQL at the beginning of this use-case. Don't we use an ORM to get rid of all the endless hand-writing of SQL? Doctrine introduces DQL which is best described as **object-query-language** and is a dialect of OQL and similar to HQL or JPQL. It does not know the concept of columns and tables, but only those of Entity-Class and property. Using the Metadata we defined before it allows for very short distinctive and powerful queries.

An important reason why DQL is favourable to the Query API of most ORMs is its similarity to SQL. The DQL language allows query constructs that most ORMs don't, GROUP BY even with HAVING, Sub-selects, Fetch-Joins of nested classes, mixed results with entities and scalar data such as COUNT() results and much more. Using DQL you should seldom come to the point where you want to throw your ORM into the dumpster, because it doesn't support some the more powerful SQL concepts.

Instead of handwriting DQL you can use the `QueryBuilder` retrieved by calling `$entityManager->createQueryBuilder()`. There are more details about this in the relevant part of the documentation.

As a last resort you can still use Native SQL and a description of the result set to retrieve entities from the database. DQL boils down to a Native SQL statement and a `ResultSetMapping` instance itself. Using Native SQL you could even use stored procedures for data retrieval, or make use of advanced non-portable database queries like PostgreSql's recursive queries.

Array Hydration of the Bug List

In the previous use-case we retrieved the results as their respective object instances. We are not limited to retrieving objects only from Doctrine however. For a simple list view like the previous one we only need read access to our entities and can switch the hydration from objects to simple PHP arrays instead.

Hydration can be an expensive process so only retrieving what you need can yield considerable performance benefits for read-only requests.

Implementing the same list view as before using array hydration we can rewrite our code:

```php
<?php
// list_bugs_array.php
require_once "bootstrap.php";

$dql = "SELECT b, e, r, p FROM Bug b JOIN b.engineer e ".
       "JOIN b.reporter r JOIN b.products p ORDER BY b.created DESC";
$query = $entityManager->createQuery($dql);
$bugs = $query->getArrayResult();

foreach ($bugs as $bug) {
    echo $bug['description'] . " - " . $bug['created']->format('d.m.Y')."\n";
    echo "    Reported by: ".$bug['reporter']['name']."\n";
    echo "    Assigned to: ".$bug['engineer']['name']."\n";
    foreach ($bug['products'] as $product) {
        echo "    Platform: ".$product['name']."\n";
    }
    echo "\n";
}
```

There is one significant difference in the DQL query however, we have to add an additional fetch-join for the products connected to a bug. The resulting SQL query for this single select statement is pretty large, however still more efficient to retrieve compared to hydrating objects.

Find by Primary Key

The next Use-Case is displaying a Bug by primary key. This could be done using DQL as in the previous example with a where clause, however there is a convenience method on the `EntityManager` that handles loading by primary key, which we have already seen in the write scenarios:

```php
<?php
// show_bug.php
require_once "bootstrap.php";

$theBugId = $argv[1];

$bug = $entityManager->find("Bug", (int)$theBugId);

echo "Bug: ".$bug->getDescription()."\n";
echo "Engineer: ".$bug->getEngineer()->getName()."\n";
```

The output of the engineer's name is fetched from the database! What is happening?

Since we only retrieved the bug by primary key both the engineer and reporter are not immediately loaded from the database but are replaced by LazyLoading proxies. These proxies will load behind the scenes, when the first method is called on them.

Sample code of this proxy generated code can be found in the specified Proxy Directory, it looks like:

```
<?php
namespace MyProject\Proxies;

/**
 * THIS CLASS WAS GENERATED BY THE DOCTRINE ORM. DO NOT EDIT THIS FILE.
 **/
class UserProxy extends \User implements \Doctrine\ORM\Proxy\Proxy
{
    // .. lazy load code here

    public function addReportedBug($bug)
    {
        $this->_load();
        return parent::addReportedBug($bug);
    }

    public function assignedToBug($bug)
    {
        $this->_load();
        return parent::assignedToBug($bug);
    }
}
```

See how upon each method call the proxy is lazily loaded from the database?

The call prints:

```
$ php show_bug.php 1
Bug: Something does not work!
Engineer: beberlei
```

Warning: Lazy loading additional data can be very convenient but the additional queries create an overhead. If you know that certain fields will always (or usually) be required by the query then you will get better performance by explicitly retrieving them all in the first query.

Dashboard of the User

For the next use-case we want to retrieve the dashboard view, a list of all open bugs the user reported or was assigned to. This will be achieved using DQL again, this time with some WHERE clauses and usage of bound parameters:

```
<?php
// dashboard.php
require_once "bootstrap.php";

$theUserId = $argv[1];

$dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ".
       "WHERE b.status = 'OPEN' AND (e.id = ?1 OR r.id = ?1) ORDER BY b.created DESC";

$myBugs = $entityManager->createQuery($dql)
                        ->setParameter(1, $theUserId)
                        ->setMaxResults(15)
                        ->getResult();

echo "You have created or assigned to " . count($myBugs) . " open bugs:\n\n";

foreach ($myBugs as $bug) {
    echo $bug->getId() . " - " . $bug->getDescription()."\n";
}
```

Number of Bugs

Until now we only retrieved entities or their array representation. Doctrine also supports the retrieval of non-entities through DQL. These values are called "scalar result values" and may even be aggregate values using COUNT, SUM, MIN, MAX or AVG functions.

We will need this knowledge to retrieve the number of open bugs grouped by product:

```
<?php
// products.php
require_once "bootstrap.php";

$dql = "SELECT p.id, p.name, count(b.id) AS openBugs FROM Bug b ".
       "JOIN b.products p WHERE b.status = 'OPEN' GROUP BY p.id";
$productBugs = $entityManager->createQuery($dql)->getScalarResult();

foreach ($productBugs as $productBug) {
    echo $productBug['name']." has " . $productBug['openBugs'] . " open bugs!\n";
}
```

Updating Entities

There is a single use-case missing from the requirements, Engineers should be able to close a bug. This looks like:

```
<?php
// src/Bug.php

class Bug
{
    public function close()
    {
        $this->status = "CLOSE";
    }
}

<?php
// close_bug.php
require_once "bootstrap.php";

$theBugId = $argv[1];

$bug = $entityManager->find("Bug", (int)$theBugId);
$bug->close();

$entityManager->flush();
```

When retrieving the Bug from the database it is inserted into the IdentityMap inside the UnitOfWork of Doctrine. This means your Bug with exactly this id can only exist once during the whole request no matter how often you call `EntityManager#find()`. It even detects entities that are hydrated using DQL and are already present in the Identity Map.

When flush is called the EntityManager loops over all the entities in the identity map and performs a comparison between the values originally retrieved from the database and those values the entity currently has. If at least one of these properties is different the entity is scheduled for an UPDATE against the database. Only the changed columns are updated, which offers a pretty good performance improvement compared to updating all the properties.

Entity Repositories

For now we have not discussed how to separate the Doctrine query logic from your model. In Doctrine 1 there was the concept of `Doctrine_Table` instances for this separation. The similar concept in Doctrine2 is called Entity Repositories, integrating the repository pattern at the heart of Doctrine.

Every Entity uses a default repository by default and offers a bunch of convenience methods that you can use to query for instances of that Entity. Take for example our Product entity. If we wanted to Query by name, we can use:

```php
<?php
$product = $entityManager->getRepository('Product')
                         ->findOneBy(array('name' => $productName));
```

The method `findOneBy()` takes an array of fields or association keys and the values to match against.

If you want to find all entities matching a condition you can use `findBy()`, for example querying for all closed bugs:

```php
<?php
$bugs = $entityManager->getRepository('Bug')
                      ->findBy(array('status' => 'CLOSED'));

foreach ($bugs as $bug) {
    // do stuff
}
```

Compared to DQL these query methods are falling short of functionality very fast. Doctrine offers you a convenient way to extend the functionalities of the default `EntityRepository` and put all the specialized DQL query logic on it. For this you have to create a subclass of `Doctrine\ORM\EntityRepository`, in our case a `BugRepository` and group all the previously discussed query functionality in it:

```php
<?php
// src/BugRepository.php

use Doctrine\ORM\EntityRepository;

class BugRepository extends EntityRepository
{
    public function getRecentBugs($number = 30)
    {
        $dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ORDER BY b.created DES

        $query = $this->getEntityManager()->createQuery($dql);
        $query->setMaxResults($number);
        return $query->getResult();
    }

    public function getRecentBugsArray($number = 30)
    {
        $dql = "SELECT b, e, r, p FROM Bug b JOIN b.engineer e ".
               "JOIN b.reporter r JOIN b.products p ORDER BY b.created DESC";
        $query = $this->getEntityManager()->createQuery($dql);
        $query->setMaxResults($number);
        return $query->getArrayResult();
    }

    public function getUsersBugs($userId, $number = 15)
    {
        $dql = "SELECT b, e, r FROM Bug b JOIN b.engineer e JOIN b.reporter r ".
               "WHERE b.status = 'OPEN' AND e.id = ?1 OR r.id = ?1 ORDER BY b.created DESC";

        return $this->getEntityManager()->createQuery($dql)
                             ->setParameter(1, $userId)
                             ->setMaxResults($number)
                             ->getResult();
    }

    public function getOpenBugsByProduct()
    {
        $dql = "SELECT p.id, p.name, count(b.id) AS openBugs FROM Bug b ".
               "JOIN b.products p WHERE b.status = 'OPEN' GROUP BY p.id";
        return $this->getEntityManager()->createQuery($dql)->getScalarResult();
    }
}
```

To be able to use this query logic through `$this->getEntityManager()->getRepository('Bug')` we have to adjust the metadata slightly.

- *PHP*

```
<?php
/**
 * @Entity(repositoryClass="BugRepository")
 * @Table(name="bugs")
 **/
class Bug
{
    //...
}
```

- *XML*

```
<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://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

      <entity name="Bug" table="bugs" repository-class="BugRepository">

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

- *YAML*

```
Bug:
  type: entity
  repositoryClass: BugRepository
```

Now we can remove our query logic in all the places and instead use them through the EntityRepository. As an example here is the code of the first use case "List of Bugs":

```
<?php
// list_bugs_repository.php
require_once "bootstrap.php";

$bugs = $entityManager->getRepository('Bug')->getRecentBugs();

foreach ($bugs as $bug) {
    echo $bug->getDescription()." - ".$bug->getCreated()->format('d.m.Y')."\n";
    echo "    Reported by: ".$bug->getReporter()->getName()."\n";
    echo "    Assigned to: ".$bug->getEngineer()->getName()."\n";
    foreach ($bug->getProducts() as $product) {
        echo "    Platform: ".$product->getName()."\n";
    }
    echo "\n";
}
```

Using EntityRepositories you can avoid coupling your model with specific query logic. You can also re-use query logic easily throughout your application.

Conclusion

This tutorial is over here, I hope you had fun. Additional content will be added to this tutorial incrementally, topics will include:

- More on Association Mappings
- Lifecycle Events triggered in the UnitOfWork
- Ordering of Collections

Additional details on all the topics discussed here can be found in the respective manual chapters.

9.1.2 Getting Started: Database First

Note: *Development Workflows*

When you *Code First*, you start with developing Objects and then map them onto your database. When you *Model First*, you are modelling your application using tools (for example UML) and generate database schema and PHP code from this model. When you have a *Database First*, you already have a database schema and generate the corresponding PHP code from it.

Note: This getting started guide is in development.

Development of new applications often starts with an existing database schema. When the database schema is the starting point for your application, then development is said to use the *Database First* approach to Doctrine.

In this workflow you would modify the database schema first and then regenerate the PHP code to use with this schema. You need a flexible code-generator for this task and up to Doctrine 2.2, the code generator hasn't been flexible enough to achieve this.

We spinned off a subproject, Doctrine CodeGenerator, that will fill this gap and allow you to do *Database First* development.

9.1.3 Getting Started: Model First

Note: *Development Workflows*

When you *Code First*, you start with developing Objects and then map them onto your database. When you *Model First*, you are modelling your application using tools (for example UML) and generate database schema and PHP code from this model. When you have a *Database First*, then you already have a database schema and generate the corresponding PHP code from it.

Note: This getting started guide is in development.

There are applications when you start with a high-level description of the model using modelling tools such as UML. Modelling tools could also be Excel, XML or CSV files that describe the model in some structured way. If your application is using a modelling tool, then the development workflow is said to be a *Model First* approach to Doctrine2.

In this workflow you always change the model description and then regenerate both PHP code and database schema from this model.

9.1.4 Working with Indexed Associations

Note: This feature is scheduled for version 2.1 of Doctrine and not included in the 2.0.x series.

Doctrine 2 collections are modelled after PHPs native arrays. PHP arrays are an ordered hashmap, but in the first version of Doctrine keys retrieved from the database were always numerical unless `INDEX BY` was used. Starting with Doctrine 2.1 you can index your collections by a value in the related entity. This is a first step towards full ordered hashmap support through the Doctrine ORM. The feature works like an implicit `INDEX BY` for the selected association but has several downsides also:

- You have to manage both the key and field if you want to change the index by field value.
- On each request the keys are regenerated from the field value not from the previous collection key.
- Values of the Index-By keys are never considered during persistence, it only exists for accessing purposes.
- Fields that are used for the index by feature **HAVE** to be unique in the database. The behavior for multiple entities with the same index-by field value is undefined.

As an example we will design a simple stock exchange list view. The domain consists of the entity `Stock` and `Market` where each Stock has a symbol and is traded on a single market. Instead of having a numerical list of stocks traded on a market they will be indexed by their symbol, which is unique across all markets.

Mapping Indexed Associations

You can map indexed associations by adding:

- `indexBy` attribute to any `@OneToMany` or `@ManyToMany` annotation.
- `index-by` attribute to any `<one-to-many />` or `<many-to-many />` xml element.
- `indexBy:` key-value pair to any association defined in `manyToMany:` or `oneToMany:` YAML mapping files.

The code and mappings for the Market entity looks like this:

- *PHP*

```php
<?php
namespace Doctrine\Tests\Models\StockExchange;

use Doctrine\Common\Collections\ArrayCollection;

/**
 * @Entity
 * @Table(name="exchange_markets")
 */
class Market
{
    /**
     * @Id @Column(type="integer") @GeneratedValue
     * @var int
     */
    private $id;

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

    /**
     * @OneToMany(targetEntity="Stock", mappedBy="market", indexBy="symbol")
     * @var Stock[]
     */
    private $stocks;

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

    public function getId()
    {
        return $this->id;
    }

    public function getName()
    {
        return $this->name;
    }

    public function addStock(Stock $stock)
    {
        $this->stocks[$stock->getSymbol()] = $stock;
    }

    public function getStock($symbol)
    {
        if (!isset($this->stocks[$symbol])) {
            throw new \InvalidArgumentException("Symbol is not traded on this market.");
        }

        return $this->stocks[$symbol];
    }
    public function getStocks()
    {
        return $this->stocks->toArray();
    }
}
```

- *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
                          http://www.doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Doctrine\Tests\Models\StockExchange\Market">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>

        <field name="name" type="string"/>

        <one-to-many target-entity="Stock" mapped-by="market" field="stocks" index-by="symbol" /
    </entity>
</doctrine-mapping>
```

- *YAML*

```
Doctrine\Tests\Models\StockExchange\Market:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type:string
  oneToMany:
    stocks:
      targetEntity: Stock
      mappedBy: market
      indexBy: symbol
```

Inside the `addStock()` method you can see how we directly set the key of the association to the symbol, so that we can work with the indexed association directly after invoking `addStock()`. Inside `getStock($symbol)` we pick a stock traded on the particular market by symbol. If this stock doesn't exist an exception is thrown.

The `Stock` entity doesn't contain any special instructions that are new, but for completeness here are the code and mappings for it:

- *PHP*

```
<?php
namespace Doctrine\Tests\Models\StockExchange;

/**
 * @Entity
 * @Table(name="exchange_stocks")
 */
```

```php
class Stock
{
    /**
     * @Id @GeneratedValue @Column(type="integer")
     * @var int
     */
    private $id;

    /**
     * For real this column would have to be unique=true. But I want to test behavior of non-uni
     *
     * @Column(type="string", unique=true)
     */
    private $symbol;

    /**
     * @ManyToOne(targetEntity="Market", inversedBy="stocks")
     * @var Market
     */
    private $market;

    public function __construct($symbol, Market $market)
    {
        $this->symbol = $symbol;
        $this->market = $market;
        $market->addStock($this);
    }

    public function getSymbol()
    {
        return $this->symbol;
    }
}
```

- *XML*

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

    <entity name="Doctrine\Tests\Models\StockExchange\Stock">
        <id name="id" type="integer">
            <generator strategy="AUTO" />
        </id>

        <field name="symbol" type="string" unique="true" />
        <many-to-one target-entity="Market" field="market" inversed-by="stocks" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Doctrine\Tests\Models\StockExchange\Stock:
  type: entity
  id:
    id:
      type: integer
```

```
          generator:
            strategy: AUTO
      fields:
        symbol:
          type: string
      manyToOne:
        market:
          targetEntity: Market
          inversedBy: stocks
```

Querying indexed associations

Now that we defined the stocks collection to be indexed by symbol we can take a look at some code, that makes use of the indexing.

First we will populate our database with two example stocks traded on a single market:

```php
<?php
// $em is the EntityManager

$market = new Market("Some Exchange");
$stock1 = new Stock("AAPL", $market);
$stock2 = new Stock("GOOG", $market);

$em->persist($market);
$em->persist($stock1);
$em->persist($stock2);
$em->flush();
```

This code is not particular interesting since the indexing feature is not yet used. In a new request we could now query for the market:

```php
<?php
// $em is the EntityManager
$marketId = 1;
$symbol = "AAPL";

$market = $em->find("Doctrine\Tests\Models\StockExchange\Market", $marketId);

// Access the stocks by symbol now:
$stock = $market->getStock($symbol);

echo $stock->getSymbol(); // will print "AAPL"
```

The implementation `Market::addStock()` in combination with `indexBy` allows to access the collection consistently by the Stock symbol. It does not matter if Stock is managed by Doctrine or not.

The same applies to DQL queries: The `indexBy` configuration acts as implicit "INDEX BY" to a join association.

```php
<?php
// $em is the EntityManager
$marketId = 1;
$symbol = "AAPL";

$dql = "SELECT m, s FROM Doctrine\Tests\Models\StockExchange\Market m JOIN m.stocks s WHERE m.id = ?
$market = $em->createQuery($dql)
             ->setParameter(1, $marketId)
             ->getSingleResult();
```

```php
// Access the stocks by symbol now:
$stock = $market->getStock($symbol);

echo $stock->getSymbol(); // will print "AAPL"
```

If you want to use `INDEX BY` explicitly on an indexed association you are free to do so. Additionally indexed associations also work with the `Collection::slice()` functionality, no matter if marked as LAZY or EXTRA_LAZY.

Outlook into the Future

For the inverse side of a many-to-many associations there will be a way to persist the keys and the order as a third and fourth parameter into the join table. This feature is discussed in DDC-213 This feature cannot be implemented for One-To-Many associations, because they are never the owning side.

9.1.5 Extra Lazy Associations

New in version 2.1.

In many cases associations between entities can get pretty large. Even in a simple scenario like a blog. where posts can be commented, you always have to assume that a post draws hundreds of comments. In Doctrine 2.0 if you accessed an association it would always get loaded completely into memory. This can lead to pretty serious performance problems, if your associations contain several hundreds or thousands of entities.

With Doctrine 2.1 a feature called **Extra Lazy** is introduced for associations. Associations are marked as **Lazy** by default, which means the whole collection object for an association is populated the first time its accessed. If you mark an association as extra lazy the following methods on collections can be called without triggering a full load of the collection:

- `Collection#contains($entity)`
- `Collection#containsKey($key)` (available with Doctrine 2.5)
- `Collection#count()`
- `Collection#get($key)` (available with Doctrine 2.4)
- `Collection#slice($offset, $length = null)`

For each of the above methods the following semantics apply:

- For each call, if the Collection is not yet loaded, issue a straight SELECT statement against the database.
- For each call, if the collection is already loaded, fallback to the default functionality for lazy collections. No additional SELECT statements are executed.

Additionally even with Doctrine 2.0 the following methods do not trigger the collection load:

- `Collection#add($entity)`
- `Collection#offsetSet($key, $entity)` - ArrayAccess with no specific key `$coll[] = $entity`, it does not work when setting specific keys like `$coll[0] = $entity`.

With extra lazy collections you can now not only add entities to large collections but also paginate them easily using a combination of `count` and `slice`.

Enabling Extra-Lazy Associations

The mapping configuration is simple. Instead of using the default value of `fetch="LAZY"` you have to switch to extra lazy as shown in these examples:

- *PHP*

```php
<?php
namespace Doctrine\Tests\Models\CMS;

/**
 * @Entity
 */
class CmsGroup
{
    /**
     * @ManyToMany(targetEntity="CmsUser", mappedBy="groups", fetch="EXTRA_LAZY")
     */
    public $users;
}
```

- *XML*

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

    <entity name="Doctrine\Tests\Models\CMS\CmsGroup">
        <!-- ... -->
        <many-to-many field="users" target-entity="CmsUser" mapped-by="groups" fetch="EXTRA_LAZY
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Doctrine\Tests\Models\CMS\CmsGroup:
  type: entity
  # ...
  manyToMany:
    users:
      targetEntity: CmsUser
      mappedBy: groups
      fetch: EXTRA_LAZY
```

9.1.6 Composite and Foreign Keys as Primary Key

New in version 2.1.

Doctrine 2 supports composite primary keys natively. Composite keys are a very powerful relational database concept and we took good care to make sure Doctrine 2 supports as many of the composite primary key use-cases. For Doctrine 2.0 composite keys of primitive data-types are supported, for Doctrine 2.1 even foreign keys as primary keys are supported.

This tutorial shows how the semantics of composite primary keys work and how they map to the database.

General Considerations

Every entity with a composite key cannot use an id generator other than “ASSIGNED”. That means the ID fields have to have their values set before you call `EntityManager#persist($entity)`.

Primitive Types only

Even in version 2.0 you can have composite keys as long as they only consist of the primitive types `integer` and `string`. Suppose you want to create a database of cars and use the model-name and year of production as primary keys:

- *PHP*

```php
<?php
namespace VehicleCatalogue\Model;

/**
 * @Entity
 */
class Car
{
    /** @Id @Column(type="string") */
    private $name;
    /** @Id @Column(type="integer") */
    private $year

    public function __construct($name, $year)
    {
        $this->name = $name;
        $this->year = $year;
    }

    public function getModelName()
    {
        return $this->name;
    }

    public function getYearOfProduction()
    {
        return $this->year;
    }
}
```

- *XML*

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

    <entity name="VehicleCatalogue\Model\Car">
        <id field="name" type="string" />
        <id field="year" type="integer" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```
VehicleCatalogue\Model\Car:
  type: entity
  id:
    name:
      type: string
    year:
      type: integer
```

Now you can use this entity:

```
<?php
namespace VehicleCatalogue\Model;

// $em is the EntityManager

$car = new Car("Audi A8", 2010);
$em->persist($car);
$em->flush();
```

And for querying you can use arrays to both DQL and EntityRepositories:

```
<?php
namespace VehicleCatalogue\Model;

// $em is the EntityManager
$audi = $em->find("VehicleCatalogue\Model\Car", array("name" => "Audi A8", "year" => 2010));

$dql = "SELECT c FROM VehicleCatalogue\Model\Car c WHERE c.id = ?1";
$audi = $em->createQuery($dql)
           ->setParameter(1, array("name" => "Audi A8", "year" => 2010))
           ->getSingleResult();
```

You can also use this entity in associations. Doctrine will then generate two foreign keys one for `name` and to `year` to the related entities.

Note: This example shows how you can nicely solve the requirement for existing values before `EntityManager#persist()`: By adding them as mandatory values for the constructor.

Identity through foreign Entities

Note: Identity through foreign entities is only supported with Doctrine 2.1

There are tons of use-cases where the identity of an Entity should be determined by the entity of one or many parent entities.

- Dynamic Attributes of an Entity (for example Article). Each Article has many attributes with primary key "article_id" and "attribute_name".
- Address object of a Person, the primary key of the address is "user_id". This is not a case of a composite primary key, but the identity is derived through a foreign entity and a foreign key.
- Join Tables with metadata can be modelled as Entity, for example connections between two articles with a little description and a score.

The semantics of mapping identity through foreign entities are easy:

- Only allowed on Many-To-One or One-To-One associations.

- Plug an `@Id` annotation onto every association.
- Set an attribute `association-key` with the field name of the association in XML.
- Set a key `associationKey:` with the field name of the association in YAML.

Use-Case 1: Dynamic Attributes

We keep up the example of an Article with arbitrary attributes, the mapping looks like this:

- *PHP*

```php
<?php
namespace Application\Model;

use Doctrine\Common\Collections\ArrayCollection;

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

    /**
     * @OneToMany(targetEntity="ArticleAttribute", mappedBy="article", cascade={"ALL"}, indexBy=
     */
    private $attributes;

    public function addAttribute($name, $value)
    {
        $this->attributes[$name] = new ArticleAttribute($name, $value, $this);
    }
}

/**
 * @Entity
 */
class ArticleAttribute
{
    /** @Id @ManyToOne(targetEntity="Article", inversedBy="attributes") */
    private $article;

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

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

    public function __construct($name, $value, $article)
    {
        $this->attribute = $name;
        $this->value = $value;
        $this->article = $article;
    }
}
```

- *XML*

```xml
<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">

     <entity name="Application\Model\ArticleAttribute">
        <id name="article" association-key="true" />
        <id name="attribute" type="string" />

        <field name="value" type="string" />

        <many-to-one field="article" target-entity="Article" inversed-by="attributes" />
     <entity>

</doctrine-mapping>
```

- *YAML*

```yaml
Application\Model\ArticleAttribute:
  type: entity
  id:
    article:
      associationKey: true
    attribute:
      type: string
  fields:
    value:
      type: string
  manyToOne:
    article:
      targetEntity: Article
      inversedBy: attributes
```

Use-Case 2: Simple Derived Identity

Sometimes you have the requirement that two objects are related by a One-To-One association and that the dependent class should re-use the primary key of the class it depends on. One good example for this is a user-address relationship:

- *PHP*

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

/**
 * @Entity
 */
class Address
{
    /** @Id @OneToOne(targetEntity="User") */
```

```
        private $user;
    }
```

- *YAML*

```
User:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: AUTO

Address:
  type: entity
  id:
    user:
      associationKey: true
  oneToOne:
    user:
      targetEntity: User
```

Use-Case 3: Join-Table with Metadata

In the classic order product shop example there is the concept of the order item which contains references to order and product and additional data such as the amount of products purchased and maybe even the current price.

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

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

    /** @ManyToOne(targetEntity="Customer") */
    private $customer;
    /** @OneToMany(targetEntity="OrderItem", mappedBy="order") */
    private $items;

    /** @Column(type="boolean") */
    private $payed = false;
    /** @Column(type="boolean") */
    private $shipped = false;
    /** @Column(type="datetime") */
    private $created;

    public function __construct(Customer $customer)
    {
        $this->customer = $customer;
        $this->items = new ArrayCollection();
        $this->created = new \DateTime("now");
    }
}

/** @Entity */
```

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

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

    /** @Column(type="decimal") */
    private $currentPrice;

    public function getCurrentPrice()
    {
        return $this->currentPrice;
    }
}

/** @Entity */
class OrderItem
{
    /** @Id @ManyToOne(targetEntity="Order") */
    private $order;

    /** @Id @ManyToOne(targetEntity="Product") */
    private $product;

    /** @Column(type="integer") */
    private $amount = 1;

    /** @Column(type="decimal") */
    private $offeredPrice;

    public function __construct(Order $order, Product $product, $amount = 1)
    {
        $this->order = $order;
        $this->product = $product;
        $this->offeredPrice = $product->getCurrentPrice();
    }
}
```

Performance Considerations

Using composite keys always comes with a performance hit compared to using entities with a simple surrogate key. This performance impact is mostly due to additional PHP code that is necessary to handle this kind of keys, most notably when using derived identifiers.

On the SQL side there is not much overhead as no additional or unexpected queries have to be executed to manage entities with derived foreign keys.

9.1.7 Ordering To-Many Associations

There are use-cases when you'll want to sort collections when they are retrieved from the database. In userland you do this as long as you haven't initially saved an entity with its associations into the database. To retrieve a sorted collection from the database you can use the `@OrderBy` annotation with an collection that specifies an DQL snippet that is appended to all queries with this collection.

Additional to any `@OneToMany` or `@ManyToMany` annotation you can specify the `@OrderBy` in the following way:

- *PHP*

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

      /**
       * @ManyToMany(targetEntity="Group")
       * @OrderBy({"name" = "ASC"})
       **/
      private $groups;
  }
  ```

- *XML*

  ```
  <doctrine-mapping>
      <entity name="User">
          <many-to-many field="groups" target-entity="Group">
              <order-by>
                  <order-by-field name="name" direction="ASC" />
              </order-by>
          </many-to-many>
      </entity>
  </doctrine-mapping>
  ```

- *YAML*

  ```
  User:
    type: entity
    manyToMany:
      groups:
        orderBy: { 'name': 'ASC' }
        targetEntity: Group
        joinTable:
          name: users_groups
          joinColumns:
            user_id:
              referencedColumnName: id
          inverseJoinColumns:
            group_id:
              referencedColumnName: id
  ```

The DQL Snippet in OrderBy is only allowed to consist of unqualified, unquoted field names and of an optional ASC/DESC positional statement. Multiple Fields are separated by a comma (,). The referenced field names have to exist on the `targetEntity` class of the `@ManyToMany` or `@OneToMany` annotation.

The semantics of this feature can be described as follows.

- `@OrderBy` acts as an implicit ORDER BY clause for the given fields, that is appended to all the explicitly given ORDER BY items.
- All collections of the ordered type are always retrieved in an ordered fashion.
- To keep the database impact low, these implicit ORDER BY items are only added to an DQL Query if the collection is fetch joined in the DQL query.

Given our previously defined example, the following would not add ORDER BY, since g is not fetch joined:

```
SELECT u FROM User u JOIN u.groups g WHERE SIZE(g) > 10
```

However the following:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10
```

...would internally be rewritten to:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10 ORDER BY g.name ASC
```

You can reverse the order with an explicit DQL ORDER BY:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10 ORDER BY g.name DESC
```

...is internally rewritten to:

```
SELECT u, g FROM User u JOIN u.groups g WHERE u.id = 10 ORDER BY g.name DESC, g.name ASC
```

9.1.8 Override Field Association Mappings In Subclasses

Sometimes there is a need to persist entities but override all or part of the mapping metadata. Sometimes also the mapping to override comes from entities using traits where the traits have mapping metadata. This tutorial explains how to override mapping metadata, i.e. attributes and associations metadata in particular. The example here shows the overriding of a class that uses a trait but is similar when extending a base class as shown at the end of this tutorial.

Suppose we have a class ExampleEntityWithOverride. This class uses trait ExampleTrait:

```php
<?php
/**
 * @Entity
 *
 * @AttributeOverrides({
 *      @AttributeOverride(name="foo",
 *          column=@Column(
 *              name     = "foo_overridden",
 *              type     = "integer",
 *              length   = 140,
 *              nullable = false,
 *              unique   = false
 *          )
 *      )
 * })
 *
 * @AssociationOverrides({
 *      @AssociationOverride(name="bar",
 *          joinColumns=@JoinColumn(
 *              name="example_entity_overridden_bar_id", referencedColumnName="id"
 *          )
 *      )
 * })
 */
class ExampleEntityWithOverride
{
    use ExampleTrait;
}

/**
 * @Entity
 */
class Bar
{
    /** @Id @Column(type="string") */
    private $id;
}
```

The docblock is showing metadata override of the attribute and association type. It basically changes the names of the columns mapped for a property `foo` and for the association `bar` which relates to Bar class shown above. Here is the trait which has mapping metadata that is overridden by the annotation above:

```
/**
 * Trait class
 */
trait ExampleTrait
{
    /** @Id @Column(type="string") */
    private $id;

    /**
     * @Column(name="trait_foo", type="integer", length=100, nullable=true, unique=true)
     */
    protected $foo;

    /**
     * @OneToOne(targetEntity="Bar", cascade={"persist", "merge"})
     * @JoinColumn(name="example_trait_bar_id", referencedColumnName="id")
     */
    protected $bar;
}
```

The case for just extending a class would be just the same but:

```
class ExampleEntityWithOverride extends BaseEntityWithSomeMapping
{
    // ...
}
```

Overriding is also supported via XML and YAML (*examples*).

9.1.9 Pagination

New in version 2.2.

Starting with version 2.2 Doctrine ships with a Paginator for DQL queries. It has a very simple API and implements the SPL interfaces `Countable` and `IteratorAggregate`.

```
<?php
use Doctrine\ORM\Tools\Pagination\Paginator;

$dql = "SELECT p, c FROM BlogPost p JOIN p.comments c";
$query = $entityManager->createQuery($dql)
                       ->setFirstResult(0)
                       ->setMaxResults(100);

$paginator = new Paginator($query, $fetchJoinCollection = true);

$c = count($paginator);
```

```php
foreach ($paginator as $post) {
    echo $post->getHeadline() . "\n";
}
```

Paginating Doctrine queries is not as simple as you might think in the beginning. If you have complex fetch-join scenarios with one-to-many or many-to-many associations using the "default" LIMIT functionality of database vendors is not sufficient to get the correct results.

By default the pagination extension does the following steps to compute the correct result:

- Perform a Count query using *DISTINCT* keyword.
- Perform a Limit Subquery with *DISTINCT* to find all ids of the entity in from on the current page.
- Perform a WHERE IN query to get all results for the current page.

This behavior is only necessary if you actually fetch join a to-many collection. You can disable this behavior by setting the `$fetchJoinCollection` flag to `false`; in that case only 2 instead of the 3 queries described are executed. We hope to automate the detection for this in the future.

9.1.10 Separating Concerns using Embeddables

Embeddables are classes which are not entities themself, but are embedded in entities and can also be queried in DQL. You'll mostly want to use them to reduce duplication or separating concerns.

For the purposes of this tutorial, we will assume that you have a `User` class in your application and you would like to store an address in the `User` class. We will model the `Address` class as an embeddable instead of simply adding the respective columns to the `User` class.

- *PHP*

```php
<?php

/** @Entity */
class User
{
    /** @Embedded(class = "Address") */
    private $address;
}

/** @Embeddable */
class Address
{
    /** @Column(type = "string") */
    private $street;

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

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

    /** @Column(type = "string") */
    private $country;
}
```

- *XML*

```xml
<doctrine-mapping>
    <entity name="User">
        <embedded name="address" class="Address" />
    </entity>

    <embeddable name="Address">
        <field name="street" type="string" />
        <field name="postalCode" type="string" />
        <field name="city" type="string" />
        <field name="country" type="string" />
    </embeddable>
</doctrine-mapping>
```

- *YAML*

```yaml
User:
  type: entity
  embedded:
    address:
      class: Address

Address:
  type: embeddable
  fields:
    street: { type: string }
    postalCode: { type: string }
    city: { type: string }
    country: { type: string }
```

In terms of your database schema, Doctrine will automatically inline all columns from the `Address` class into the table of the `User` class, just as if you had declared them directly there.

9.1.11 Column Prefixing

By default, Doctrine names your columns by prefixing them, using the value object name.

Following the example above, your columns would be named as `address_street`, `address_postalCode`...

You can change this behaviour to meet your needs by changing the `columnPrefix` attribute in the `@Embeddable` notation.

The following example shows you how to set your prefix to `myPrefix_`:

- *PHP*

```php
<?php

/** @Entity */
class User
{
    /** @Embedded(class = "Address", columnPrefix = "myPrefix_") */
    private $address;
}
```

- *XML*

```xml
<entity name="User">
    <embedded name="address" class="Address" column-prefix="myPrefix_" />
</entity>
```

- *YAML*

  ```
  User:
    type: entity
    embedded:
      address:
        class: Address
        columnPrefix: myPrefix_
  ```

To have Doctrine drop the prefix and use the value object's property name directly, set `columnPrefix=false` (`use-column-prefix="false"` for XML):

- *PHP*

  ```php
  <?php

  /** @Entity */
  class User
  {
      /** @Embedded(class = "Address", columnPrefix = false) */
      private $address;
  }
  ```

- *YAML*

  ```
  User:
    type: entity
    embedded:
      address:
        class: Address
        columnPrefix: false
  ```

- *XML*

  ```xml
  <entity name="User">
      <embedded name="address" class="Address" use-column-prefix="false" />
  </entity>
  ```

9.1.12 DQL

You can also use mapped fields of embedded classes in DQL queries, just as if they were declared in the `User` class:

```sql
SELECT u FROM User u WHERE u.address.city = :myCity
```

9.2 Reference Guide

9.2.1 Architecture

This chapter gives an overview of the overall architecture, terminology and constraints of Doctrine 2. It is recommended to read this chapter carefully.

Using an Object-Relational Mapper

As the term ORM already hints at, Doctrine 2 aims to simplify the translation between database rows and the PHP object model. The primary use case for Doctrine are therefore applications that utilize the Object-Oriented Programming

Paradigm. For applications that do not primarily work with objects Doctrine 2 is not suited very well.

Requirements

Doctrine 2 requires a minimum of PHP 5.4. 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`.

Terminology

Entities

An entity is a lightweight, persistent domain object. An entity can be any regular PHP class observing the following restrictions:

- An entity class must not be final or contain final methods.

- All persistent properties/field of any entity class should always be private or protected, otherwise lazy-loading might not work as expected. In case you serialize entities (for example Session) properties should be protected (See Serialize section below).
- An entity class must not implement `__clone` or *do so safely*.
- An entity class must not implement `__wakeup` or *do so safely*. Also consider implementing Serializable instead.
- 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. That is, if B inherits from A then B must not have a mapped field with the same name as an already mapped field that is inherited from A.
- An entity cannot make use of func_get_args() to implement variable parameters. Generated proxies do not support this for performance reasons and your code might actually fail to work when violating this restriction.

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.

Note: The constructor of an entity is only ever invoked when *you* construct a new instance with the *new* keyword. Doctrine never calls entity constructors, thus you are free to use them as you wish and even have it require arguments of any type.

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.

Serializing entities

Serializing entities can be problematic and is not really recommended, at least not as long as an entity instance still holds references to proxy objects or is still managed by an EntityManager. If you intend to serialize (and unserialize) entity instances that still hold references to proxy objects you may run into problems with private properties because of technical limitations. Proxy objects implement __sleep and it is not possible for __sleep to return names of private properties in parent classes. On the other hand it is not a solution for proxy objects to implement Serializable because Serializable does not work well with any potential cyclic object references (at least we did not find a way yet, if you did, please contact us).

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 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.

9.2.2 Installation

The installation chapter has moved to Installation and Configuration.

9.2.3 Installation and Configuration

Doctrine can be installed with Composer. For older versions we still have PEAR packages.

Define the following requirement in your composer.json file:

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

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

Class loading

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

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

Obtaining an EntityManager

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

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

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

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

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

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

Or if you prefer XML:

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

Or if you prefer YAML:

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

Inside the `Setup` methods several assumptions are made:

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

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

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

Setting up the Commandline Tool

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

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

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

On Doctrine 2.4 and above:

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

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

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

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

On Doctrine 2.3 and below:

```
<?php
// cli-config.php
require_once 'my_bootstrap.php';

// Any way to access the EntityManager from  your application
$em = GetMyEntityManager();

$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($em->getConnection()),
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em)
));
```

9.2.4 Frequently Asked Questions

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

Database Schema

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

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

Entity Classes

I access a variable and its null, what is wrong?

If this variable is a public variable then you are violating one of the criteria for entities. All properties have to be protected or private for the proxy object pattern to work.

How can I add default values to a column?

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

```
class User
{
    const STATUS_DISABLED = 0;
    const STATUS_ENABLED = 1;

    private $algorithm = "sha1";
    private $status = self:STATUS_DISABLED;
}
```

.

Mapping

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

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

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

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

Associations

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

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

You can solve this exception by:

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

How can I filter an association?

Natively you can't filter associations in 2.0 and 2.1. You should use DQL queries to query for the filtered set of entities.

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

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

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

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

The many-to-many association is only supporting foreign keys in the table definition To work with many-to-many tables containing extra columns you have to use the foreign keys as primary keys feature of Doctrine introduced in version 2.1.

See *the tutorial on composite primary keys for more information*.

How can i paginate fetch-joined collections?

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

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

- DoctrineExtensions
- Pagerfanta

Why does pagination not work correctly with fetch joins?

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

See the previous question for a solution to this task.

Inheritance

Can I use Inheritance with Doctrine 2?

Yes, you can use Single- or Joined-Table Inheritance in Doctrine 2.

See the documentation chapter on *inheritance mapping* for the details.

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

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

EntityGenerator

Why does the EntityGenerator not do X?

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

Why does the EntityGenerator not generate inheritance correctly?

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

Performance

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

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

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

Doctrine Query Language

What is DQL?

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

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

It also has some drawbacks of course:

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

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

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

A Query fails, how can I debug it?

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

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

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

9.2.5 Basic Mapping

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

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

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

Guide Assumptions

You should have already *installed and configure* Doctrine.

Creating Classes for the Database

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

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

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

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

- *Docblock Annotations*
- *XML*
- *YAML*
- *PHP code*

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

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

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

- *PHP*

```php
<?php
/** @Entity */
class Message
{
    //...
}
```

- *XML*

```xml
<doctrine-mapping>
  <entity name="Message">
      <!-- ... -->
  </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Message:
  type: entity
  # ...
```

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

- *PHP*

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

- *XML*

```xml
<doctrine-mapping>
  <entity name="Message" table="message">
      <!-- ... -->
  </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Message:
  type: entity
  table: message
  # ...
```

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

Property Mapping

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

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

- *PHP*

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

- *XML*

```xml
<doctrine-mapping>
  <entity name="Message">
    <field name="id" type="integer" />
    <field name="text" length="140" />
    <field name="postedAt" column="posted_at" type="datetime" />
  </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Message:
  type: entity
  fields:
    id:
      type: integer
    text:
      length: 140
    postedAt:
      type: datetime
      column: posted_at
```

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

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

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

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

Doctrine Mapping Types

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

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

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

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

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

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

Warning: All Date types assume that you are exclusively using the default timezone set by date_default_timezone_set() or by the php.ini configuration `date.timezone`. Working with different timezones will cause troubles and unexpected behavior.
If you need specific timezone handling you have to handle this in your domain, converting all the values back and forth from UTC. There is also a *cookbook entry* on working with datetimes that gives hints for implementing multi timezone applications.

Identifiers / Primary Keys

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

- *PHP*

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

- *XML*

```xml
<doctrine-mapping>
  <entity name="Message">
    <id name="id" type="integer">
        <generator strategy="AUTO" />
    </id>
    <!-- -->
  </entity>
</doctrine-mapping>
```

- *YAML*

```
Message:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    # fields here
```

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

Identifier Generation Strategies

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

Here is the list of possible generation strategies:

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

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

- *PHP*

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

- *XML*

```xml
<doctrine-mapping>
  <entity name="Message">
    <id name="id" type="integer">
        <generator strategy="SEQUENCE" />
        <sequence-generator sequence-name="message_seq" allocation-size="100" initial-value="1"
    </id>
  </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Message:
  type: entity
  id:
    id:
      type: integer
      generator:
        strategy: SEQUENCE
      sequenceGenerator:
        sequenceName: message_seq
        allocationSize: 100
        initialValue: 1
```

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

The allocationSize is a powerful feature to optimize INSERT performance of Doctrine. The allocationSize specifies by how much values the sequence is incremented whenever the next value is retrieved. If this is larger than 1 (one) Doctrine can generate identifier values for the allocationSizes amount of entities. In the above example with `allocationSize=100` Doctrine 2 would only need to access the sequence once to generate the identifiers for 100 new entities.

The default allocationSize for a @SequenceGenerator is currently 10.

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

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

Composite Keys

with Doctrine 2 you can use composite primary keys, using `@Id` on more then one column. Some restrictions exist opposed to using a single identifier in this case: The use of the `@GeneratedValue` annotation is not supported, which means you can only use composite keys if you generate the primary key values yourself before calling `EntityManager#persist()` on the entity.

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

Quoting Reserved Words

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

```php
<?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.

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

For more control over column quoting the `Doctrine\ORM\Mapping\QuoteStrategy` interface was introduced in 2.3. It is invoked for every column, table, alias and other SQL names. You can implement the QuoteStrategy and set it by calling `Doctrine\ORM\Configuration#setQuoteStrategy()`.

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

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

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

9.2.6 Association Mapping

This chapter explains mapping associations between objects.

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

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

This chapter is split into three different sections.

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

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

Many-To-One, Unidirectional

A many-to-one association is the most common association between objects.

- *PHP*

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

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

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

- *XML*

```
<doctrine-mapping>
    <entity name="User">
        <many-to-one field="address" target-entity="Address">
            <join-column name="address_id" referenced-column-name="id" />
        </many-to-one>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  manyToOne:
    address:
      targetEntity: Address
      joinColumn:
        name: address_id
        referencedColumnName: id
```

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

Generated MySQL Schema:

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

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

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

One-To-One, Unidirectional

Here is an example of a one-to-one association with a `Product` entity that references one `Shipping` entity. The `Shipping` does not reference back to the `Product` so that the reference is said to be unidirectional, in one direction only.

- *PHP*

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

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

    // ...
}

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

- *XML*

```xml
<doctrine-mapping>
    <entity class="Product">
        <one-to-one field="shipping" target-entity="Shipping">
            <join-column name="shipping_id" referenced-column-name="id" />
        </one-to-one>
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Product:
  type: entity
  oneToOne:
    shipping:
      targetEntity: Shipping
      joinColumn:
        name: shipping_id
        referencedColumnName: id
```

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

Generated MySQL Schema:

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

One-To-One, Bidirectional

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

- *PHP*

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

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

    // ...
}

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

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

    // ...
}
```

- *XML*

```xml
<doctrine-mapping>
    <entity name="Customer">
        <one-to-one field="cart" target-entity="Cart" mapped-by="customer" />
    </entity>
    <entity name="Cart">
        <one-to-one field="customer" target-entity="Customer" inversed-by="cart">
            <join-column name="customer_id" referenced-column-name="id" />
        </one-to-one>
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Customer:
  oneToOne:
    cart:
      targetEntity: Cart
      mappedBy: customer
Cart:
  oneToOne:
    customer:
      targetEntity: Customer
      inversedBy: cart
```

```
          joinColumn:
            name: customer_id
            referencedColumnName: id
```

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

Generated MySQL Schema:

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

See how the foreign key is defined on the owning side of the relation, the table `Cart`.

One-To-One, Self-referencing

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

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

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

    // ...
}
```

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

With the generated MySQL Schema:

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

One-To-Many, Bidirectional

A one-to-many association has to be bidirectional, unless you are using an additional join-table. This is necessary, because of the foreign key in a one-to-many association being defined on the "many" side. Doctrine needs a many-to-one association that defines the mapping of this foreign key.

This bidirectional mapping requires the `mappedBy` attribute on the `OneToMany` association and the `inversedBy` attribute on the `ManyToOne` association.

- *PHP*

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

  /** @Entity **/
  class Product
  {
      // ...
      /**
       * @OneToMany(targetEntity="Feature", mappedBy="product")
       **/
      private $features;
      // ...

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

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

- *XML*

  ```xml
  <doctrine-mapping>
      <entity name="Product">
          <one-to-many field="features" target-entity="Feature" mapped-by="product" />
      </entity>
      <entity name="Feature">
          <many-to-one field="product" target-entity="Product" inversed-by="features">
              <join-column name="product_id" referenced-column-name="id" />
          </many-to-one>
      </entity>
  </doctrine-mapping>
  ```

- *YAML*

  ```yaml
  Product:
    type: entity
    oneToMany:
      features:
        targetEntity: Feature
        mappedBy: product
  Feature:
    type: entity
    manyToOne:
      product:
  ```

```
          targetEntity: Product
          inversedBy: features
          joinColumn:
            name: product_id
            referencedColumnName: id
```

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

Generated MySQL Schema:

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

One-To-Many, Unidirectional with Join Table

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

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

- *PHP*

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

    /**
     * @ManyToMany(targetEntity="Phonenumber")
     * @JoinTable(name="users_phonenumbers",
     *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="phonenumber_id", referencedColumnName="id", un
     *      )
     **/
    private $phonenumbers;

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

    // ...
}

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

- *XML*

  ```
  <doctrine-mapping>
      <entity name="User">
          <many-to-many field="phonenumbers" target-entity="Phonenumber">
              <join-table name="users_phonenumbers">
                  <join-columns>
                      <join-column name="user_id" referenced-column-name="id" />
                  </join-columns>
                  <inverse-join-columns>
                      <join-column name="phonenumber_id" referenced-column-name="id" unique="true"
                  </inverse-join-columns>
              </join-table>
          </many-to-many>
      </entity>
  </doctrine-mapping>
  ```

- *YAML*

  ```
  User:
    type: entity
    manyToMany:
      phonenumbers:
        targetEntity: Phonenumber
        joinTable:
          name: users_phonenumbers
          joinColumns:
            user_id:
              referencedColumnName: id
          inverseJoinColumns:
            phonenumber_id:
              referencedColumnName: id
              unique: true
  ```

Generates the following MySQL Schema:

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

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

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

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

One-To-Many, Self-referencing

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

- *PHP*

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

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

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

- *XML*

```xml
<doctrine-mapping>
    <entity name="Category">
        <one-to-many field="children" target-entity="Category" mapped-by="parent" />
        <many-to-one field="parent" target-entity="Category" inversed-by="children" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Category:
  type: entity
  oneToMany:
    children:
      targetEntity: Category
      mappedBy: parent
  manyToOne:
    parent:
      targetEntity: Category
      inversedBy: children
```

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

Generated MySQL Schema:

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

Many-To-Many, Unidirectional

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

- *PHP*

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

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

    // ...

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

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

- *XML*

```xml
<doctrine-mapping>
    <entity name="User">
        <many-to-many field="groups" target-entity="Group">
            <join-table name="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>
```

- *YAML*

```
User:
  type: entity
  manyToMany:
    groups:
      targetEntity: Group
      joinTable:
        name: users_groups
        joinColumns:
          user_id:
            referencedColumnName: id
        inverseJoinColumns:
          group_id:
            referencedColumnName: id
```

Generated MySQL Schema:

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

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

Many-To-Many, Bidirectional

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

- *PHP*

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

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

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

    // ...
}

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

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

    // ...
}
```

- *XML*

```
<doctrine-mapping>
    <entity name="User">
        <many-to-many field="groups" inversed-by="users" target-entity="Group">
            <join-table name="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>

    <entity name="Group">
        <many-to-many field="users" mapped-by="groups" target-entity="User"/>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
User:
  type: entity
  manyToMany:
    groups:
      targetEntity: Group
      inversedBy: users
      joinTable:
        name: users_groups
        joinColumns:
          user_id:
            referencedColumnName: id
        inverseJoinColumns:
          group_id:
            referencedColumnName: id
```

```
Group:
  type: entity
  manyToMany:
    users:
      targetEntity: User
      mappedBy: groups
```

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

Owning and Inverse Side on a ManyToMany association

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

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

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

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

class Tag
{
    private $articles;

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

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

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

Many-To-Many, Self-referencing

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

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

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

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

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

    // ...
}
```

Generated MySQL Schema:

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

Mapping Defaults

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

```
name: "<fieldname>_id"
referencedColumnName: "id"
```

As an example, consider this mapping:

- *PHP*

  ```php
  <?php
  /** @OneToOne(targetEntity="Shipping") **/
  private $shipping;
  ```

- *XML*

```xml
<doctrine-mapping>
    <entity class="Product">
        <one-to-one field="shipping" target-entity="Shipping" />
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Product:
  type: entity
  oneToOne:
    shipping:
      targetEntity: Shipping
```

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

- *PHP*

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

- *XML*

```xml
<doctrine-mapping>
    <entity class="Product">
        <one-to-one field="shipping" target-entity="Shipping">
            <join-column name="shipping_id" referenced-column-name="id" />
        </one-to-one>
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
Product:
  type: entity
  oneToOne:
    shipping:
      targetEntity: Shipping
      joinColumn:
        name: shipping_id
        referencedColumnName: id
```

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

- *PHP*

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

- *XML*

  ```
  <doctrine-mapping>
      <entity class="User">
          <many-to-many field="groups" target-entity="Group" />
      </entity>
  </doctrine-mapping>
  ```

- *YAML*

  ```
  User:
    type: entity
    manyToMany:
      groups:
        targetEntity: Group
  ```

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

- *PHP*

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

- *XML*

  ```
  <doctrine-mapping>
      <entity class="User">
          <many-to-many field="groups" target-entity="Group">
              <join-table name="User_Group">
                  <join-columns>
                      <join-column id="User_id" referenced-column-name="id" />
                  </join-columns>
                  <inverse-join-columns>
                      <join-column id="Group_id" referenced-column-name="id" />
                  </inverse-join-columns>
              </join-table>
          </many-to-many>
      </entity>
  </doctrine-mapping>
  ```

- *YAML*

  ```
  User:
    type: entity
    manyToMany:
      groups:
        targetEntity: Group
        joinTable:
          name: User_Group
  ```

```
            joinColumns:
              User_id:
                referencedColumnName: id
            inverseJoinColumns:
              Group_id:
                referencedColumnName: id
```

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

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

Collections

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

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

Initializing Collections

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

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

/** @Entity **/
class User
{
    /** @ManyToMany(targetEntity="Group") **/
    private $groups;

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

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

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

```php
<?php
$group = new Group();
$user = new User();
$user->getGroups()->add($group);
```

9.2.7 Inheritance Mapping

Mapped Superclasses

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

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

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

Example:

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

    // ... more fields and methods
}

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

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

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

```sql
CREATE TABLE EntitySubClass (mapped1 INTEGER NOT NULL, mapped2 TEXT NOT NULL, id INTEGER NOT NULL, na
```

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

Single Table Inheritance

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

Example:

- *PHP*

```php
<?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
{
    // ...
}
```

- *YAML*

```yaml
MyProject\Model\Person:
  type: entity
  inheritanceType: SINGLE_TABLE
  discriminatorColumn:
    name: discr
    type: string
  discriminatorMap:
    person: Person
    employee: Employee

MyProject\Model\Employee:
  type: entity
```

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`.

- All entity classes that is part of the mapped entity hierarchy (including the topmost class) should be specified in the @DiscriminatorMap. In the case above Person class included.
- The names of the classes in the discriminator map do not need to be fully qualified if the classes are contained in the same namespace as the entity class on which the discriminator map is applied.
- If no discriminator map is provided, then the map is generated automatically. The automatically generated discriminator map contains the lowercase short name of each class as key.

Design-time considerations

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

Performance impact

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

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

SQL Schema considerations

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

Class Table Inheritance

Class Table Inheritance is an inheritance mapping strategy where each class in a hierarchy is mapped to several tables: its own table and the tables of all parent classes. The table of a child class is linked to the table of a parent class through a foreign key constraint. Doctrine 2 implements this strategy through the use of a discriminator column in the topmost table of the hierarchy because this is the easiest way to achieve polymorphic queries with Class Table Inheritance.

Example:

```php
<?php
namespace MyProject\Model;

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

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

Things to note:

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

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

Design-time considerations

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

Performance impact

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

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

SQL Schema considerations

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

Overrides

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

Association Override

Override a mapping for an entity relationship.

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

Example:

- *PHP*

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

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

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

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

- *XML*

```
<!-- user mapping -->
<doctrine-mapping>
  <mapped-superclass name="MyProject\Model\User">
        <!-- other fields mapping -->
        <many-to-many field="groups" target-entity="Group" inversed-by="users">
            <cascade>
                <cascade-persist/>
                <cascade-merge/>
                <cascade-detach/>
            </cascade>
            <join-table name="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>
    </mapped-superclass>
</doctrine-mapping>

<!-- admin mapping -->
<doctrine-mapping>
    <entity name="MyProject\Model\Admin">
        <association-overrides>
            <association-override name="groups">
                <join-table name="users_admingroups">
                    <join-columns>
                        <join-column name="adminuser_id"/>
                    </join-columns>
                    <inverse-join-columns>
                        <join-column name="admingroup_id"/>
                    </inverse-join-columns>
                </join-table>
            </association-override>
            <association-override name="address">
                <join-columns>
                    <join-column name="adminaddress_id" referenced-column-name="id"/>
                </join-columns>
            </association-override>
        </association-overrides>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
# user mapping
MyProject\Model\User:
  type: mappedSuperclass
```

```
    # other fields mapping
    manyToOne:
      address:
        targetEntity: Address
        joinColumn:
          name: address_id
          referencedColumnName: id
        cascade: [ persist, merge ]
    manyToMany:
      groups:
        targetEntity: Group
        joinTable:
          name: users_groups
          joinColumns:
            user_id:
              referencedColumnName: id
          inverseJoinColumns:
            group_id:
              referencedColumnName: id
        cascade: [ persist, merge, detach ]

  # admin mapping
  MyProject\Model\Admin:
    type: entity
    associationOverride:
      address:
        joinColumn:
          adminaddress_id:
            name: adminaddress_id
            referencedColumnName: id
      groups:
        joinTable:
          name: users_admingroups
          joinColumns:
            adminuser_id:
              referencedColumnName: id
          inverseJoinColumns:
            admingroup_id:
              referencedColumnName: id
```

Things to note:

- The “association override” specifies the overrides base on the property name.
- This feature is available for all kind of associations. (OneToOne, OneToMany, ManyToOne, ManyToMany)
- The association type *CANNOT* be changed.
- The override could redefine the joinTables or joinColumns depending on the association type.

Attribute Override

Override the mapping of a field.

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

- *PHP*

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

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

    // other fields mapping
}

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

- *XML*

```
<!-- user mapping -->
<doctrine-mapping>
  <mapped-superclass name="MyProject\Model\User">
        <id name="id" type="integer" column="user_id" length="150">
            <generator strategy="AUTO"/>
        </id>
        <field name="name" column="user_name" type="string" length="250" nullable="true" unique=
        <many-to-one field="address" target-entity="Address">
            <cascade>
                <cascade-persist/>
                <cascade-merge/>
            </cascade>
            <join-column name="address_id" referenced-column-name="id"/>
        </many-to-one>
        <!-- other fields mapping -->
    </mapped-superclass>
</doctrine-mapping>

<!-- admin mapping -->
<doctrine-mapping>
    <entity name="MyProject\Model\Guest">
        <attribute-overrides>
            <attribute-override name="id">
                <field column="guest_id" length="140"/>
            </attribute-override>
            <attribute-override name="name">
                <field column="guest_name" type="string" length="240" nullable="false" unique="t
            </attribute-override>
        </attribute-overrides>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
# user mapping
MyProject\Model\User:
  type: mappedSuperclass
  id:
    id:
      type: integer
      column: user_id
      length: 150
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
      column: user_name
      length: 250
      nullable: true
      unique: false
  #other fields mapping


# guest mapping
MyProject\Model\Guest:
  type: entity
  attributeOverride:
    id:
      column: guest_id
      type: integer
      length: 140
    name:
      column: guest_name
      type: string
      length: 240
      nullable: false
      unique: true
```

Things to note:

- The “attribute override” specifies the overrides base on the property name.

- The column type *CANNOT* be changed. If the column type is not equal you get a `MappingException`
- The override can redefine all the columns except the type.

Query the Type

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

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

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

```php
<?php
$query = $em->createQuery("SELECT staff FROM MyProject\Model\Staff staff WHERE staff INSTANCE OF MyPr
$staffs = $query->getResult();
```

9.2.8 Working with Objects

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

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

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

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

Entities and the Identity Map

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

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

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

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

In this case the Article is accessed from the entity manager twice, but modified in between. Doctrine 2 realizes this and will only ever give you access to one instance of the Article with ID 1234, no matter how often do you retrieve it from the EntityManager and even no matter what kind of Query method you are using (find, Repository Finder or DQL). This is called “Identity Map” pattern, which means Doctrine keeps a map of each entity and ids that have been retrieved per PHP request and keeps returning you the same instances.

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

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

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

Entity Object Graph Traversal

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

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

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

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

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

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

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

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

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

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

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

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

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

// accessing a method of the user instance triggers the lazy-load
```

```php
echo "Author: " . $article->getAuthor()->getName() . "\n";

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

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

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

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

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

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

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

Persisting entities

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

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

Example:

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

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

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

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

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.

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

Example:

```php
<?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 as a result of the flush operation.

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

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

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

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

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
<?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
<?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.
```

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

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

- If X is a detached entity, the state of X is copied onto a pre-existing managed entity instance X' of the same identity.
- If X is a new entity instance, a new managed copy X' will be created and the state of X is copied onto this managed instance.
- If X is a removed entity instance, an InvalidArgumentException will be thrown.
- If X is a managed entity, it is ignored by the merge operation, however, the merge operation is cascaded to entities referenced by relationships from X if these relationships have been mapped with the cascade element value MERGE or ALL (see "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 don't match. This usually means that the entity has been modified while being detached.

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

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

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

Synchronization with the Database

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

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

Effects of Database and UnitOfWork being Out-Of-Sync

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

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

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

Synchronizing New and Managed Entities

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

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

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

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

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

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

Synchronizing Removed Entities

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

The size of a Unit of Work

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

The cost of flushing

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

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

You can get the size of a UnitOfWork as follows:

```php
<?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.

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

Direct access to a Unit of Work

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

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

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

Entity State

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

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

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

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

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

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

Querying

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

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

By Simple Conditions

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

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

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

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

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

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

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

Be careful that this only works by passing the ID of the associated entity, not yet by passing the associated entity itself.

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

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

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

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

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

```php
<?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 Criteria

New in version 2.3.

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

See the *Working with Associations: Filtering collections*.

By Eager Loading

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

By Lazy Loading

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

By DQL

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

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

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

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

Note that this query contains no knowledge about the relational schema, only about the object model. DQL supports positional as well as named parameters, many functions, (fetch) joins, aggregates, subqueries and much more. Detailed information about DQL and its syntax as well as the Doctrine class can be found in *the dedicated chapter*. For programmatically building up queries based on conditions that are only known at runtime, Doctrine provides the special `Doctrine\ORM\QueryBuilder` class. More information on constructing queries with a QueryBuilder can be found *in Query Builder chapter*.

By Native Queries

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

Custom Repositories

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

```php
<?php
namespace MyDomain\Model;

use Doctrine\ORM\EntityRepository;

/**
 * @entity(repositoryClass="MyDomain\Model\UserRepository")
 */
class User
{

}

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

You can access your repository now by calling:

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

$admins = $em->getRepository('MyDomain\Model\User')->getAllAdminUsers();
```

9.2.9 Working with Associations

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

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

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

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

Association Example Entities

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Establishing Associations

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

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

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

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

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

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

$em->flush();

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

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

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

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

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

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

```
class Comment
{
    // ...

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

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

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

$em->flush();

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

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

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

Removing Associations

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

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

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

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

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

Notice how both sides of the bidirectional association are always updated. Unidirectional associations are consequently simpler to handle. Also note that if you use type-hinting in your methods, i.e. `setAddress(Address $address)`, PHP will only allow null values if `null` is set as default value. Otherwise 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.

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

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

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

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

Association Management Methods

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Synchronizing Bidirectional Collections

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

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

```
<?php
$user->getFavorites()->add($favoriteComment);
// not calling $favoriteComment->getUserFavorites()->add($user);

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

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

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

Transitive persistence / Cascade Operations

Persisting, removing, detaching, refreshing and merging individual entities can become pretty cumbersome, especially when a highly interweaved object graph 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.
- refresh : Cascades refresh operations to the associated entities.
- all : Cascades persist, remove, merge, refresh and detach operations to associated entities.

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

However, pulling objects graph into memory on cascade can cause considerable performance overhead, especially when cascading collections are large. Makes sure to weigh the benefits and downsides of each cascade operation that you define.

To rely on the database level cascade operations for the delete operation instead, you can configure each join column with the **onDelete** option. See the respective mapping driver chapters for more information.

The following example is an extension to the User-Comment example of this chapter. Suppose in our application a user is created whenever he writes his first comment. In this case we would use the following code:

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

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

Even if you *persist* a new User that contains our new Comment this code would fail if you removed the call to `EntityManager#persist($myFirstComment)`. Doctrine 2 does not cascade the persist operation to all nested entities that are new as well.

More complicated is the deletion of all of a user's comments when he is removed from the system:

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

foreach ($user->getAuthoredComments() as $comment) {
    $em->remove($comment);
}
$em->remove($user);
$em->flush();
```

Without the loop over all the authored comments Doctrine would use an UPDATE statement only to set the foreign key to NULL and only the User would be deleted from the database during the flush()-Operation.

To have Doctrine handle both cases automatically we can change the `User#commentsAuthored` property to cascade both the "persist" and the "remove" operation.

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

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

Persistence by Reachability: Cascade Persist

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

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

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

Orphan Removal

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

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

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

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

```
<?php

namespace Addressbook;

use Doctrine\Common\Collections\ArrayCollection;

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

    /** @OneToOne(targetEntity="StandingData", orphanRemoval=true) */
    private $standingData;

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

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

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

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

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

```
<?php

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

$em->flush();
```

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

Filtering Collections

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

```
<?php

use Doctrine\Common\Collections\Criteria;

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

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

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

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

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

```
<?php

use Doctrine\Common\Collections;

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

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

- `andX($arg1, $arg2, ...)`
- `orX($arg1, $arg2, ...)`
- `eq($field, $value)`
- `gt($field, $value)`
- `lt($field, $value)`
- `lte($field, $value)`
- `gte($field, $value)`
- `neq($field, $value)`
- `isNull($field)`
- `in($field, array $values)`
- `notIn($field, array $values)`
- `contains($field, $value)`

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

9.2.10 Events

Doctrine 2 features a lightweight event system that is part of the Common package. Doctrine uses it to dispatch system events, mainly *lifecycle events*. You can also use it for your own custom events.

The Event System

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

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

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

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

    private $_evm;

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

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

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

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

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

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

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

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

The Doctrine 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
{
    public $preFooInvoked = false;

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

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

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

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

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

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

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

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

Naming convention

Events being used with the Doctrine 2 EventManager are best named with camelcase and the value of the corresponding constant should be the name of the constant itself, even with spelling. This has several reasons:

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

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

Lifecycle Events

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

- preRemove - The preRemove event occurs for a given entity before the respective EntityManager remove operation for that entity is executed. It is not called for a DQL DELETE statement.
- postRemove - The postRemove event occurs for an entity after the entity has been deleted. It will be invoked after the database delete operations. It is not called for a DQL DELETE statement.
- prePersist - The prePersist event occurs for a given entity before the respective EntityManager persist operation for that entity is executed. It should be noted that this event is only triggered on *initial* persist of an entity (i.e. it does not trigger on future updates).
- postPersist - The postPersist event occurs for an entity after the entity has been made persistent. It will be invoked after the database insert operations. Generated primary key values are available in the postPersist event.
- preUpdate - The preUpdate event occurs before the database update operations to entity data. It is not called for a DQL UPDATE statement.
- postUpdate - The postUpdate event occurs after the database update operations to entity data. It is not called for a DQL UPDATE statement.
- postLoad - The postLoad event occurs for an entity after the entity has been loaded into the current EntityManager from the database or after the refresh operation has been applied to it.
- loadClassMetadata - The loadClassMetadata event occurs after the mapping metadata for a class has been loaded from a mapping source (annotations/xml/yaml). This event is not a lifecycle callback.
- onClassMetadataNotFound - Loading class metadata for a particular requested class name failed. Manipulating the given event args instance allows providing fallback metadata even when no actual metadata exists or could be found. This event is not a lifecycle callback.
- preFlush - The preFlush event occurs at the very beginning of a flush operation. This event is not a lifecycle callback.
- onFlush - The onFlush event occurs after the change-sets of all managed entities are computed. This event is not a lifecycle callback.
- postFlush - The postFlush event occurs at the end of a flush operation. This event is not a lifecycle callback.

- onClear - The onClear event occurs when the EntityManager#clear() operation is invoked, after all references to entities have been removed from the unit of work. This event is not a lifecycle callback.

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

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

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

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

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

- Lifecycle Callbacks are methods on the entity classes that are called when the event is triggered. As of v2.4 they receive some kind of `EventArgs` instance.
- Lifecycle Event Listeners and Subscribers are classes with specific callback methods that receives some kind of `EventArgs` instance.

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

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

Lifecycle Callbacks

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

```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:i:s');
    }

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

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

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

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

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

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

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

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

XML would look something like this:

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

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

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

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

```php
<?php
// ...

class User
{
    // ...

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

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

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

Lifecycle Callbacks Event Argument

New in version 2.4.

Since 2.4 the triggered event is given to the lifecycle-callback.

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

```php
<?php
// ...

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

Listening and subscribing to Lifecycle Events

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

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

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

A lifecycle event listener looks like the following:

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

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

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

A lifecycle event subscriber may looks like this:

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

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

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

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

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

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

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

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

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

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

Implementing Event Listeners

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

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

prePersist

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

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

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

The following restrictions apply to `prePersist`:

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

preRemove

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

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

preFlush

`preFlush` is called at `EntityManager#flush()` before anything else. `EntityManager#flush()` can be called safely inside its listeners.

```php
<?php

use Doctrine\ORM\Event\PreFlushEventArgs;

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

onFlush

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

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

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

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

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

        }

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

        }

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

        }

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

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

        }
    }
}
```

The following restrictions apply to the onFlush event:

- If you create and persist a new entity in `onFlush`, then calling `EntityManager#persist()` is not enough. You have to execute an additional call to `$unitOfWork->computeChangeSet($classMetadata, $entity)`.
- Changing primitive fields or associations requires you to explicitly trigger a re-computation of the changeset of the affected entity. This can be done by calling `$unitOfWork->recomputeSingleEntityChangeSet($classMetadata, $entity)`.

postFlush

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

```php
<?php

use Doctrine\ORM\Event\PostFlushEventArgs;

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

preUpdate

PreUpdate is the most restrictive to use event, since it is called right before an update statement is called for an entity inside the `EntityManager#flush()` method.

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

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

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

A simple example for this event looks like:

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

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

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

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

Restrictions for this event:

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

postUpdate, postRemove, postPersist

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

postLoad

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

Entity listeners

New in version 2.4.

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

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

  ```php
  <?php
  namespace MyProject\Entity;

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

- *XML*

  ```xml
  <doctrine-mapping>
      <entity name="MyProject\Entity\User">
          <entity-listeners>
              <entity-listener class="UserListener"/>
          </entity-listeners>
          <!-- .... -->
      </entity>
  </doctrine-mapping>
  ```

- *YAML*

  ```yaml
  MyProject\Entity\User:
    type: entity
    entityListeners:
      UserListener:
    # ....
  ```

Entity listeners class

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

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

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

```
    }
}
```

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

- *PHP*

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

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

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

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

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

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

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

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

- *XML*

```xml
<doctrine-mapping>
    <entity name="MyProject\Entity\User">
         <entity-listeners>
            <entity-listener class="UserListener">
                <lifecycle-callback type="preFlush"      method="preFlushHandler"/>
                <lifecycle-callback type="postLoad"      method="postLoadHandler"/>

                <lifecycle-callback type="postPersist"   method="postPersistHandler"/>
                <lifecycle-callback type="prePersist"    method="prePersistHandler"/>

                <lifecycle-callback type="postUpdate"    method="postUpdateHandler"/>
                <lifecycle-callback type="preUpdate"     method="preUpdateHandler"/>

                <lifecycle-callback type="postRemove"    method="postRemoveHandler"/>
                <lifecycle-callback type="preRemove"     method="preRemoveHandler"/>
            </entity-listener>
        </entity-listeners>
        <!-- .... -->
    </entity>
</doctrine-mapping>
```

- *YAML*

```
MyProject\Entity\User:
  type: entity
  entityListeners:
    UserListener:
      preFlush: [preFlushHandler]
      postLoad: [postLoadHandler]

      postPersist: [postPersistHandler]
      prePersist: [prePersistHandler]

      postUpdate: [postUpdateHandler]
      preUpdate: [preUpdateHandler]

      postRemove: [postRemoveHandler]
      preRemove: [preRemoveHandler]
  # ....
```

Entity listeners resolver

Doctrine invokes the listener resolver to get the listener instance.

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

Specifying an entity listener instance :

```
<?php
// User.php

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

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

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

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

Implementing your own resolver :

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

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

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

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

Load ClassMetadata Event

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

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

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

9.2.11 Doctrine Internals explained

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

How Doctrine keeps track of Objects

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

How Doctrine Detects Changes

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

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

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

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

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

Query Internals

The different ORM Layers

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

Hydration

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

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

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

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

Persisters

tbr

UnitOfWork

tbr

ResultSetMapping

tbr

DQL Parser

tbr

SQLWalker

tbr

EntityManager

tbr

ClassMetadataFactory

tbr

9.2.12 Association Updates: Owning Side and Inverse Side

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

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

Bidirectional Associations

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

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

Important concepts

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

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

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

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

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

9.2.13 Transactions and Concurrency

Transaction Demarcation

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

For the most part, Doctrine 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 transaction.

However, Doctrine 2 also allows (and encourages) you to take over and control transaction demarcation yourself.

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

Approach 1: Implicitly

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

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

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

Approach 2: Explicitly

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

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

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

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

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

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

Exception Handling

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

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

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

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

Locking Support

Doctrine 2 offers support for Pessimistic- and Optimistic-locking strategies natively. This allows to take very fine-grained control over what kind of locking is required for your Entities in your application.

Optimistic Locking

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

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

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

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

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

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

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

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

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

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

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

$theEntityId = 1;
$expectedVersion = 184;

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

    // do the work

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

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

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

$theEntityId = 1;
$expectedVersion = 184;

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

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

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

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

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

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

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

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

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

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

And the change headline action (POST Request):

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

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

Pessimistic Locking

Doctrine 2 supports Pessimistic Locking at the database level. No attempt is being made to implement pessimistic locking inside Doctrine, rather vendor-specific and ANSI-SQL commands are used to acquire row-level locks. Every Entity can be part of a pessimistic lock, there is no special metadata required to use this feature.

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

Doctrine 2 currently supports two pessimistic lock modes:

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

You can use pessimistic locks in three different scenarios:

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

9.2.14 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.

Warning: 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!
    }
}
$em->flush(); //Persist objects that did not make up an entire batch
$em->clear();
```

Bulk Updates

There are 2 possibilities for bulk updates with Doctrine.

DQL UPDATE

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

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

Iterating results

An alternative solution for bulk updates is to use the `Query#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
$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;
}
$em->flush();
```

Note: 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.

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

Bulk Deletes

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

DQL DELETE

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

Example:

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

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

Iterating Large Results for Data-Processing

You can use the `iterate()` method just to iterate over a large result and no UPDATE or DELETE intention. The `IterableResult` instance returned from `$query->iterate()` implements the Iterator interface so you can process a large result without memory problems using the following approach:

```php
<?php
$q = $this->_em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
foreach ($iterableResult as $row) {
    // do stuff with the data in the row, $row[0] is always the object

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

Note: 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.

9.2.15 Doctrine Query Language

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

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

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

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

Types of DQL queries

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

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

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

SELECT queries

DQL SELECT clause

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:

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

Lets examine the query:

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

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

The SELECT clause allows to specify both class identification variables that signal the hydration of a complete entity class or just fields of the entity using the syntax `u.name`. Combinations of both are also allowed and it is possible to wrap both fields and identification values into aggregation and DQL functions. Numerical fields can be part of computations using mathematical operations. See the sub-section on Functions, Operators, Aggregates for more information.

Joins

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

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

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

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

Example:

Regular join of the address:

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

Fetch join of the address:

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

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

Note: Doctrine allows you to walk all the associations between all the objects in your domain model. Objects that were not already loaded from the database are replaced with lazy load proxy instances. Non-loaded Collections are also replaced by lazy-load instances that fetch all the contained objects upon first access. However relying on the lazy-load mechanism leads to many small queries executed against the database, which can significantly affect the performance of your application. **Fetch Joins** are the solution to hydrate most or all of the entities that you need in a single SELECT query.

Named and Positional Parameters

DQL supports both named and positional parameters, however in contrast to many SQL dialects positional parameters are specified with numbers, for example "?1", "?2" and so on. Named parameters are specified with ":name1", ":name2" and so on.

When referencing the parameters in `Query#setParameter($param, $value)` both named and positional parameters are used **without** their prefixes.

DQL SELECT Examples

This section contains a large set of DQL queries and some explanations of what is happening. The actual result also depends on the hydration mode.

Hydrate all User entities:

```
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u');
$users = $query->getResult(); // array of User objects
```

Retrieve the IDs of all CmsUsers:

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u');
$ids = $query->getResult(); // array of CmsUser ids
```

Retrieve the IDs of all users that have written an article:

```
<?php
$query = $em->createQuery('SELECT DISTINCT u.id FROM CmsArticle a JOIN a.user u');
$ids = $query->getResult(); // array of CmsUser ids
```

Retrieve all articles and sort them by the name of the articles users instance:

```
<?php
$query = $em->createQuery('SELECT a FROM CmsArticle a JOIN a.user u ORDER BY u.name ASC');
$articles = $query->getResult(); // array of CmsArticle objects
```

Retrieve the Username and Name of a CmsUser:

```
<?php
$query = $em->createQuery('SELECT u.username, u.name FROM CmsUser u');
$users = $query->getResult(); // array of CmsUser username and name values
echo $users[0]['username'];
```

Retrieve a ForumUser and his single associated entity:

```
<?php
$query = $em->createQuery('SELECT u, a FROM ForumUser u JOIN u.avatar a');
$users = $query->getResult(); // array of ForumUser objects with the avatar association loaded
echo get_class($users[0]->getAvatar());
```

Retrieve a CmsUser and fetch join all the phonenumbers he has:

```
<?php
$query = $em->createQuery('SELECT u, p FROM CmsUser u JOIN u.phonenumbers p');
$users = $query->getResult(); // array of CmsUser objects with the phonenumbers association loaded
$phonenumbers = $users[0]->getPhonenumbers();
```

Hydrate a result in Ascending:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id ASC');
$users = $query->getResult(); // array of ForumUser objects
```

Or in Descending Order:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u ORDER BY u.id DESC');
$users = $query->getResult(); // array of ForumUser objects
```

Using Aggregate Functions:

```
<?php
$query = $em->createQuery('SELECT COUNT(u.id) FROM Entities\User u');
$count = $query->getSingleScalarResult();
```

```
$query = $em->createQuery('SELECT u, count(g.id) FROM Entities\User u JOIN u.groups g GROUP BY u.id')
$result = $query->getResult();
```

With WHERE Clause and Positional Parameter:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$users = $query->getResult(); // array of ForumUser objects
```

With WHERE Clause and Named Parameter:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE u.username = :name');
$query->setParameter('name', 'Bob');
$users = $query->getResult(); // array of ForumUser objects
```

With Nested Conditions in WHERE Clause:

```
<?php
$query = $em->createQuery('SELECT u FROM ForumUser u WHERE (u.username = :name OR u.username = :name2
$query->setParameters(array(
    'name' => 'Bob',
    'name2' => 'Alice',
    'id' => 321,
));
$users = $query->getResult(); // array of ForumUser objects
```

With COUNT DISTINCT:

```
<?php
$query = $em->createQuery('SELECT COUNT(DISTINCT u.name) FROM CmsUser');
$users = $query->getResult(); // array of ForumUser objects
```

With Arithmetic Expression in WHERE clause:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE ((u.id + 5000) * u.id + 3) < 10000000');
$users = $query->getResult(); // array of ForumUser objects
```

Retrieve user entities with Arithmetic Expression in ORDER clause, using the `HIDDEN` keyword:

```
<?php
$query = $em->createQuery('SELECT u, u.posts_count + u.likes_count AS HIDDEN score FROM CmsUser u ORD
$users = $query->getResult(); // array of User objects
```

Using a LEFT JOIN to hydrate all user-ids and optionally associated article-ids:

```
<?php
$query = $em->createQuery('SELECT u.id, a.id as article_id FROM CmsUser u LEFT JOIN u.articles a');
$results = $query->getResult(); // array of user ids and every article_id for each user
```

Restricting a JOIN clause by additional conditions:

```
<?php
$query = $em->createQuery("SELECT u FROM CmsUser u LEFT JOIN u.articles a WITH a.topic LIKE :foo");
$query->setParameter('foo', '%foo%');
$users = $query->getResult();
```

Using several Fetch JOINs:

```
<?php
$query = $em->createQuery('SELECT u, a, p, c FROM CmsUser u JOIN u.articles a JOIN u.phonenumbers p
$users = $query->getResult();
```

BETWEEN in WHERE clause:

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id BETWEEN ?1 AND ?2');
$query->setParameter(1, 123);
$query->setParameter(2, 321);
$usernames = $query->getResult();
```

DQL Functions in WHERE clause:

```
<?php
$query = $em->createQuery("SELECT u.name FROM CmsUser u WHERE TRIM(u.name) = 'someone'");
$usernames = $query->getResult();
```

IN() Expression:

```
<?php
$query = $em->createQuery('SELECT u.name FROM CmsUser u WHERE u.id IN(46)');
$usernames = $query->getResult();

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id IN (1, 2)');
$users = $query->getResult();

$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.id NOT IN (1)');
$users = $query->getResult();
```

CONCAT() DQL Function:

```
<?php
$query = $em->createQuery("SELECT u.id FROM CmsUser u WHERE CONCAT(u.name, 's') = ?1");
$query->setParameter(1, 'Jess');
$ids = $query->getResult();

$query = $em->createQuery('SELECT CONCAT(u.id, u.name) FROM CmsUser u WHERE u.id = ?1');
$query->setParameter(1, 321);
$idUsernames = $query->getResult();
```

EXISTS in WHERE clause with correlated Subquery

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE EXISTS (SELECT p.phonenumber FROM CmsPhon
$ids = $query->getResult();
```

Get all users who are members of $group.

```
<?php
$query = $em->createQuery('SELECT u.id FROM CmsUser u WHERE :groupId MEMBER OF u.groups');
$query->setParameter('groupId', $group);
$ids = $query->getResult();
```

Get all users that have more than 1 phonenumber

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE SIZE(u.phonenumbers) > 1');
$users = $query->getResult();
```

Get all users that have no phonenumber

```php
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u WHERE u.phonenumbers IS EMPTY');
$users = $query->getResult();
```

Get all instances of a specific type, for use with inheritance hierarchies:

New in version 2.1.

```php
<?php
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u INSTAN
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u INSTAN
$query = $em->createQuery('SELECT u FROM Doctrine\Tests\Models\Company\CompanyPerson u WHERE u NOT IN
```

Get all users visible on a given website that have chosen certain gender:

New in version 2.2.

```php
<?php
$query = $em->createQuery('SELECT u FROM User u WHERE u.gender IN (SELECT IDENTITY(agl.gender) FROM S
```

New in version 2.4.

Starting with 2.4, the IDENTITY() DQL function also works for composite primary keys:

```php
<?php
$query = $em->createQuery("SELECT IDENTITY(c.location, 'latitude') AS latitude, IDENTITY(c.location,
```

Joins between entities without associations were not possible until version 2.4, where you can generate an arbitrary join with the following syntax:

```php
<?php
$query = $em->createQuery('SELECT u FROM User u JOIN Blacklist b WITH u.email = b.email');
```

Partial Object Syntax By default when you run a DQL query in Doctrine and select only a subset of the fields for a given entity, you do not receive objects back. Instead, you receive only arrays as a flat rectangular result set, similar to how you would if you were just using SQL directly and joining some data.

If you want to select partial objects you can use the `partial` DQL keyword:

```php
<?php
$query = $em->createQuery('SELECT partial u.{id, username} FROM CmsUser u');
$users = $query->getResult(); // array of partially loaded CmsUser objects
```

You use the partial syntax when joining as well:

```php
<?php
$query = $em->createQuery('SELECT partial u.{id, username}, partial a.{id, name} FROM CmsUser u JOIN
$users = $query->getResult(); // array of partially loaded CmsUser objects
```

"NEW" Operator Syntax New in version 2.4.

Using the `NEW` operator you can construct Data Transfer Objects (DTOs) directly from DQL queries.

- When using `SELECT NEW` you don't need to specify a mapped entity.
- You can specify any PHP class, it's only require that the constructor of this class matches the `NEW` statement.
- This approach involves determining exactly which columns you really need, and instantiating data-transfer object that containing a constructor with those arguments.

If you want to select data-transfer objects you should create a class:

```
<?php
class CustomerDTO
{
    public function __construct($name, $email, $city, $value = null)
    {
        // Bind values to the object properties.
    }
}
```

And then use the `NEW` DQL keyword :

```
<?php
$query = $em->createQuery('SELECT NEW CustomerDTO(c.name, e.email, a.city) FROM Customer c JOIN c.ema
$users = $query->getResult(); // array of CustomerDTO

<?php
$query = $em->createQuery('SELECT NEW CustomerDTO(c.name, e.email, a.city, SUM(o.value)) FROM Custome
$users = $query->getResult(); // array of CustomerDTO
```

Note that you can only pass scalar expressions to the constructor.

Using INDEX BY

The INDEX BY construct is nothing that directly translates into SQL but that affects object and array hydration. After each FROM and JOIN clause you specify by which field this class should be indexed in the result. By default a result is incremented by numerical keys starting with 0. However with INDEX BY you can specify any other column to be the key of your result, it really only makes sense with primary or unique fields though:

```
SELECT u.id, u.status, upper(u.name) nameUpper FROM User u INDEX BY u.id
JOIN u.phonenumbers p INDEX BY p.phonenumber
```

Returns an array of the following kind, indexed by both user-id then phonenumber-id:

```
array
  0 =>
    array
      1 =>
        object(stdClass)[299]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser' (length=33)
          public 'id' => int 1
          ..
      'nameUpper' => string 'ROMANB' (length=6)
  1 =>
    array
      2 =>
        object(stdClass)[298]
          public '__CLASS__' => string 'Doctrine\Tests\Models\CMS\CmsUser' (length=33)
          public 'id' => int 2
          ...
      'nameUpper' => string 'JWAGE' (length=5)
```

UPDATE queries

DQL not only allows to select your Entities using field names, you can also execute bulk updates on a set of entities using an DQL-UPDATE query. The Syntax of an UPDATE query works as expected, as the following example shows:

```
UPDATE MyProject\Model\User u SET u.password = 'new' WHERE u.id IN (1, 2, 3)
```

References to related entities are only possible in the WHERE clause and using sub-selects.

Warning: DQL UPDATE statements are ported directly into a Database UPDATE statement and therefore bypass any locking scheme, events and do not increment the version column. Entities that are already loaded into the persistence context will *NOT* be synced with the updated database state. It is recommended to call `EntityManager#clear()` and retrieve new instances of any affected entity.

DELETE queries

DELETE queries can also be specified using DQL and their syntax is as simple as the UPDATE syntax:

```
DELETE MyProject\Model\User u WHERE u.id = 4
```

The same restrictions apply for the reference of related entities.

Warning: DQL DELETE statements are ported directly into a Database DELETE statement and therefore bypass any events and checks for the version column if they are not explicitly added to the WHERE clause of the query. Additionally Deletes of specifies entities are *NOT* cascaded to related entities even if specified in the metadata.

Functions, Operators, Aggregates

DQL Functions

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

- IDENTITY(single_association_path_expression [, fieldMapping]) - Retrieve the foreign key column of association of the owning side
- ABS(arithmetic_expression)
- CONCAT(str1, str2)
- CURRENT_DATE() - Return the current date
- CURRENT_TIME() - Returns the current time
- CURRENT_TIMESTAMP() - Returns a timestamp of the current date and time.
- LENGTH(str) - Returns the length of the given string
- LOCATE(needle, haystack [, offset]) - Locate the first occurrence of the substring in the string.
- LOWER(str) - returns the string lowercased.
- MOD(a, b) - Return a MOD b.
- SIZE(collection) - Return the number of elements in the specified collection
- SQRT(q) - Return the square-root of q.
- SUBSTRING(str, start [, length]) - Return substring of given string.
- TRIM([LEADING | TRAILING | BOTH] ['trchar' FROM] str) - Trim the string by the given trim char, defaults to whitespaces.
- UPPER(str) - Return the upper-case of the given string.

- DATE_ADD(date, days, unit) - Add the number of days to a given date. (Supported units are DAY, MONTH)
- DATE_SUB(date, days, unit) - Substract the number of days from a given date. (Supported units are DAY, MONTH)
- DATE_DIFF(date1, date2) - Calculate the difference in days between date1-date2.

Arithmetic operators

You can do math in DQL using numeric values, for example:

```
SELECT person.salary * 1.5 FROM CompanyPerson person WHERE person.salary < 100000
```

Aggregate Functions

The following aggregate functions are allowed in SELECT and GROUP BY clauses: AVG, COUNT, MIN, MAX, SUM

Other Expressions

DQL offers a wide-range of additional expressions that are known from SQL, here is a list of all the supported constructs:

- `ALL/ANY/SOME` - Used in a WHERE clause followed by a sub-select this works like the equivalent constructs in SQL.
- `BETWEEN a AND b` and `NOT BETWEEN a AND b` can be used to match ranges of arithmetic values.
- `IN (x1, x2, ...)` and `NOT IN (x1, x2, ..)` can be used to match a set of given values.
- `LIKE ..` and `NOT LIKE ..` match parts of a string or text using % as a wildcard.
- `IS NULL` and `IS NOT NULL` to check for null values
- `EXISTS` and `NOT EXISTS` in combination with a sub-select

Adding your own functions to the DQL language

By default DQL comes with functions that are part of a large basis of underlying databases. However you will most likely choose a database platform at the beginning of your project and most likely never change it. For this cases you can easily extend the DQL parser with own specialized platform functions.

You can register custom DQL functions in your ORM Configuration:

```php
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($name, $class);
$config->addCustomNumericFunction($name, $class);
$config->addCustomDatetimeFunction($name, $class);

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

The functions have to return either a string, numeric or datetime value depending on the registered function type. As an example we will add a MySQL specific FLOOR() functionality. All the given classes have to implement the base class :

```php
<?php
namespace MyProject\Query\AST;

use \Doctrine\ORM\Query\AST\Functions\FunctionNode;
use \Doctrine\ORM\Query\Lexer;

class MysqlFloor extends FunctionNode
{
    public $simpleArithmeticExpression;

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'FLOOR(' . $sqlWalker->walkSimpleArithmeticExpression(
            $this->simpleArithmeticExpression
        ) . ')';
    }

    public function parse(\Doctrine\ORM\Query\Parser $parser)
    {
        $lexer = $parser->getLexer();

        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);

        $this->simpleArithmeticExpression = $parser->SimpleArithmeticExpression();

        $parser->match(Lexer::T_CLOSE_PARENTHESIS);
    }
}
```

We will register the function by calling and can then use it:

```php
<?php
$config = $em->getConfiguration();
$config->registerNumericFunction('FLOOR', 'MyProject\Query\MysqlFloor');

$dql = "SELECT FLOOR(person.salary * 1.75) FROM CompanyPerson person";
```

Querying Inherited Classes

This section demonstrates how you can query inherited classes and what type of results to expect.

Single Table

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

First we need to setup an example set of entities to use. In this scenario it is a generic Person and Employee example:

```php
<?php
namespace Entities;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    /**
     * @Id @Column(type="integer")
     * @GeneratedValue
     */
    protected $id;

    /**
     * @Column(type="string", length=50)
     */
    protected $name;

    // ...
}

/**
 * @Entity
 */
class Employee extends Person
{
    /**
     * @Column(type="string", length=50)
     */
    private $department;

    // ...
}
```

First notice that the generated SQL to create the tables for these entities looks like the following:

```
CREATE TABLE Person (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    discr VARCHAR(255) NOT NULL,
    department VARCHAR(50) NOT NULL
)
```

Now when persist a new `Employee` instance it will set the discriminator value for us automatically:

```
<?php
$employee = new \Entities\Employee();
$employee->setName('test');
$employee->setDepartment('testing');
$em->persist($employee);
$em->flush();
```

Now lets run a simple query to retrieve the `Employee` we just created:

```
SELECT e FROM Entities\Employee e WHERE e.name = 'test'
```

If we check the generated SQL you will notice it has some special conditions added to ensure that we will only get back `Employee` entities:

```
SELECT p0_.id AS id0, p0_.name AS name1, p0_.department AS department2,
       p0_.discr AS discr3 FROM Person p0_
WHERE (p0_.name = ?) AND p0_.discr IN ('employee')
```

Class Table Inheritance

Class Table Inheritance is an inheritance mapping strategy where each class in a hierarchy is mapped to several tables: its own table and the tables of all parent classes. The table of a child class is linked to the table of a parent class through a foreign key constraint. Doctrine 2 implements this strategy through the use of a discriminator column in the topmost table of the hierarchy because this is the easiest way to achieve polymorphic queries with Class Table Inheritance.

The example for class table inheritance is the same as single table, you just need to change the inheritance type from `SINGLE_TABLE` to `JOINED`:

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

Now take a look at the SQL which is generated to create the table, you'll notice some differences:

```
CREATE TABLE Person (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    discr VARCHAR(255) NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
CREATE TABLE Employee (
    id INT NOT NULL,
    department VARCHAR(50) NOT NULL,
    PRIMARY KEY(id)
) ENGINE = InnoDB;
ALTER TABLE Employee ADD FOREIGN KEY (id) REFERENCES Person(id) ON DELETE CASCADE
```

- The data is split between two tables
- A foreign key exists between the two tables

Now if were to insert the same `Employee` as we did in the `SINGLE_TABLE` example and run the same example query it will generate different SQL joining the `Person` information automatically for you:

```
SELECT p0_.id AS id0, p0_.name AS name1, e1_.department AS department2,
       p0_.discr AS discr3
FROM Employee e1_ INNER JOIN Person p0_ ON e1_.id = p0_.id
WHERE p0_.name = ?
```

The Query class

An instance of the `Doctrine\ORM\Query` class represents a DQL query. You create a Query instance be calling `EntityManager#createQuery($dql)`, passing the DQL query string. Alternatively you can create an empty `Query` instance and invoke `Query#setDql($dql)` afterwards. Here are some examples:

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

// example1: passing a DQL string
$q = $em->createQuery('select u from MyProject\Model\User u');

// example2: using setDql
$q = $em->createQuery();
$q->setDql('select u from MyProject\Model\User u');
```

Query Result Formats

The format in which the result of a DQL SELECT query is returned can be influenced by a so-called `hydration mode`. A hydration mode specifies a particular way in which a SQL result set is transformed. Each hydration mode has its own dedicated method on the Query class. Here they are:

- `Query#getResult()`: Retrieves a collection of objects. The result is either a plain collection of objects (pure) or an array where the objects are nested in the result rows (mixed).
- `Query#getSingleResult()`: Retrieves a single object. If the result contains more than one object, an `NonUniqueResultException` is thrown. If the result contains no objects, an `NoResultException` is thrown. The pure/mixed distinction does not apply.
- `Query#getOneOrNullResult()`: Retrieve a single object. If no object is found null will be returned.
- `Query#getArrayResult()`: Retrieves an array graph (a nested array) that is largely interchangeable with the object graph generated by `Query#getResult()` for read-only purposes.

  Note: An array graph can differ from the corresponding object graph in certain scenarios due to the difference of the identity semantics between arrays and objects.

- `Query#getScalarResult()`: Retrieves a flat/rectangular result set of scalar values that can contain duplicate data. The pure/mixed distinction does not apply.
- `Query#getSingleScalarResult()`: Retrieves a single scalar value from the result returned by the dbms. If the result contains more than a single scalar value, an exception is thrown. The pure/mixed distinction does not apply.

Instead of using these methods, you can alternatively use the general-purpose method `Query#execute(array $params = array(), $hydrationMode = Query::HYDRATE_OBJECT)`. Using this method you can directly supply the hydration mode as the second parameter via one of the Query constants. In fact, the methods mentioned earlier are just convenient shortcuts for the execute method. For example, the method `Query#getResult()` internally invokes execute, passing in `Query::HYDRATE_OBJECT` as the hydration mode.

The use of the methods mentioned earlier is generally preferred as it leads to more concise code.

Pure and Mixed Results

The nature of a result returned by a DQL SELECT query retrieved through `Query#getResult()` or `Query#getArrayResult()` can be of 2 forms: **pure** and **mixed**. In the previous simple examples, you already saw a "pure" query result, with only objects. By default, the result type is **pure** but **as soon as scalar values, such as aggregate values or other scalar values that do not belong to an entity, appear in the SELECT part of the DQL query, the result becomes mixed**. A mixed result has a different structure than a pure result in order to accommodate for the scalar values.

A pure result usually looks like this:

```
$dql = "SELECT u FROM User u";

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

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

```
$dql = "SELECT u, 'some scalar string', count(u.groups) AS num FROM User u JOIN u.groups g GROUP BY

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

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

```
SELECT u, UPPER(u.name) nameUpper FROM MyProject\Model\User u
```

This query makes use of the `UPPER` DQL function that returns a scalar value and because there is now a scalar value in the SELECT clause, we get a mixed result.

Conventions for mixed results are as follows:

- The object fetched in the FROM clause is always positioned with the key '0'.
- Every scalar without a name is numbered in the order given in the query, starting with 1.
- Every aliased scalar is given with its alias-name as the key. The case of the name is kept.
- If several objects are fetched from the FROM clause they alternate every row.

Here is how the result could look like:

```
array
    array
        [0] => User (Object)
        ['nameUpper'] => "ROMAN"
    array
        [0] => User (Object)
        ['nameUpper'] => "JONATHAN"
    ...
```

And here is how you would access it in PHP code:

```php
<?php
foreach ($results as $row) {
    echo "Name: " . $row[0]->getName();
    echo "Name UPPER: " . $row['nameUpper'];
}
```

Fetching Multiple FROM Entities

If you fetch multiple entities that are listed in the FROM clause then the hydration will return the rows iterating the different top-level entities.

```
$dql = "SELECT u, g FROM User u, Group g";

array
    [0] => Object (User)
    [1] => Object (Group)
    [2] => Object (User)
    [3] => Object (Group)
```

Hydration Modes

Each of the Hydration Modes makes assumptions about how the result is returned to user land. You should know about all the details to make best use of the different result formats:

The constants for the different hydration modes are:

- Query::HYDRATE_OBJECT
- Query::HYDRATE_ARRAY
- Query::HYDRATE_SCALAR
- Query::HYDRATE_SINGLE_SCALAR

Object Hydration Object hydration hydrates the result set into the object graph:

```php
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_OBJECT);
```

Array Hydration You can run the same query with array hydration and the result set is hydrated into an array that represents the object graph:

```php
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_ARRAY);
```

You can use the `getArrayResult()` shortcut as well:

```php
<?php
$users = $query->getArrayResult();
```

Scalar Hydration If you want to return a flat rectangular result set instead of an object graph you can use scalar hydration:

```php
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$users = $query->getResult(Query::HYDRATE_SCALAR);
echo $users[0]['u_id'];
```

The following assumptions are made about selected fields using Scalar Hydration:

1. Fields from classes are prefixed by the DQL alias in the result. A query of the kind 'SELECT u.name ..' returns a key 'u_name' in the result rows.

Single Scalar Hydration If you have a query which returns just a single scalar value you can use single scalar hydration:

```
<?php
$query = $em->createQuery('SELECT COUNT(a.id) FROM CmsUser u LEFT JOIN u.articles a WHERE u.username
$query->setParameter(1, 'jwage');
$numArticles = $query->getResult(Query::HYDRATE_SINGLE_SCALAR);
```

You can use the `getSingleScalarResult()` shortcut as well:

```
<?php
$numArticles = $query->getSingleScalarResult();
```

Custom Hydration Modes You can easily add your own custom hydration modes by first creating a class which extends `AbstractHydrator`:

```
<?php
namespace MyProject\Hydrators;

use Doctrine\ORM\Internal\Hydration\AbstractHydrator;

class CustomHydrator extends AbstractHydrator
{
    protected function _hydrateAll()
    {
        return $this->_stmt->fetchAll(PDO::FETCH_ASSOC);
    }
}
```

Next you just need to add the class to the ORM configuration:

```
<?php
$em->getConfiguration()->addCustomHydrationMode('CustomHydrator', 'MyProject\Hydrators\CustomHydrato
```

Now the hydrator is ready to be used in your queries:

```
<?php
$query = $em->createQuery('SELECT u FROM CmsUser u');
$results = $query->getResult('CustomHydrator');
```

Iterating Large Result Sets

There are situations when a query you want to execute returns a very large result-set that needs to be processed. All the previously described hydration modes completely load a result-set into memory which might not be feasible with large result sets. See the Batch Processing section on details how to iterate large result sets.

Functions

The following methods exist on the `AbstractQuery` which both `Query` and `NativeQuery` extend from.

Parameters Prepared Statements that use numerical or named wildcards require additional parameters to be executable against the database. To pass parameters to the query the following methods can be used:

- `AbstractQuery::setParameter($param, $value)` - Set the numerical or named wildcard to the given value.
- `AbstractQuery::setParameters(array $params)` - Set an array of parameter key-value pairs.
- `AbstractQuery::getParameter($param)`
- `AbstractQuery::getParameters()`

Both named and positional parameters are passed to these methods without their ? or : prefix.

Cache related API You can cache query results based either on all variables that define the result (SQL, Hydration Mode, Parameters and Hints) or on user-defined cache keys. However by default query results are not cached at all. You have to enable the result cache on a per query basis. The following example shows a complete workflow using the Result Cache API:

```php
<?php
$query = $em->createQuery('SELECT u FROM MyProject\Model\User u WHERE u.id = ?1');
$query->setParameter(1, 12);

$query->setResultCacheDriver(new ApcCache());

$query->useResultCache(true)
      ->setResultCacheLifeTime($seconds = 3600);

$result = $query->getResult(); // cache miss

$query->expireResultCache(true);
$result = $query->getResult(); // forced expire, cache miss

$query->setResultCacheId('my_query_result');
$result = $query->getResult(); // saved in given result cache id.

// or call useResultCache() with all parameters:
$query->useResultCache(true, $seconds = 3600, 'my_query_result');
$result = $query->getResult(); // cache hit!

// Introspection
$queryCacheProfile = $query->getQueryCacheProfile();
$cacheDriver = $query->getResultCacheDriver();
$lifetime = $query->getLifetime();
$key = $query->getCacheKey();
```

Note: You can set the Result Cache Driver globally on the `Doctrine\ORM\Configuration` instance so that it is passed to every `Query` and `NativeQuery` instance.

Query Hints You can pass hints to the query parser and hydrators by using the `AbstractQuery::setHint($name, $value)` method. Currently there exist mostly internal query hints that are not be consumed in userland. However the following few hints are to be used in userland:

- Query::HINT_FORCE_PARTIAL_LOAD - Allows to hydrate objects although not all their columns are fetched. This query hint can be used to handle memory consumption problems with large result-sets that contain char or binary data. Doctrine has no way of implicitly reloading this data. Partially loaded objects have to be passed to `EntityManager::refresh()` if they are to be reloaded fully from the database.

- Query::HINT_REFRESH - This query is used internally by `EntityManager::refresh()` and can be used in userland as well. If you specify this hint and a query returns the data for an entity that is already managed by the UnitOfWork, the fields of the existing entity will be refreshed. In normal operation a result-set that loads data of an already existing entity is discarded in favor of the already existing entity.
- Query::HINT_CUSTOM_TREE_WALKERS - An array of additional `Doctrine\ORM\Query\TreeWalker` instances that are attached to the DQL query parsing process.

Query Cache (DQL Query Only) Parsing a DQL query and converting it into a SQL query against the underlying database platform obviously has some overhead in contrast to directly executing Native SQL queries. That is why there is a dedicated Query Cache for caching the DQL parser results. In combination with the use of wildcards you can reduce the number of parsed queries in production to zero.

The Query Cache Driver is passed from the `Doctrine\ORM\Configuration` instance to each `Doctrine\ORM\Query` instance by default and is also enabled by default. This also means you don't regularly need to fiddle with the parameters of the Query Cache, however if you do there are several methods to interact with it:

- `Query::setQueryCacheDriver($driver)` - Allows to set a Cache instance
- `Query::setQueryCacheLifeTime($seconds = 3600)` - Set lifetime of the query caching.
- `Query::expireQueryCache($bool)` - Enforce the expiring of the query cache if set to true.
- `Query::getExpireQueryCache()`
- `Query::getQueryCacheDriver()`
- `Query::getQueryCacheLifeTime()`

First and Max Result Items (DQL Query Only) You can limit the number of results returned from a DQL query as well as specify the starting offset, Doctrine then uses a strategy of manipulating the select query to return only the requested number of results:

- `Query::setMaxResults($maxResults)`
- `Query::setFirstResult($offset)`

Note: If your query contains a fetch-joined collection specifying the result limit methods are not working as you would expect. Set Max Results restricts the number of database result rows, however in the case of fetch-joined collections one root entity might appear in many rows, effectively hydrating less than the specified number of results.

Temporarily change fetch mode in DQL While normally all your associations are marked as lazy or extra lazy you will have cases where you are using DQL and don't want to fetch join a second, third or fourth level of entities into your result, because of the increased cost of the SQL JOIN. You can mark a many-to-one or one-to-one association as fetched temporarily to batch fetch these entities using a WHERE .. IN query.

```php
<?php
$query = $em->createQuery("SELECT u FROM MyProject\User u");
$query->setFetchMode("MyProject\User", "address", \Doctrine\ORM\Mapping\ClassMetadata::FETCH_EAGER);
$query->execute();
```

Given that there are 10 users and corresponding addresses in the database the executed queries will look something like:

```sql
SELECT * FROM users;
SELECT * FROM address WHERE id IN (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Note: Changing the fetch mode during a query is only possible for one-to-one and many-to-one relations.

EBNF

The following context-free grammar, written in an EBNF variant, describes the Doctrine Query Language. You can consult this grammar whenever you are unsure about what is possible with DQL or what the correct syntax for a particular query should be.

Document syntax:

- non-terminals begin with an upper case character
- terminals begin with a lower case character
- parentheses (...) are used for grouping
- square brackets [...] are used for defining an optional part, e.g. zero or one time
- curly brackets {...} are used for repetition, e.g. zero or more times
- double quotation marks "..." define a terminal string a vertical bar | represents an alternative

Terminals

- identifier (name, email, ...)
- string ('foo', 'bar''s house', '%ninja%', ...)
- char ('/', '\', ' ', ...)
- integer (-1, 0, 1, 34, ...)
- float (-0.23, 0.007, 1.245342E+8, ...)
- boolean (false, true)

Query Language

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

Statements

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

Identifiers

```
/* Alias Identification usage (the "u" of "u.name") */
IdentificationVariable ::= identifier

/* Alias Identification declaration (the "u" of "FROM User u") */
AliasIdentificationVariable :: = identifier

/* identifier that must be a class name (the "User" of "FROM User u") */
AbstractSchemaName ::= identifier

/* Alias ResultVariable declaration (the "total" of "COUNT(*) AS total") */
AliasResultVariable = identifier

/* ResultVariable identifier usage of mapped field aliases (the "total" of "COUNT(*) AS total") */
ResultVariable = identifier

/* identifier that must be a field (the "name" of "u.name") */
/* This is responsible to know if the field exists in Object, no matter if it's a relation or a simpl
FieldIdentificationVariable ::= identifier

/* identifier that must be a collection-valued association field (to-many) (the "Phonenumbers" of "u
CollectionValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be a single-valued association field (to-one) (the "Group" of "u.Group") */
SingleValuedAssociationField ::= FieldIdentificationVariable

/* identifier that must be an embedded class state field */
EmbeddedClassStateField ::= FieldIdentificationVariable

/* identifier that must be a simple state field (name, email, ...) (the "name" of "u.name") */
/* The difference between this and FieldIdentificationVariable is only semantical, because it points
SimpleStateField ::= FieldIdentificationVariable
```

Path Expressions

```
/* "u.Group" or "u.Phonenumbers" declarations */
JoinAssociationPathExpression             ::= IdentificationVariable "." (CollectionValuedAssociatio

/* "u.Group" or "u.Phonenumbers" usages */
AssociationPathExpression                 ::= CollectionValuedPathExpression | SingleValuedAssociati

/* "u.name" or "u.Group" */
SingleValuedPathExpression                ::= StateFieldPathExpression | SingleValuedAssociationPath

/* "u.name" or "u.Group.name" */
StateFieldPathExpression                  ::= IdentificationVariable "." StateField

/* "u.Group" */
SingleValuedAssociationPathExpression     ::= IdentificationVariable "." SingleValuedAssociationFiel

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

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

Clauses

```
SelectClause        ::= "SELECT" ["DISTINCT"] SelectExpression {"," SelectExpression}*
SimpleSelectClause  ::= "SELECT" ["DISTINCT"] SimpleSelectExpression
UpdateClause        ::= "UPDATE" AbstractSchemaName ["AS"] AliasIdentificationVariable "SET" UpdateIt
DeleteClause        ::= "DELETE" ["FROM"] AbstractSchemaName ["AS"] AliasIdentificationVariable
FromClause          ::= "FROM" IdentificationVariableDeclaration {"," IdentificationVariableDeclarat
SubselectFromClause ::= "FROM" SubselectIdentificationVariableDeclaration {"," SubselectIdentificatio
WhereClause         ::= "WHERE" ConditionalExpression
HavingClause        ::= "HAVING" ConditionalExpression
GroupByClause       ::= "GROUP" "BY" GroupByItem {"," GroupByItem}*
OrderByClause       ::= "ORDER" "BY" OrderByItem {"," OrderByItem}*
Subselect           ::= SimpleSelectClause SubselectFromClause [WhereClause] [GroupByClause] [Having
```

Items

```
UpdateItem  ::= SingleValuedPathExpression "=" NewValue
OrderByItem ::= (SimpleArithmeticExpression | SingleValuedPathExpression | ScalarExpression | ResultV
GroupByItem ::= IdentificationVariable | ResultVariable | SingleValuedPathExpression
NewValue    ::= SimpleArithmeticExpression | "NULL"
```

From, Join and Index by

```
IdentificationVariableDeclaration          ::= RangeVariableDeclaration [IndexBy] {Join}*
SubselectIdentificationVariableDeclaration ::= IdentificationVariableDeclaration
RangeVariableDeclaration                   ::= AbstractSchemaName ["AS"] AliasIdentificationVariable
JoinAssociationDeclaration                 ::= JoinAssociationPathExpression ["AS"] AliasIdentificati
Join                                       ::= ["LEFT" ["OUTER"] | "INNER"] "JOIN" (JoinAssociationDe
IndexBy                                    ::= "INDEX" "BY" StateFieldPathExpression
```

Select Expressions

```
SelectExpression        ::= (IdentificationVariable | ScalarExpression | AggregateExpression | Functi
SimpleSelectExpression  ::= (StateFieldPathExpression | IdentificationVariable | FunctionDeclaration
PartialObjectExpression ::= "PARTIAL" IdentificationVariable "." PartialFieldSet
PartialFieldSet         ::= "{" SimpleStateField {"," SimpleStateField}* "}"
NewObjectExpression     ::= "NEW" IdentificationVariable "(" NewObjectArg {"," NewObjectArg}* ")"
NewObjectArg            ::= ScalarExpression | "(" Subselect ")"
```

Conditional Expressions

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

Collection Expressions

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

Literal Values

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

Input Parameter

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

Arithmetic Expressions

```
ArithmeticExpression       ::= SimpleArithmeticExpression | "(" Subselect ")"
SimpleArithmeticExpression ::= ArithmeticTerm {("+" | "-") ArithmeticTerm}*
ArithmeticTerm             ::= ArithmeticFactor {("*" | "/") ArithmeticFactor}*
ArithmeticFactor           ::= [("+" | "-")] ArithmeticPrimary
ArithmeticPrimary          ::= SingleValuedPathExpression | Literal | "(" SimpleArithmeticExpression
                               | FunctionsReturningNumerics | AggregateExpression | FunctionsReturnin
                               | FunctionsReturningDatetime | IdentificationVariable | ResultVariable
                               | InputParameter | CaseExpression
```

Scalar and Type Expressions

```
ScalarExpression       ::= SimpleArithmeticExpression | StringPrimary | DateTimePrimary | StateFieldP
StringExpression       ::= StringPrimary | ResultVariable | "(" Subselect ")"
StringPrimary          ::= StateFieldPathExpression | string | InputParameter | FunctionsReturningStr
BooleanExpression      ::= BooleanPrimary | "(" Subselect ")"
BooleanPrimary         ::= StateFieldPathExpression | boolean | InputParameter
EntityExpression       ::= SingleValuedAssociationPathExpression | SimpleEntityExpression
SimpleEntityExpression ::= IdentificationVariable | InputParameter
DatetimeExpression     ::= DatetimePrimary | "(" Subselect ")"
DatetimePrimary        ::= StateFieldPathExpression | InputParameter | FunctionsReturningDatetime | A
```

Note: Parts of CASE expressions are not yet implemented.

Aggregate Expressions

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

Case Expressions

```
CaseExpression        ::= GeneralCaseExpression | SimpleCaseExpression | CoalesceExpression | NullifE
GeneralCaseExpression ::= "CASE" WhenClause {WhenClause}* "ELSE" ScalarExpression "END"
WhenClause            ::= "WHEN" ConditionalExpression "THEN" ScalarExpression
SimpleCaseExpression  ::= "CASE" CaseOperand SimpleWhenClause {SimpleWhenClause}* "ELSE" ScalarExpres
CaseOperand           ::= StateFieldPathExpression | TypeDiscriminator
SimpleWhenClause      ::= "WHEN" ScalarExpression "THEN" ScalarExpression
CoalesceExpression    ::= "COALESCE" "(" ScalarExpression {"," ScalarExpression}* ")"
NullifExpression      ::= "NULLIF" "(" ScalarExpression "," ScalarExpression ")"
```

Other Expressions

QUANTIFIED/BETWEEN/COMPARISON/LIKE/NULL/EXISTS

```
QuantifiedExpression     ::= ("ALL" | "ANY" | "SOME") "(" Subselect ")"
BetweenExpression        ::= ArithmeticExpression ["NOT"] "BETWEEN" ArithmeticExpression "AND" Arithm
ComparisonExpression     ::= ArithmeticExpression ComparisonOperator ( QuantifiedExpression | Arithme
InExpression             ::= SingleValuedPathExpression ["NOT"] "IN" "(" (InParameter {"," InParamete
InstanceOfExpression     ::= IdentificationVariable ["NOT"] "INSTANCE" ["OF"] (InstanceOfParameter |
InstanceOfParameter      ::= AbstractSchemaName | InputParameter
LikeExpression           ::= StringExpression ["NOT"] "LIKE" StringPrimary ["ESCAPE" char]
NullComparisonExpression ::= (InputParameter | NullIfExpression | CoalesceExpression | AggregateExpre
ExistsExpression         ::= ["NOT"] "EXISTS" "(" Subselect ")"
ComparisonOperator       ::= "=" | "<" | "<=" | "<>" | ">" | ">=" | "!="
```

Functions

```
FunctionDeclaration ::= FunctionsReturningStrings | FunctionsReturningNumerics | FunctionsReturningDa

FunctionsReturningNumerics ::=
        "LENGTH" "(" StringPrimary ")" |
        "LOCATE" "(" StringPrimary "," StringPrimary ["," SimpleArithmeticExpression]")" |
        "ABS" "(" SimpleArithmeticExpression ")" |
        "SQRT" "(" SimpleArithmeticExpression ")" |
        "MOD" "(" SimpleArithmeticExpression "," SimpleArithmeticExpression ")" |
        "SIZE" "(" CollectionValuedPathExpression ")" |
        "DATE_DIFF" "(" ArithmeticPrimary "," ArithmeticPrimary ")" |
        "BIT_AND" "(" ArithmeticPrimary "," ArithmeticPrimary ")" |
        "BIT_OR" "(" ArithmeticPrimary "," ArithmeticPrimary ")"

FunctionsReturningDateTime ::=
        "CURRENT_DATE" |
        "CURRENT_TIME" |
        "CURRENT_TIMESTAMP" |
        "DATE_ADD" "(" ArithmeticPrimary "," ArithmeticPrimary "," StringPrimary ")" |
        "DATE_SUB" "(" ArithmeticPrimary "," ArithmeticPrimary "," StringPrimary ")"

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

9.2.16 The QueryBuilder

A `QueryBuilder` provides an API that is designed for conditionally constructing a DQL query in several steps.

It provides a set of classes and methods that is able to programmatically build queries, and also provides a fluent API. This means that you can change between one methodology to the other as you want, 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 have 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 to increase performance. Any changes that may affect the generated DQL actually modifies the state of `QueryBuilder` to a stage we call STATE_DIRTY. One `QueryBuilder` can be in two different states:

- `QueryBuilder::STATE_CLEAN`, which means DQL haven't been altered since last retrieval or nothing were added since its instantiation
- `QueryBuilder::STATE_DIRTY`, means DQL query must (and will) be processed on next retrieval

Working with QueryBuilder

High level API methods

To simplify even more the way you build a query in Doctrine, we can take advantage of what we call Helper methods. For all base code, there is a set of useful methods to simplify a programmer's life. To illustrate how to work with them, here is the same example 6 re-written using `QueryBuilder` helper methods:

```php
<?php
// $qb instanceof QueryBuilder

$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name', 'ASC');
```

`QueryBuilder` helper methods are considered the standard way to 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
<?php
// $qb instanceof QueryBuilder

$qb->select(array('u')) // string 'u' is converted to array internally
   ->from('User', 'u')
   ->where($qb->expr()->orX(
       $qb->expr()->eq('u.id', '?1'),
       $qb->expr()->like('u.nickname', '?2')
   ))
   ->orderBy('u.surname', 'ASC'));
```

Here is a complete list of helper methods available in `QueryBuilder`:

```php
<?php
class QueryBuilder
{
    // Example - $qb->select('u')
    // Example - $qb->select(array('u', 'p'))
    // Example - $qb->select($qb->expr()->select('u', 'p'))
    public function select($select = null);

    // addSelect does not override previous calls to select
    //
    // Example - $qb->select('u');
    //                ->addSelect('p.area_code');
    public function addSelect($select = null);

    // Example - $qb->delete('User', 'u')
    public function delete($delete = null, $alias = null);

    // Example - $qb->update('Group', 'g')
    public function update($update = null, $alias = null);

    // Example - $qb->set('u.firstName', $qb->expr()->literal('Arnold'))
    // Example - $qb->set('u.numChilds', 'u.numChilds + ?1')
    // Example - $qb->set('u.numChilds', $qb->expr()->sum('u.numChilds', '?1'))
    public function set($key, $value);
```

```
// Example - $qb->from('Phonenumber', 'p')
// Example - $qb->from('Phonenumber', 'p', 'p.id')
public function from($from, $alias, $indexBy = null);

// Example - $qb->join('u.Group', 'g', Expr\Join::WITH, $qb->expr()->eq('u.status_id', '?1'))
// Example - $qb->join('u.Group', 'g', 'WITH', 'u.status = ?1')
// Example - $qb->join('u.Group', 'g', 'WITH', 'u.status = ?1', 'g.id')
public function join($join, $alias, $conditionType = null, $condition = null, $indexBy = null);

// Example - $qb->innerJoin('u.Group', 'g', Expr\Join::WITH, $qb->expr()->eq('u.status_id', '?1',
// Example - $qb->innerJoin('u.Group', 'g', 'WITH', 'u.status = ?1')
// Example - $qb->innerJoin('u.Group', 'g', 'WITH', 'u.status = ?1', 'g.id')
public function innerJoin($join, $alias, $conditionType = null, $condition = null, $indexBy = nu

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

// 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(
// Example - $qb->where('u.firstName = ?1 AND u.surname = ?2')
public function where($where);

// NOTE: ->andWhere() can be used directly, without any ->where() before
//
// Example - $qb->andWhere($qb->expr()->orX($qb->expr()->lte('u.age', 40), 'u.numChild = 0'))
public function andWhere($where);

// Example - $qb->orWhere($qb->expr()->between('u.id', 1, 10));
public function orWhere($where);

// NOTE: -> groupBy() overrides all previously set grouping conditions
//
// Example - $qb->groupBy('u.id')
public function groupBy($groupBy);

// Example - $qb->addGroupBy('g.name')
public function addGroupBy($groupBy);

// NOTE: -> having() overrides all previously set having conditions
//
// Example - $qb->having('u.salary >= ?1')
// Example - $qb->having($qb->expr()->gte('u.salary', '?1'))
public function having($having);

// Example - $qb->andHaving($qb->expr()->gt($qb->expr()->count('u.numChild'), 0))
public function andHaving($having);

// Example - $qb->orHaving($qb->expr()->lte('g.managerLevel', '100'))
public function orHaving($having);

// NOTE: -> orderBy() overrides all previously set ordering conditions
//
// Example - $qb->orderBy('u.surname', 'DESC')
public function orderBy($sort, $order = null);
```

```
    // Example - $qb->addOrderBy('u.firstName')
    public function addOrderBy($sort, $order = null); // Default $order = 'ASC'
}
```

Binding parameters to your query

Doctrine supports dynamic binding of parameters to your query, similar to preparing queries. You can use both strings and numbers as placeholders, although both have a slightly different syntax. Additionally, you must make your choice: Mixing both styles is not allowed. Binding parameters can simply be achieved as follows:

```
<?php
// $qb instanceof QueryBuilder

$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = ?1')
   ->orderBy('u.name', 'ASC')
   ->setParameter(1, 100); // Sets ?1 to 100, and thus we will fetch a user with u.id = 100
```

You are not forced to enumerate your placeholders as the alternative syntax is available:

```
<?php
// $qb instanceof QueryBuilder

$qb->select('u')
   ->from('User', 'u')
   ->where('u.id = :identifier')
   ->orderBy('u.name', 'ASC')
   ->setParameter('identifier', 100); // Sets :identifier to 100, and thus we will fetch a user with
```

Note that numeric placeholders start with a ? followed by a number while the named placeholders start with a : followed by a string.

Calling `setParameter()` automatically infers which type you are setting as value. This works for integers, arrays of strings/integers, DateTime instances and for managed entities. If you want to set a type explicitly you can call the third argument to `setParameter()` explicitly. It accepts either a PDO type or a DBAL Type name for conversion.

If you've got several parameters to bind to your query, you can also use setParameters() instead of setParameter() with the following syntax:

```
<?php
// $qb instanceof QueryBuilder

// Query here...
$qb->setParameters(array(1 => 'value for ?1', 2 => 'value for ?2'));
```

Getting already bound parameters is easy - simply use the above mentioned syntax with "getParameter()" or "getParameters()":

```
<?php
// $qb instanceof QueryBuilder

// See example above
$params = $qb->getParameters();
// $params instanceof \Doctrine\Common\Collections\ArrayCollection

// Equivalent to
```

```
$param = $qb->getParameter(1);
// $param instanceof \Doctrine\ORM\Query\Parameter
```

Note: If you try to get a parameter that was not bound yet, getParameter() simply returns NULL.

The API of a Query Parameter is:

```
namespace Doctrine\ORM\Query;

class Parameter
{
    public function getName();
    public function getValue();
    public function getType();
    public function setValue($value, $type = null);
}
```

Limiting the Result

To limit a result the query builder has some methods in common with the Query object which can be retrieved from `EntityManager#createQuery()`.

```
<?php
// $qb instanceof QueryBuilder
$offset = (int)$_GET['offset'];
$limit = (int)$_GET['limit'];

$qb->add('select', 'u')
   ->add('from', 'User u')
   ->add('orderBy', 'u.name ASC')
   ->setFirstResult( $offset )
   ->setMaxResults( $limit );
```

Executing a Query

The QueryBuilder is a builder object only, it has no means of actually executing the Query. Additionally a set of parameters such as query hints cannot be set on the QueryBuilder itself. This is why you always have to convert a querybuilder instance into a Query object:

```
<?php
// $qb instanceof QueryBuilder
$query = $qb->getQuery();

// Set additional Query options
$query->setQueryHint('foo', 'bar');
$query->useResultCache('my_cache_id');

// Execute Query
$result = $query->getResult();
$single = $query->getSingleResult();
$array = $query->getArrayResult();
$scalar = $query->getScalarResult();
$singleScalar = $query->getSingleScalarResult();
```

The Expr class

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

```php
<?php
// $qb instanceof QueryBuilder

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

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

```php
<?php
class Expr
{
    /** Conditional objects **/

    // Example - $qb->expr()->andX($cond1 [, $condN])->add(...)->...
    public function andX($x = null); // Returns Expr\AndX instance

    // Example - $qb->expr()->orX($cond1 [, $condN])->add(...)->...
    public function orX($x = null); // Returns Expr\OrX instance


    /** Comparison objects **/

    // Example - $qb->expr()->eq('u.id', '?1') => u.id = ?1
    public function eq($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->neq('u.id', '?1') => u.id <> ?1
    public function neq($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->lt('u.id', '?1') => u.id < ?1
    public function lt($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->lte('u.id', '?1') => u.id <= ?1
    public function lte($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->gt('u.id', '?1') => u.id > ?1
    public function gt($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->gte('u.id', '?1') => u.id >= ?1
    public function gte($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->isNull('u.id') => u.id IS NULL
    public function isNull($x); // Returns string

    // Example - $qb->expr()->isNotNull('u.id') => u.id IS NOT NULL
    public function isNotNull($x); // Returns string


    /** Arithmetic objects **/

    // Example - $qb->expr()->prod('u.id', '2') => u.id * 2
    public function prod($x, $y); // Returns Expr\Math instance

    // Example - $qb->expr()->diff('u.id', '2') => u.id - 2
    public function diff($x, $y); // Returns Expr\Math instance

    // Example - $qb->expr()->sum('u.id', '2') => u.id + 2
    public function sum($x, $y); // Returns Expr\Math instance

    // Example - $qb->expr()->quot('u.id', '2') => u.id / 2
    public function quot($x, $y); // Returns Expr\Math instance


    /** Pseudo-function objects **/

    // Example - $qb->expr()->exists($qb2->getDql())
    public function exists($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->all($qb2->getDql())
    public function all($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->some($qb2->getDql())
    public function some($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->any($qb2->getDql())
    public function any($subquery); // Returns Expr\Func instance

    // Example - $qb->expr()->not($qb->expr()->eq('u.id', '?1'))
    public function not($restriction); // Returns Expr\Func instance

    // Example - $qb->expr()->in('u.id', array(1, 2, 3))
    // Make sure that you do NOT use something similar to $qb->expr()->in('value', array('stringvalue
    // Instead, use $qb->expr()->in('value', array('?1')) and bind your parameter to ?1 (see section
    public function in($x, $y); // Returns Expr\Func instance

    // Example - $qb->expr()->notIn('u.id', '2')
    public function notIn($x, $y); // Returns Expr\Func instance

    // Example - $qb->expr()->like('u.firstname', $qb->expr()->literal('Gui%'))
    public function like($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->notLike('u.firstname', $qb->expr()->literal('Gui%'))
    public function notLike($x, $y); // Returns Expr\Comparison instance

    // Example - $qb->expr()->between('u.id', '1', '10')
    public function between($val, $x, $y); // Returns Expr\Func


    /** Function objects **/

    // Example - $qb->expr()->trim('u.firstname')
    public function trim($x); // Returns Expr\Func
    // Example - $qb->expr()->concat('u.firstname', $qb->expr()->concat($qb->expr()->literal(' '), '
    public function concat($x, $y); // Returns Expr\Func

    // Example - $qb->expr()->substring('u.firstname', 0, 1)
    public function substring($x, $from, $len); // Returns Expr\Func

    // Example - $qb->expr()->lower('u.firstname')
    public function lower($x); // Returns Expr\Func

    // Example - $qb->expr()->upper('u.firstname')
    public function upper($x); // Returns Expr\Func

    // Example - $qb->expr()->length('u.firstname')
    public function length($x); // Returns Expr\Func

    // Example - $qb->expr()->avg('u.age')
    public function avg($x); // Returns Expr\Func

    // Example - $qb->expr()->max('u.age')
    public function max($x); // Returns Expr\Func

    // Example - $qb->expr()->min('u.age')
    public function min($x); // Returns Expr\Func

    // Example - $qb->expr()->abs('u.currentBalance')
    public function abs($x); // Returns Expr\Func

    // Example - $qb->expr()->sqrt('u.currentBalance')
    public function sqrt($x); // Returns Expr\Func

    // Example - $qb->expr()->count('u.firstname')
    public function count($x); // Returns Expr\Func

    // Example - $qb->expr()->countDistinct('u.surname')
    public function countDistinct($x); // Returns Expr\Func
}
```

Low Level API

Now we have describe the low level (thought of as the hardcore method) of creating queries. It may be useful to work at this level for optimization purposes, but most of the time it is preferred to work at a higher level of abstraction.

All helper methods in `QueryBuilder` actually rely on a single one: `add()`. This method is responsible of building every piece of DQL. It takes 3 parameters: `$dqlPartName`, `$dqlPart` and `$append` (default=false)

- `$dqlPartName`: Where the `$dqlPart` should be placed. Possible values: select, from, where, groupBy, having, orderBy
- `$dqlPart`: What should be placed in `$dqlPartName`. Accepts a string or any instance of `Doctrine\ORM\Query\Expr\*`
- `$append`: Optional flag (default=false) if the `$dqlPart` should override all previously defined items in `$dqlPartName` or not (no effect on the `where` and `having` DQL query parts, which always override all previously defined items)
-

```php
<?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.

9.2.17 Native SQL

With `NativeQuery` you can execute native SELECT SQL statements and map the results to Doctrine entities or any other result format supported by Doctrine.

In order to make this mapping possible, you need to describe to Doctrine what columns in the result map to which entity property. This description is represented by a `ResultSetMapping` object.

With this feature you can map arbitrary SQL code to objects, such as highly vendor-optimized SQL or stored-procedures.

Writing `ResultSetMapping` from scratch is complex, but there is a convenience wrapper around it called a `ResultSetMappingBuilder`. It can generate the mappings for you based on Entities and even generates the `SELECT` clause based on this information for you.

Note: If you want to execute DELETE, UPDATE or INSERT statements the Native SQL API cannot be used and will probably throw errors. Use `EntityManager#getConnection()` to access the native database connection and call the `executeUpdate()` method for these queries.

The NativeQuery class

To create a `NativeQuery` you use the method `EntityManager#createNativeQuery($sql, $resultSetMapping)`. As you can see in the signature of this method, it expects 2 ingredients: The SQL you want to execute and the `ResultSetMapping` that describes how the results will be mapped.

Once you obtained an instance of a `NativeQuery`, you can bind parameters to it with the same API that `Query` has and execute it.

```
<?php
use Doctrine\ORM\Query\ResultSetMapping;

$rsm = new ResultSetMapping();
// build rsm here

$query = $entityManager->createNativeQuery('SELECT id, name, discr FROM users WHERE name = ?', $rsm);
$query->setParameter(1, 'romanb');

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

ResultSetMappingBuilder

An easy start into ResultSet mapping is the `ResultSetMappingBuilder` object. This has several benefits:

- The builder takes care of automatically updating your `ResultSetMapping` when the fields or associations change on the metadata of an entity.
- You can generate the required `SELECT` expression for a builder by converting it to a string.
- The API is much simpler than the usual `ResultSetMapping` API.

One downside is that the builder API does not yet support entities with inheritance hierachies.

```
<?php

use Doctrine\ORM\Query\ResultSetMappingBuilder;

$sql = "SELECT u.id, u.name, a.id AS address_id, a.street, a.city " .
       "FROM users u INNER JOIN address a ON u.address_id = a.id";

$rsm = new ResultSetMappingBuilder($entityManager);
$rsm->addRootEntityFromClassMetadata('MyProject\User', 'u');
$rsm->addJoinedEntityFromClassMetadata('MyProject\Address', 'a', 'u', 'address', array('id' => 'addre
```

The builder extends the `ResultSetMapping` class and as such has all the functionality of it as well.

New in version 2.4.

Starting with Doctrine ORM 2.4 you can generate the `SELECT` clause from a `ResultSetMappingBuilder`. You can either cast the builder object to `(string)` and the DQL aliases are used as SQL table aliases or use the `generateSelectClause($tableAliases)` method and pass a mapping from DQL alias (key) to SQL alias (value)

```
<?php

$selectClause = $builder->generateSelectClause(array(
    'u' => 't1',
    'g' => 't2'
));
$sql = "SELECT " . $selectClause . " FROM users t1 JOIN groups t2 ON t1.group_id = t2.id";
```

The ResultSetMapping

Understanding the `ResultSetMapping` is the key to using a `NativeQuery`. A Doctrine result can contain the following components:

- Entity results. These represent root result elements.
- Joined entity results. These represent joined entities in associations of root entity results.
- Field results. These represent a column in the result set that maps to a field of an entity. A field result always belongs to an entity result or joined entity result.
- Scalar results. These represent scalar values in the result set that will appear in each result row. Adding scalar results to a ResultSetMapping can also cause the overall result to become **mixed** (see DQL - Doctrine Query Language) if the same ResultSetMapping also contains entity results.
- Meta results. These represent columns that contain meta-information, such as foreign keys and discriminator columns. When querying for objects (`getResult()`), all meta columns of root entities or joined entities must be present in the SQL query and mapped accordingly using `ResultSetMapping#addMetaResult`.

Note: It might not surprise you that Doctrine uses `ResultSetMapping` internally when you create DQL queries. As the query gets parsed and transformed to SQL, Doctrine fills a `ResultSetMapping` that describes how the results should be processed by the hydration routines.

We will now look at each of the result types that can appear in a ResultSetMapping in detail.

Entity results

An entity result describes an entity type that appears as a root element in the transformed result. You add an entity result through `ResultSetMapping#addEntityResult()`. Let's take a look at the method signature in detail:

```php
<?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
<?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 resul
 * @param object $relation The association field that connects the parent entity result with the joi
 */
public function addJoinedEntityResult($class, $alias, $parentAlias, $relation)
```

The first parameter is the class name of the joined entity. The second parameter is an arbitrary alias for the joined entity that must be unique within the `ResultSetMapping`. You use this alias to attach field results to the entity result. The third parameter is the alias of the entity result that is the parent type of the joined relationship. The fourth and last parameter is the name of the field on the parent entity result that should contain the joined entity result.

Field results

A field result describes the mapping of a single column in a SQL result set to a field in an entity. As such, field results are inherently bound to entity results. You add a field result through `ResultSetMapping#addFieldResult()`. Again, let's examine the method signature in detail:

```
<?php
/**
 * Adds a field result that is part of an entity result or joined entity result.
 *
 * @param string $alias The alias of the entity result or joined entity result.
 * @param string $columnName The name of the column in the SQL result set.
 * @param string $fieldName The name of the field on the (joined) entity.
 */
public function addFieldResult($alias, $columnName, $fieldName)
```

The first parameter is the alias of the entity result to which the field result will belong. The second parameter is the name of the column in the SQL result set. Note that this name is case sensitive, i.e. if you use a native query against Oracle it must be all uppercase. The third parameter is the name of the field on the entity result identified by `$alias` into which the value of the column should be set.

Scalar results

A scalar result describes the mapping of a single column in a SQL result set to a scalar value in the Doctrine result. Scalar results are typically used for aggregate values but any column in the SQL result set can be mapped as a scalar value. To add a scalar result use `ResultSetMapping#addScalarResult()`. The method signature in detail:

```
<?php
/**
 * Adds a scalar result mapping.
 *
 * @param string $columnName The name of the column in the SQL result set.
 * @param string $alias The result alias with which the scalar result should be placed in the result
 */
public function addScalarResult($columnName, $alias)
```

The first parameter is the name of the column in the SQL result set and the second parameter is the result alias under which the value of the column will be placed in the transformed Doctrine result.

Meta results

A meta result describes a single column in a SQL result set that is either a foreign key or a discriminator column. These columns are essential for Doctrine to properly construct objects out of SQL result sets. To add a column as a meta result use `ResultSetMapping#addMetaResult()`. The method signature in detail:

```php
<?php
/**
 * Adds a meta column (foreign key or discriminator column) to the result set.
 *
 * @param string  $alias
 * @param string  $columnAlias
 * @param string  $columnName
 * @param boolean $isIdentifierColumn
 */
public function addMetaResult($alias, $columnAlias, $columnName, $isIdentifierColumn = false)
```

The first parameter is the alias of the entity result to which the meta column belongs. A meta result column (foreign key or discriminator column) always belongs to an entity result. The second parameter is the column alias/name of the column in the SQL result set and the third parameter is the column name used in the mapping. The fourth parameter should be set to true in case the primary key of the entity is the foreign key you're adding.

Discriminator Column

When joining an inheritance tree you have to give Doctrine a hint which meta-column is the discriminator column of this tree.

```php
<?php
/**
 * Sets a discriminator column for an entity result or joined entity result.
 * The discriminator column will be used to determine the concrete class name to
 * instantiate.
 *
 * @param string $alias The alias of the entity result or joined entity result the discriminator
 *                      column should be used for.
 * @param string $discrColumn The name of the discriminator column in the SQL result set.
 */
public function setDiscriminatorColumn($alias, $discrColumn)
```

Examples

Understanding a ResultSetMapping is probably easiest through looking at some examples.

First a basic example that describes the mapping of a single entity.

```php
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User owns no associations.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');

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

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

The result would look like this:

```
array(
    [0] => User (Object)
)
```

Note that this would be a partial object if the entity has more fields than just id and name. In the example above the column and field names are identical but that is not necessary, of course. Also note that the query string passed to createNativeQuery is **real native SQL**. Doctrine does not touch this SQL in any way.

In the previous basic example, a User had no relations and the table the class is mapped to owns no foreign keys. The next example assumes User has a unidirectional or bidirectional one-to-one association to a CmsAddress, where the User is the owning side and thus owns the foreign key.

```
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User owns an association to an Address but the Address is not loaded in the query.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addMetaResult('u', 'address_id', 'address_id');

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

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

Foreign keys are used by Doctrine for lazy-loading purposes when querying for objects. In the previous example, each user object in the result will have a proxy (a "ghost") in place of the address that contains the address_id. When the ghost proxy is accessed, it loads itself based on this key.

Consequently, associations that are *fetch-joined* do not require the foreign keys to be present in the SQL result set, only associations that are lazy.

```
<?php
// Equivalent DQL query: "select u from User u join u.address a WHERE u.name = ?1"
// User owns association to an Address and the Address is loaded in the query.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addJoinedEntityResult('Address' , 'a', 'u', 'address');
$rsm->addFieldResult('a', 'address_id', 'id');
$rsm->addFieldResult('a', 'street', 'street');
$rsm->addFieldResult('a', 'city', 'city');

$sql = 'SELECT u.id, u.name, a.id AS address_id, a.street, a.city FROM users u ' .
       'INNER JOIN address a ON u.address_id = a.id WHERE u.name = ?';
$query = $this->_em->createNativeQuery($sql, $rsm);
$query->setParameter(1, 'romanb');

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

In this case the nested entity `Address` is registered with the `ResultSetMapping#addJoinedEntityResult` method, which notifies Doctrine that this entity is not hydrated at the root level, but as a joined entity somewhere inside the object graph. In this case we specify the alias 'u' as third parameter and `address` as fourth parameter, which means the `Address` is hydrated into the `User::$address` property.

If a fetched entity is part of a mapped hierarchy that requires a discriminator column, this column must be present in the result set as a meta column so that Doctrine can create the appropriate concrete type. This is shown in the following example where we assume that there are one or more subclasses that extend User and either Class Table Inheritance or Single Table Inheritance is used to map the hierarchy (both use a discriminator column).

```php
<?php
// Equivalent DQL query: "select u from User u where u.name=?1"
// User is a mapped base class for other classes. User owns no associations.
$rsm = new ResultSetMapping;
$rsm->addEntityResult('User', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');
$rsm->addMetaResult('u', 'discr', 'discr'); // discriminator column
$rsm->setDiscriminatorColumn('u', 'discr');

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

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

Note that in the case of Class Table Inheritance, an example as above would result in partial objects if any objects in the result are actually a subtype of User. When using DQL, Doctrine automatically includes the necessary joins for this mapping strategy but with native SQL it is your responsibility.

Named Native Query

You can also map a native query using a named native query mapping.

To achieve that, you must describe the SQL resultset structure using named native query (and sql resultset mappings if is a several resultset mappings).

Like named query, a named native query can be defined at class level or in a XML or YAML file.

A resultSetMapping parameter is defined in @NamedNativeQuery, it represents the name of a defined @SqlResultSetMapping.

- *PHP*

```php
<?php
namespace MyProject\Model;
/**
 * @NamedNativeQueries({
 *      @NamedNativeQuery(
 *          name            = "fetchMultipleJoinsEntityResults",
 *          resultSetMapping= "mappingMultipleJoinsEntityResults",
 *          query           = "SELECT u.id AS u_id, u.name AS u_name, u.status AS u_status, a.ic
 *      ),
 * })
 * @SqlResultSetMappings({
 *      @SqlResultSetMapping(
 *          name    = "mappingMultipleJoinsEntityResults",
 *          entities= {
 *              @EntityResult(
 *                  entityClass = "__CLASS__",
 *                  fields      = {
 *                      @FieldResult(name = "id",       column="u_id"),
 *                          @FieldResult(name = "name",      column="u_name"),
 *                          @FieldResult(name = "status",    column="u_status"),
 *                      }
 *                 ),
 *                 @EntityResult(
 *                     entityClass = "Address",
 *                     fields      = {
 *                          @FieldResult(name = "id",        column="a_id"),
 *                          @FieldResult(name = "zip",       column="a_zip"),
 *                          @FieldResult(name = "country",   column="a_country"),
 *                      }
 *                 )
 *             },
 *             columns = {
 *                 @ColumnResult("numphones")
 *             }
 *         )
 *})
 */
 class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;

    /** @Column(type="string", length=50, nullable=true) */
    public $status;

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

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

    /** @OneToMany(targetEntity="Phonenumber") */
    public $phonenumbers;

    /** @OneToOne(targetEntity="Address") */
    public $address;

    // ....
}
```

- *XML*

```
<doctrine-mapping>
    <entity name="MyProject\Model\User">
        <named-native-queries>
            <named-native-query name="fetchMultipleJoinsEntityResults" result-set-mapping="mappi
                <query>SELECT u.id AS u_id, u.name AS u_name, u.status AS u_status, a.id AS a_id
            </named-native-query>
        </named-native-queries>
        <sql-result-set-mappings>
            <sql-result-set-mapping name="mappingMultipleJoinsEntityResults">
                <entity-result entity-class="__CLASS__">
                    <field-result name="id" column="u_id"/>
                    <field-result name="name" column="u_name"/>
                    <field-result name="status" column="u_status"/>
                </entity-result>
                <entity-result entity-class="Address">
                    <field-result name="id" column="a_id"/>
                    <field-result name="zip" column="a_zip"/>
                    <field-result name="country" column="a_country"/>
                </entity-result>
                <column-result name="numphones"/>
            </sql-result-set-mapping>
        </sql-result-set-mappings>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
MyProject\Model\User:
  type: entity
  namedNativeQueries:
    fetchMultipleJoinsEntityResults:
      name: fetchMultipleJoinsEntityResults
      resultSetMapping: mappingMultipleJoinsEntityResults
      query: SELECT u.id AS u_id, u.name AS u_name, u.status AS u_status, a.id AS a_id, a.zip AS
  sqlResultSetMappings:
    mappingMultipleJoinsEntityResults:
      name: mappingMultipleJoinsEntityResults
      columnResult:
        0:
          name: numphones
      entityResult:
        0:
          entityClass: __CLASS__
          fieldResult:
            0:
              name: id
              column: u_id
            1:
              name: name
              column: u_name
            2:
              name: status
              column: u_status
        1:
          entityClass: Address
          fieldResult:
            0:
              name: id
              column: a_id
            1:
              name: zip
              column: a_zip
            2:
              name: country
              column: a_country
```

Things to note:

- The resultset mapping declares the entities retrieved by this native query.
- Each field of the entity is bound to a SQL alias (or column name).
- All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query.

- Field definitions are optional provided that they map to the same column name as the one declared on the class property.
- __CLASS__ is an alias for the mapped class

In the above example, the `fetchJoinedAddress` named query use the joinMapping result set mapping. This mapping returns 2 entities, User and Address, each property is declared and associated to a column name, actually the column name retrieved by the query.

Let's now see an implicit declaration of the property / column.

- *PHP*

```
<?php
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name                = "findAll",
     *          resultSetMapping    = "mappingFindAll",
     *          query               = "SELECT * FROM addresses"
     *      ),
     * })
     * @SqlResultSetMappings({
     *      @SqlResultSetMapping(
     *          name    = "mappingFindAll",
     *          entities= {
     *              @EntityResult(
     *                  entityClass = "Address"
     *              )
     *          }
     *      )
     * })
     */
   class Address
   {
        /**  @Id @Column(type="integer") @GeneratedValue */
        public $id;

        /** @Column() */
        public $country;

        /** @Column() */
        public $zip;

        /** @Column()*/
        public $city;

        // ....
    }
```

- *XML*

```
<doctrine-mapping>
    <entity name="MyProject\Model\Address">
        <named-native-queries>
            <named-native-query name="findAll" result-set-mapping="mappingFindAll">
                <query>SELECT * FROM addresses</query>
            </named-native-query>
        </named-native-queries>
```

```
            <sql-result-set-mappings>
                <sql-result-set-mapping name="mappingFindAll">
                    <entity-result entity-class="Address"/>
                </sql-result-set-mapping>
            </sql-result-set-mappings>
        </entity>
    </doctrine-mapping>
```

- *YAML*

```
MyProject\Model\Address:
  type: entity
  namedNativeQueries:
    findAll:
      resultSetMapping: mappingFindAll
      query: SELECT * FROM addresses
  sqlResultSetMappings:
    mappingFindAll:
      name: mappingFindAll
      entityResult:
        address:
          entityClass: Address
```

In this example, we only describe the entity member of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the model property is bound to the model_txt column. If the association to a related entity involve a composite primary key, a @FieldResult element should be used for each foreign key column. The @FieldResult name is composed of the property name for the relationship, followed by a dot ("."), followed by the name or the field or property of the primary key.

- *PHP*

```
<?php
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name            = "fetchJoinedAddress",
     *          resultSetMapping= "mappingJoinedAddress",
     *          query           = "SELECT u.id, u.name, u.status, a.id AS a_id, a.country AS a_c
     *      ),
     * })
     * @SqlResultSetMappings({
     *      @SqlResultSetMapping(
     *          name    = "mappingJoinedAddress",
     *          entities= {
     *              @EntityResult(
     *                  entityClass = "__CLASS__",
     *                  fields      = {
     *                      @FieldResult(name = "id"),
     *                      @FieldResult(name = "name"),
     *                      @FieldResult(name = "status"),
     *                      @FieldResult(name = "address.id", column = "a_id"),
     *                      @FieldResult(name = "address.zip", column = "a_zip"),
     *                      @FieldResult(name = "address.city", column = "a_city"),
     *                      @FieldResult(name = "address.country", column = "a_country"),
     *                  }
     *              )
     *          }
     *      )
```

```
 * })
 */
class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;

    /** @Column(type="string", length=50, nullable=true) */
    public $status;

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

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

    /** @OneToOne(targetEntity="Address") */
    public $address;

    // ....
}
```

- *XML*

```
<doctrine-mapping>
    <entity name="MyProject\Model\User">
        <named-native-queries>
            <named-native-query name="fetchJoinedAddress" result-set-mapping="mappingJoinedAddre
                <query>SELECT u.id, u.name, u.status, a.id AS a_id, a.country AS a_country, a.zi
            </named-native-query>
        </named-native-queries>
        <sql-result-set-mappings>
            <sql-result-set-mapping name="mappingJoinedAddress">
                <entity-result entity-class="__CLASS__">
                    <field-result name="id"/>
                    <field-result name="name"/>
                    <field-result name="status"/>
                    <field-result name="address.id" column="a_id"/>
                    <field-result name="address.zip"  column="a_zip"/>
                    <field-result name="address.city"  column="a_city"/>
                    <field-result name="address.country" column="a_country"/>
                </entity-result>
            </sql-result-set-mapping>
        </sql-result-set-mappings>
    </entity>
</doctrine-mapping>
```

- *YAML*

```
MyProject\Model\User:
  type: entity
  namedNativeQueries:
    fetchJoinedAddress:
      name: fetchJoinedAddress
      resultSetMapping: mappingJoinedAddress
      query: SELECT u.id, u.name, u.status, a.id AS a_id, a.country AS a_country, a.zip AS a_zip
  sqlResultSetMappings:
    mappingJoinedAddress:
      entityResult:
        0:
          entityClass: __CLASS__
          fieldResult:
            0:
              name: id
            1:
              name: name
            2:
              name: status
            3:
              name: address.id
              column: a_id
            4:
              name: address.zip
              column: a_zip
            5:
              name: address.city
              column: a_city
            6:
              name: address.country
              column: a_country
```

If you retrieve a single entity and if you use the default mapping, you can use the resultClass attribute instead of resultSetMapping:

- *PHP*

```php
<?php
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name            = "find-by-id",
     *          resultClass     = "Address",
     *          query           = "SELECT * FROM addresses"
     *      ),
     * })
     */
   class Address
   {
        // ....
   }
```

- *XML*

```xml
<doctrine-mapping>
    <entity name="MyProject\Model\Address">
        <named-native-queries>
            <named-native-query name="find-by-id" result-class="Address">
                <query>SELECT * FROM addresses WHERE id = ?</query>
            </named-native-query>
        </named-native-queries>
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
MyProject\Model\Address:
  type: entity
  namedNativeQueries:
    findAll:
      name: findAll
      resultClass: Address
      query: SELECT * FROM addresses
```

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the @SqlResultsetMapping through @ColumnResult. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

- *PHP*

```php
<?php
namespace MyProject\Model;
    /**
     * @NamedNativeQueries({
     *      @NamedNativeQuery(
     *          name            = "count",
     *          resultSetMapping= "mappingCount",
     *          query           = "SELECT COUNT(*) AS count FROM addresses"
     *      )
     * })
     * @SqlResultSetMappings({
     *      @SqlResultSetMapping(
     *          name    = "mappingCount",
     *          columns = {
     *              @ColumnResult(
     *                  name = "count"
     *              )
     *          }
     *      )
     * })
     */
   class Address
   {
        // ....
   }
```

- *XML*

```xml
<doctrine-mapping>
    <entity name="MyProject\Model\Address">
        <named-native-query name="count" result-set-mapping="mappingCount">
            <query>SELECT COUNT(*) AS count FROM addresses</query>
        </named-native-query>
        <sql-result-set-mappings>
            <sql-result-set-mapping name="mappingCount">
                <column-result name="count"/>
            </sql-result-set-mapping>
        </sql-result-set-mappings>
    </entity>
</doctrine-mapping>
```

- *YAML*

```yaml
MyProject\Model\Address:
  type: entity
  namedNativeQueries:
    count:
      name: count
      resultSetMapping: mappingCount
      query: SELECT COUNT(*) AS count FROM addresses
  sqlResultSetMappings:
    mappingCount:
      name: mappingCount
      columnResult:
        count:
          name: count
```

9.2.18 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 Doctrine 2 only considers entities 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:

```php
<?php
/**
 * @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 namespace. As a guideline, such an implementation can look as follows:

```php
<?php
use Doctrine\Common\NotifyPropertyChanged,
    Doctrine\Common\PropertyChangedListener;

/**
 * @Entity
 * @ChangeTrackingPolicy("NOTIFY")
 */
class MyEntity implements NotifyPropertyChanged
{
    // ...

    private $_listeners = array();

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

Then, in each property setter of this class or derived classes, you need to notify all the `PropertyChangedListener` instances. As an example we add a convenience method on `MyEntity` that shows this behaviour:

```php
<?php
// ...

class MyEntity implements NotifyPropertyChanged
{
    // ...

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

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

You have to invoke `_onPropertyChanged` inside every method that changes the persistent state of `MyEntity`.

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.

9.2.19 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.

Note: 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.

Warning: Use of partial objects is tricky. Fields that are not retrieved from the database will not be updated by the UnitOfWork even if they get changed in your objects. You can only promote a partial object to a fully-loaded object by calling `EntityManager#refresh()` or a DQL query with the refresh flag.

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 don't 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 `partial` keyword as follows:

```php
<?php
$q = $em->createQuery("select partial u.{id,name} from MyApp\Domain\User u");
```

You can also get a partial reference instead of a proxy reference by calling:

```
<?php
$reference = $em->getPartialReference('MyApp\Domain\User', 1);
```

Partial references are objects with only the identifiers set as they are passed to the second argument of the `getPartialReference()` method. All other fields are null.

When should I force partial objects?

Mainly for optimization purposes, but be careful of premature optimization as partial objects lead to potentially more fragile code.

9.2.20 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. 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
                   https://raw.github.com/doctrine/doctrine2/master/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 (.). For example an Entity with the fully qualified class-name "MyProject" would require a mapping file "MyProject.Entities.User.dcm.xml" unless the extension is changed.
- 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');
```

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 = new \Doctrine\ORM\Configuration();
$driver = new \Doctrine\ORM\Mapping\Driver\XmlDriver(array('/path/to/files1', '/path/to/files2'));
$config->setMetadataDriverImpl($driver);
```

Warning: Note that Doctrine ORM does not modify any settings for `libxml`, therefore, external XML entities may or may not be enabled or configured correctly. XML mappings are not XXE/XEE attack vectors since they are not related with user input, but it is recommended that you do not use external XML entities in your mapping files to avoid running into unexpected behaviour.

Simplified XML Driver

The Symfony project sponsored a driver that simplifies usage of the XML Driver. The changes between the original driver are:

1. File Extension is .orm.xml
2. Filenames are shortened, “MyProjectEntitiesUser” will become User.orm.xml
3. You can add a global file and add multiple entities in this file.

Configuration of this client works a little bit different:

```
<?php
$namespaces = array(
    '/path/to/files1' => 'MyProject\Entities',
    '/path/to/files2' => 'OtherProject\Entities'
);
$driver = new \Doctrine\ORM\Mapping\Driver\SimplifiedXmlDriver($namespaces);
$driver->setGlobalBasename('global'); // global.orm.xml
```

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
                          http://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

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

        <indexes>
            <index name="name_idx" columns="name"/>
            <index columns="user_email"/>
        </indexes>

        <unique-constraints>
            <unique-constraint columns="name,user_email" name="search_idx" />
        </unique-constraints>

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

        <id name="id" type="integer" column="id">
            <generator strategy="AUTO"/>
            <sequence-generator sequence-name="tablename_seq" allocation-size="100" initial-value="1"
        </id>

        <field name="name" column="name" type="string" length="50" nullable="true" unique="true" />
        <field name="email" column="user_email" type="string" column-definition="CHAR(32) NOT NULL" /

        <one-to-one field="address" target-entity="Address" inversed-by="user">
            <cascade><cascade-remove /></cascade>
            <join-column name="address_id" referenced-column-name="id" on-delete="CASCADE" on-update=
        </one-to-one>

        <one-to-many field="phonenumbers" target-entity="Phonenumber" mapped-by="user">
            <cascade>
                <cascade-persist/>
            </cascade>
            <order-by>
                <order-by-field name="number" direction="ASC" />
            </order-by>
        </one-to-many>

        <many-to-many field="groups" target-entity="Group">
            <cascade>
                <cascade-all/>
            </cascade>
            <join-table name="cms_users_groups">
                <join-columns>
                    <join-column name="user_id" referenced-column-name="id" nullable="false" unique="
                </join-columns>
                <inverse-join-columns>
                    <join-column name="group_id" referenced-column-name="id" column-definition="INT N
                </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.

XML-Element Reference

The XML-Element reference explains all the tags and attributes that the Doctrine Mapping XSD Schema defines. You should read the Basic-, Association- and Inheritance Mapping chapters to understand what each of this definitions means in detail.

Defining an Entity

Each XML Mapping File contains the definition of one entity, specified as the `<entity />` element as a direct child of the `<doctrine-mapping />` element:

```
<doctrine-mapping>
    <entity name="MyProject\User" table="cms_users" schema="schema_name" repository-class="MyProject\
        <!-- definition here -->
    </entity>
</doctrine-mapping>
```

Required attributes:

- name - The fully qualified class-name of the entity.

Optional attributes:

- **table** - The Table-Name to be used for this entity. Otherwise the Unqualified Class-Name is used by default.
- **repository-class** - The fully qualified class-name of an alternative `Doctrine\ORM\EntityRepository` implementation to be used with this entity.
- **inheritance-type** - The type of inheritance, defaults to none. A more detailed description follows in the *Defining Inheritance Mappings* section.
- **read-only** - (>= 2.1) Specifies that this entity is marked as read only and not considered for change-tracking. Entities of this type can be persisted and removed though.
- **schema** - (>= 2.5) The schema the table lies in, for platforms that support schemas

Defining Fields

Each entity class can contain zero to infinite fields that are managed by Doctrine. You can define them using the `<field />` element as a children to the `<entity />` element. The field element is only used for primitive types that are not the ID of the entity. For the ID mapping you have to use the `<id />` element.

```xml
<entity name="MyProject\User">

    <field name="name" type="string" length="50" />
    <field name="username" type="string" unique="true" />
    <field name="age" type="integer" nullable="true" />
    <field name="isActive" column="is_active" type="boolean" />
    <field name="weight" type="decimal" scale="5" precision="2" />
    <field name="login_count" type="integer" nullable="false">
        <options>
            <option name="comment">The number of times the user has logged in.</option>
            <option name="default">0</option>
        </options>
    </field>
</entity>
```

Required attributes:

- name - The name of the Property/Field on the given Entity PHP class.

Optional attributes:

- type - The `Doctrine\DBAL\Types\Type` name, defaults to “string”
- column - Name of the column in the database, defaults to the field name.
- length - The length of the given type, for use with strings only.
- unique - Should this field contain a unique value across the table? Defaults to false.
- nullable - Should this field allow NULL as a value? Defaults to false.
- version - Should this field be used for optimistic locking? Only works on fields with type integer or datetime.
- scale - Scale of a decimal type.
- precision - Precision of a decimal type.
- options - Array of additional options:

  - default - The default value to set for the column if no value is supplied.
  - unsigned - Boolean value to determine if the column should be capable of representing only non-negative integers (applies only for integer column and might not be supported by all vendors).
  - fixed - Boolean value to determine if the specified length of a string column should be fixed or varying (applies only for string/binary column and might not be supported by all vendors).
  - comment - The comment of the column in the schema (might not be supported by all vendors).
  - customSchemaOptions - Array of additional schema options which are mostly vendor specific.

- column-definition - Optional alternative SQL representation for this column. This definition begin after the field-name and has to specify the complete column definition. Using this feature will turn this field dirty for Schema-Tool update commands at all times.

Note: For more detailed information on each attribute, please refer to the DBAL `Schema-Representation` documentation.

Defining Identity and Generator Strategies

An entity has to have at least one `<id />` element. For composite keys you can specify more than one id-element, however surrogate keys are recommended for use with Doctrine 2. The Id field allows to define properties of the identifier and allows a subset of the `<field />` element attributes:

```xml
<entity name="MyProject\User">
    <id name="id" type="integer" column="user_id" />
</entity>
```

Required attributes:

- name - The name of the Property/Field on the given Entity PHP class.
- type - The `Doctrine\DBAL\Types\Type` name, preferably "string" or "integer".

Optional attributes:

- column - Name of the column in the database, defaults to the field name.

Using the simplified definition above Doctrine will use no identifier strategy for this entity. That means you have to manually set the identifier before calling `EntityManager#persist($entity)`. This is the so called `ASSIGNED` strategy.

If you want to switch the identifier generation strategy you have to nest a `<generator />` element inside the id-element. This of course only works for surrogate keys. For composite keys you always have to use the `ASSIGNED` strategy.

```xml
<entity name="MyProject\User">
    <id name="id" type="integer" column="user_id">
        <generator strategy="AUTO" />
    </id>
</entity>
```

The following values are allowed for the `<generator />` strategy attribute:

- AUTO - Automatic detection of the identifier strategy based on the preferred solution of the database vendor.
- IDENTITY - Use of a IDENTIFY strategy such as Auto-Increment IDs available to Doctrine AFTER the INSERT statement has been executed.

- SEQUENCE - Use of a database sequence to retrieve the entity-ids. This is possible before the INSERT statement is executed.

If you are using the SEQUENCE strategy you can define an additional element to describe the sequence:

```xml
<entity name="MyProject\User">
    <id name="id" type="integer" column="user_id">
        <generator strategy="SEQUENCE" />
        <sequence-generator sequence-name="user_seq" allocation-size="5" initial-value="1" />
    </id>
</entity>
```

Required attributes for `<sequence-generator />`:

- sequence-name - The name of the sequence

Optional attributes for `<sequence-generator />`:

- allocation-size - By how much steps should the sequence be incremented when a value is retrieved. Defaults to 1
- initial-value - What should the initial value of the sequence be.

  NOTE

  If you want to implement a cross-vendor compatible application you have to specify and additionally define the <sequence-generator /> element, if Doctrine chooses the sequence strategy for a platform.

Defining a Mapped Superclass

Sometimes you want to define a class that multiple entities inherit from, which itself is not an entity however. The chapter on *Inheritance Mapping* describes a Mapped Superclass in detail. You can define it in XML using the `<mapped-superclass />` tag.

```xml
<doctrine-mapping>
    <mapped-superclass name="MyProject\BaseClass">
        <field name="created" type="datetime" />
        <field name="updated" type="datetime" />
    </mapped-superclass>
</doctrine-mapping>
```

Required attributes:

- name - Class name of the mapped superclass.

You can nest any number of `<field />` and unidirectional `<many-to-one />` or `<one-to-one />` associations inside a mapped superclass.

Defining Inheritance Mappings

There are currently two inheritance persistence strategies that you can choose from when defining entities that inherit from each other. Single Table inheritance saves the fields of the complete inheritance hierarchy in a single table, joined table inheritance creates a table for each entity combining the fields using join conditions.

You can specify the inheritance type in the `<entity />` element and then use the `<discriminator-column />` and `<discriminator-mapping />` attributes.

```xml
<entity name="MyProject\Animal" inheritance-type="JOINED">
    <discriminator-column name="discr" type="string" />
    <discriminator-map>
```

```xml
        <discriminator-mapping value="cat" class="MyProject\Cat" />
        <discriminator-mapping value="dog" class="MyProject\Dog" />
        <discriminator-mapping value="mouse" class="MyProject\Mouse" />
    </discriminator-map>
</entity>
```

The allowed values for inheritance-type attribute are `JOINED` or `SINGLE_TABLE`.

Note: All inheritance related definitions have to be defined on the root entity of the hierarchy.

Defining Lifecycle Callbacks

You can define the lifecycle callback methods on your entities using the `<lifecycle-callbacks />` element:

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

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

Defining One-To-One Relations

You can define One-To-One Relations/Associations using the `<one-to-one />` element. The required and optional attributes depend on the associations being on the inverse or owning side.

For the inverse side the mapping is as simple as:

```xml
<entity class="MyProject\User">
    <one-to-one field="address" target-entity="Address" mapped-by="user" />
</entity>
```

Required attributes for inverse One-To-One:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANT:* No leading backslash!
- mapped-by - Name of the field on the owning side (here Address entity) that contains the owning side association.

For the owning side this mapping would look like:

```xml
<entity class="MyProject\Address">
    <one-to-one field="user" target-entity="User" inversed-by="address" />
</entity>
```

Required attributes for owning One-to-One:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANT:* No leading backslash!

Optional attributes for owning One-to-One:

- inversed-by - If the association is bidirectional the inversed-by attribute has to be specified with the name of the field on the inverse entity that contains the back-reference.

- orphan-removal - If true, the inverse side entity is always deleted when the owning side entity is. Defaults to false.
- fetch - Either LAZY or EAGER, defaults to LAZY. This attribute makes only sense on the owning side, the inverse side *ALWAYS* has to use the `FETCH` strategy.

The definition for the owning side relies on a bunch of mapping defaults for the join column names. Without the nested `<join-column />` element Doctrine assumes to foreign key to be called `user_id` on the Address Entities table. This is because the `MyProject\Address` entity is the owning side of this association, which means it contains the foreign key.

The completed explicitly defined mapping is:

```
<entity class="MyProject\Address">
    <one-to-one field="user" target-entity="User" inversed-by="address">
        <join-column name="user_id" referenced-column-name="id" />
    </one-to-one>
</entity>
```

Defining Many-To-One Associations

The many-to-one association is *ALWAYS* the owning side of any bidirectional association. This simplifies the mapping compared to the one-to-one case. The minimal mapping for this association looks like:

```
<entity class="MyProject\Article">
    <many-to-one field="author" target-entity="User" />
</entity>
```

Required attributes:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANT:* No leading backslash!

Optional attributes:

- inversed-by - If the association is bidirectional the inversed-by attribute has to be specified with the name of the field on the inverse entity that contains the back-reference.
- orphan-removal - If true the entity on the inverse side is always deleted when the owning side entity is and it is not connected to any other owning side entity anymore. Defaults to false.
- fetch - Either LAZY or EAGER, defaults to LAZY.

This definition relies on a bunch of mapping defaults with regards to the naming of the join-column/foreign key. The explicitly defined mapping includes a `<join-column />` tag nested inside the many-to-one association tag:

```
<entity class="MyProject\Article">
    <many-to-one field="author" target-entity="User">
        <join-column name="author_id" referenced-column-name="id" />
    </many-to-one>
</entity>
```

The join-column attribute `name` specifies the column name of the foreign key and the `referenced-column-name` attribute specifies the name of the primary key column on the User entity.

Defining One-To-Many Associations

The one-to-many association is *ALWAYS* the inverse side of any association. There exists no such thing as a uni-directional one-to-many association, which means this association only ever exists for bi-directional associations.

```
<entity class="MyProject\User">
    <one-to-many field="phonenumbers" target-entity="Phonenumber" mapped-by="user" />
</entity>
```

Required attributes:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANT:* No leading backslash!
- mapped-by - Name of the field on the owning side (here Phonenumber entity) that contains the owning side association.

Optional attributes:

- fetch - Either LAZY, EXTRA_LAZY or EAGER, defaults to LAZY.
- index-by: Index the collection by a field on the target entity.

Defining Many-To-Many Associations

From all the associations the many-to-many has the most complex definition. When you rely on the mapping defaults you can omit many definitions and rely on their implicit values.

```
<entity class="MyProject\User">
    <many-to-many field="groups" target-entity="Group" />
</entity>
```

Required attributes:

- field - Name of the property/field on the entity's PHP class.
- target-entity - Name of the entity associated entity class. If this is not qualified the namespace of the current class is prepended. *IMPORTANT:* No leading backslash!

Optional attributes:

- mapped-by - Name of the field on the owning side that contains the owning side association if the defined many-to-many association is on the inverse side.
- inversed-by - If the association is bidirectional the inversed-by attribute has to be specified with the name of the field on the inverse entity that contains the back-reference.
- fetch - Either LAZY, EXTRA_LAZY or EAGER, defaults to LAZY.
- index-by: Index the collection by a field on the target entity.

The mapping defaults would lead to a join-table with the name "User_Group" being created that contains two columns "user_id" and "group_id". The explicit definition of this mapping would be:

```
<entity class="MyProject\User">
    <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>
```

Here both the `<join-columns>` and `<inverse-join-columns>` tags are necessary to tell Doctrine for which side the specified join-columns apply. These are nested inside a `<join-table />` attribute which allows to specify the table name of the many-to-many join-table.

Cascade Element

Doctrine allows cascading of several UnitOfWork operations to related entities. You can specify the cascade operations in the `<cascade />` element inside any of the association mapping tags.

```
<entity class="MyProject\User">
    <many-to-many field="groups" target-entity="Group">
        <cascade>
            <cascade-all/>
        </cascade>
    </many-to-many>
</entity>
```

Besides `<cascade-all />` the following operations can be specified by their respective tags:

- `<cascade-persist />`
- `<cascade-merge />`
- `<cascade-remove />`
- `<cascade-refresh />`

Join Column Element

In any explicitly defined association mapping you will need the `<join-column />` tag. It defines how the foreign key and primary key names are called that are used for joining two entities.

Required attributes:

- name - The column name of the foreign key.
- referenced-column-name - The column name of the associated entities primary key

Optional attributes:

- unique - If the join column should contain a UNIQUE constraint. This makes sense for Many-To-Many join-columns only to simulate a one-to-many unidirectional using a join-table.
- nullable - should the join column be nullable, defaults to true.
- on-delete - Foreign Key Cascade action to perform when entity is deleted, defaults to NO ACTION/RESTRICT but can be set to "CASCADE".

Defining Order of To-Many Associations

You can require one-to-many or many-to-many associations to be retrieved using an additional `ORDER BY`.

```xml
<entity class="MyProject\User">
    <many-to-many field="groups" target-entity="Group">
        <order-by>
            <order-by-field name="name" direction="ASC" />
        </order-by>
    </many-to-many>
</entity>
```

Defining Indexes or Unique Constraints

To define additional indexes or unique constraints on the entities table you can use the `<indexes />` and `<unique-constraints />` elements:

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

    <indexes>
        <index name="name_idx" columns="name"/>
        <index columns="user_email"/>
    </indexes>

    <unique-constraints>
        <unique-constraint columns="name,user_email" name="search_idx" />
    </unique-constraints>
</entity>
```

You have to specify the column and not the entity-class field names in the index and unique-constraint definitions.

Derived Entities ID syntax

If the primary key of an entity contains a foreign key to another entity we speak of a derived entity relationship. You can define this in XML with the "association-key" attribute in the `<id>` tag.

```xml
<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://raw.github.com/doctrine/doctrine2/master/doctrine-mapping.xsd">

     <entity name="Application\Model\ArticleAttribute">
        <id name="article" association-key="true" />
        <id name="attribute" type="string" />

        <field name="value" type="string" />

        <many-to-one field="article" target-entity="Article" inversed-by="attributes" />
     </entity>

</doctrine-mapping>
```

9.2.21 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
<?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
<?php
use Doctrine\ORM\Mapping\Driver\YamlDriver;

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

Simplified YAML Driver

The Symfony project sponsored a driver that simplifies usage of the YAML Driver. The changes between the original driver are:

- File Extension is .orm.yml
- Filenames are shortened, "MyProject\Entities\User" will become User.orm.yml
- You can add a global file and add multiple entities in this file.

Configuration of this client works a little bit different:

```php
<?php
$namespaces = array(
    '/path/to/files1' => 'MyProject\Entities',
    '/path/to/files2' => 'OtherProject\Entities'
);
$driver = new \Doctrine\ORM\Mapping\Driver\SimplifiedYamlDriver($namespaces);
$driver->setGlobalBasename('global'); // global.orm.yml
```

Example

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

```yaml
# Doctrine.Tests.ORM.Mapping.User.dcm.yml
Doctrine\Tests\ORM\Mapping\User:
  type: entity
  repositoryClass: Doctrine\Tests\ORM\Mapping\UserRepository
  table: cms_users
  schema: schema_name # The schema the table lies in, for platforms that support schemas (Optional,
  readOnly: true
  indexes:
    name_index:
      columns: [ name ]
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
      length: 50
    email:
      type: string
      length: 32
      column: user_email
      unique: true
      options:
        fixed: true
        comment: User's email address
    loginCount:
      type: integer
      column: login_count
      nullable: false
      options:
        unsigned: true
        default: 0
  oneToOne:
    address:
      targetEntity: Address
      joinColumn:
        name: address_id
        referencedColumnName: id
        onDelete: CASCADE
  oneToMany:
    phonenumbers:
      targetEntity: Phonenumber
      mappedBy: user
      cascade: ["persist", "merge"]
  manyToMany:
    groups:
      targetEntity: Group
      joinTable:
        name: cms_users_groups
        joinColumns:
          user_id:
            referencedColumnName: id
        inverseJoinColumns:
          group_id:
            referencedColumnName: id
  lifecycleCallbacks:
    prePersist: [ doStuffOnPrePersist, doOtherStuffOnPrePersistToo ]
    postPersist: [ doStuffOnPostPersist ]
```

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

Reference

Unique Constraints

It is possible to define unique constraints by the following declaration:

```
# ECommerceProduct.orm.yml
ECommerceProduct:
  type: entity
  fields:
    # definition of some fields
  uniqueConstraints:
    search_idx:
      columns: [ name, email ]
```

9.2.22 Annotations Reference

You've probably used docblock annotations in some form already, most likely to provide documentation metadata for a tool like `PHPDocumentor` (@author, @link, ...). Docblock annotations are a tool to embed metadata inside the documentation section which can then be processed by some tool. Doctrine 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.

Note: 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.

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

Index

- *@JoinColumn*
- *@JoinColumns*
- *@JoinTable*
- *@ManyToOne*
- *@ManyToMany*
- *@MappedSuperclass*
- *@NamedNativeQuery*
- *@OneToOne*
- *@OneToMany*
- *@OrderBy*
- *@PostLoad*
- *@PostPersist*
- *@PostRemove*
- *@PostUpdate*
- *@PrePersist*
- *@PreRemove*
- *@PreUpdate*
- *@SequenceGenerator*
- *@SqlResultSetMapping*
- *@Table*
- *@UniqueConstraint*
- *@Version*

Reference

@Column

Marks an annotated instance variable as "persistent". It has to be inside the instance variables PHP DocBlock comment. Any value hold inside this variable will be saved to and loaded from the database as part of the lifecycle of the instance variables entity-class.

Required attributes:

- **type**: Name of the Doctrine Type which is converted between PHP and Database representation.

Optional attributes:

- **name**: By default the property name is used for the database column name also, however the 'name' attribute allows you to determine the column name.
- **length**: Used by the "string" type to determine its maximum length in the database. Doctrine does not validate the length of a string values for you.
- **precision**: The precision for a decimal (exact numeric) column (applies only for decimal column), which is the maximum number of digits that are stored for the values.

- **scale**: The scale for a decimal (exact numeric) column (applies only for decimal column), which represents the number of digits to the right of the decimal point and must not be greater than *precision*.
- **unique**: Boolean value to determine if the value of the column should be unique across all rows of the underlying entities table.
- **nullable**: Determines if NULL values allowed for this column.
- **options**: Array of additional options:
  - `default`: The default value to set for the column if no value is supplied.
  - `unsigned`: Boolean value to determine if the column should be capable of representing only non-negative integers (applies only for integer column and might not be supported by all vendors).
  - `fixed`: Boolean value to determine if the specified length of a string column should be fixed or varying (applies only for string/binary column and might not be supported by all vendors).
  - `comment`: The comment of the column in the schema (might not be supported by all vendors).
  - `customSchemaOptions`: Array of additional schema options which are mostly vendor specific.
- **columnDefinition**: DDL SQL snippet that starts after the column name and specifies the complete (non-portable!) column definition. This attribute allows to make use of advanced RMDBS features. However you should make careful use of this feature and the consequences. SchemaTool will not detect changes on the column correctly anymore if you use "columnDefinition".

  Additionally you should remember that the "type" attribute still handles the conversion between PHP and Database values. If you use this attribute on a column that is used for joins between tables you should also take a look at *@JoinColumn*.

Note: For more detailed information on each attribute, please refer to the DBAL `Schema-Representation` documentation.

Examples:

```
<?php
/**
 * @Column(type="string", length=32, unique=true, nullable=false)
 */
protected $username;

/**
 * @Column(type="string", columnDefinition="CHAR(2) NOT NULL")
 */
protected $country;

/**
 * @Column(type="decimal", precision=2, scale=1)
 */
protected $height;

/**
 * @Column(type="string", length=2, options={"fixed":true, "comment":"Initial letters of first and la
 */
protected $initials;

/**
 * @Column(type="integer", name="login_count" nullable=false, options={"unsigned":true, "default":0})
 */
protected $loginCount;
```

@ColumnResult

References name of a column in the SELECT clause of a SQL query. Scalar result types can be included in the query result by specifying this annotation in the metadata.

Required attributes:

- **name**: The name of a column in the SELECT clause of a SQL query

@Cache

Add caching strategy to a root entity or a collection.

Optional attributes:

- **usage**: One of `READ_ONLY`, `READ_WRITE` or `NONSTRICT_READ_WRITE`, By default this is `READ_ONLY`.
- **region**: An specific region name

@ChangeTrackingPolicy

The Change Tracking Policy annotation allows to specify how the Doctrine 2 UnitOfWork should detect changes in properties of entities during flush. By default each entity is checked according to a deferred implicit strategy, which means upon flush UnitOfWork compares all the properties of an entity to a previously stored snapshot. This works out of the box, however you might want to tweak the flush performance where using another change tracking policy is an interesting option.

The *details on all the available change tracking policies* can be found in the configuration section.

Example:

```
<?php
/**
 * @Entity
 * @ChangeTrackingPolicy("DEFERRED_IMPLICIT")
 * @ChangeTrackingPolicy("DEFERRED_EXPLICIT")
 * @ChangeTrackingPolicy("NOTIFY")
 */
class User {}
```

@DiscriminatorColumn

This annotation is a required annotation for the topmost/super class of an inheritance hierarchy. It specifies the details of the column which saves the name of the class, which the entity is actually instantiated as.

Required attributes:

- **name**: The column name of the discriminator. This name is also used during Array hydration as key to specify the class-name.

Optional attributes:

- **type**: By default this is string.
- **length**: By default this is 255.

@DiscriminatorMap

The discriminator map is a required annotation on the topmost/super class in an inheritance hierarchy. Its only argument is an array which defines which class should be saved under which name in the database. Keys are the database value and values are the classes, either as fully- or as unqualified class names depending on whether the classes are in the namespace or not.

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

@Entity

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

Optional attributes:

- **repositoryClass**: Specifies the FQCN of a subclass of the EntityRepository. Use of repositories for entities is encouraged to keep specialized DQL and SQL operations separated from the Model/Domain Layer.
- **readOnly**: (>= 2.1) Specifies that this entity is marked as read only and not considered for change-tracking. Entities of this type can be persisted and removed though.

Example:

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

@EntityResult

References an entity in the SELECT clause of a SQL query. If this annotation is used, the SQL statement should select all of the columns that are mapped to the entity object. This should include foreign key columns to related entities. The results obtained when insufficient data is available are undefined.

Required attributes:

- **entityClass**: The class of the result.

Optional attributes:

- **fields**: Array of @FieldResult, Maps the columns specified in the SELECT list of the query to the properties or fields of the entity class.

- **discriminatorColumn**: Specifies the column name of the column in the SELECT list that is used to determine the type of the entity instance.

@FieldResult

Is used to map the columns specified in the SELECT list of the query to the properties or fields of the entity class.

Required attributes:

- **name**: Name of the persistent field or property of the class.

Optional attributes:

- **column**: Name of the column in the SELECT clause.

@GeneratedValue

Specifies which strategy is used for identifier generation for an instance variable which is annotated by *@Id*. This annotation is optional and only has meaning when used in conjunction with @Id.

If this annotation is not specified with @Id the NONE strategy is used as default.

Required attributes:

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

Example:

```php
<?php
/**
 * @Id
 * @Column(type="integer")
 * @GeneratedValue(strategy="IDENTITY")
 */
protected $id = null;
```

@HasLifecycleCallbacks

Annotation which has to be set on the entity-class PHP DocBlock to notify Doctrine that this entity has entity lifecycle callback annotations set on at least one of its methods. Using @PostLoad, @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate or @PostUpdate without this marker annotation will make Doctrine ignore the callbacks.

Example:

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

@Index

Annotation is used inside the *@Table* annotation on the entity-class level. It provides a hint to the SchemaTool to generate a database index on the specified table columns. It only has meaning in the SchemaTool schema generation context.

Required attributes:

- **name**: Name of the Index
- **columns**: Array of columns.

Optional attributes:

- **options**: Array of platform specific options:
  - `where`: SQL WHERE condition to be used for partial indexes. It will only have effect on supported platforms.

Basic example:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",indexes={@Index(name="search_idx", columns={"name", "email"})})
 */
class ECommerceProduct
{
}
```

Example with partial indexes:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",indexes={@Index(name="search_idx", columns={"name", "email"}, opt
 */
class ECommerceProduct
{
}
```

@Id

The annotated instance variable will be marked as entity identifier, the primary key in the database. This annotation is a marker only and has no required or optional attributes. For entities that have multiple identifier columns each column has to be marked with @Id.

Example:

```
<?php
/**
 * @Id
 * @Column(type="integer")
 */
protected $id = null;
```

@InheritanceType

In an inheritance hierarchy you have to use this annotation on the topmost/super class to define which strategy should be used for inheritance. Currently Single Table and Class Table Inheritance are supported.

This annotation has always been used in conjunction with the *@DiscriminatorMap* and *@DiscriminatorColumn* annotations.

Examples:

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

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

@JoinColumn

This annotation is used in the context of relations in *@ManyToOne*, *@OneToOne* fields and in the Context of *@JoinTable* nested inside a @ManyToMany. This annotation is not required. If it is not specified the attributes *name* and *referencedColumnName* are inferred from the table and primary key names.

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 whether this relation is exclusive between the affected entities and should be enforced as such on the database constraint level. Defaults to false.
- **nullable**: Determine whether the related entity is required, or if null is an allowed state for the relation. Defaults to true.
- **onDelete**: Cascade Action (Database-level)
- **columnDefinition**: DDL SQL snippet that starts after the column name and specifies the complete (non-portable!) column definition. This attribute enables the use of advanced RMDBS features. Using this attribute on @JoinColumn is necessary if you need slightly different column definitions for joining columns, for example

regarding NULL/NOT NULL defaults. However by default a "columnDefinition" attribute on *@Column* also sets the related @JoinColumn's columnDefinition. This is necessary to make foreign keys work.

Example:

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

@JoinColumns

An array of @JoinColumn annotations for a *@ManyToOne* or *@OneToOne* relation with an entity that has multiple identifiers.

@JoinTable

Using *@OneToMany* or *@ManyToMany* on the owning side of the relation requires to specify the @JoinTable annotation which describes the details of the database join table. If you do not specify @JoinTable on these relations reasonable mapping defaults apply using the affected table and the column names.

Optional attributes:

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

Example:

```
<?php
/**
 * @ManyToMany(targetEntity="Phonenumber")
 * @JoinTable(name="users_phonenumbers",
 *      joinColumns={@JoinColumn(name="user_id", referencedColumnName="id")},
 *      inverseJoinColumns={@JoinColumn(name="phonenumber_id", referencedColumnName="id", unique=true
 * )
 */
public $phonenumbers;
```

@ManyToOne

Defines that the annotated instance variable holds a reference that describes a many-to-one relationship between two entities.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **cascade**: Cascade Option

- **fetch**: One of LAZY or EAGER
- inversedBy - The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.

Example:

```
<?php
/**
 * @ManyToOne(targetEntity="Cart", cascade={"all"}, fetch="EAGER")
 */
private $cart;
```

@ManyToMany

Defines that the annotated instance variable holds a many-to-many relationship between two entities. *@JoinTable* is an additional, optional annotation that has reasonable default configuration values using the table and names of the two related entities.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **mappedBy**: This option specifies the property name on the targetEntity that is the owning side of this relation. It is a required attribute for the inverse side of a relationship.
- **inversedBy**: The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.
- **cascade**: Cascade Option
- **fetch**: One of LAZY, EXTRA_LAZY or EAGER
- **indexBy**: Index the collection by a field on the target entity.

Note: For ManyToMany bidirectional relationships either side may be the owning side (the side that defines the @JoinTable and/or does not make use of the mappedBy attribute, thus using a default join table).

Example:

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

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

@MappedSuperclass

A mapped superclass is an abstract or concrete class that provides persistent entity state and mapping information for its subclasses, but which is not itself an entity. This annotation is specified on the Class docblock and has no additional attributes.

The @MappedSuperclass annotation cannot be used in conjunction with @Entity. See the Inheritance Mapping section for *more details on the restrictions of mapped superclasses*.

Optional attributes:

- **repositoryClass**: (>= 2.2) Specifies the FQCN of a subclass of the EntityRepository. That will be inherited for all subclasses of that Mapped Superclass.

Example:

```php
<?php
/**
 * @MappedSuperclass
 */
class MappedSuperclassBase
{
    // ... fields and methods
}

/**
 * @Entity
 */
class EntitySubClassFoo extends MappedSuperclassBase
{
    // ... fields and methods
}
```

@NamedNativeQuery

Is used to specify a native SQL named query. The NamedNativeQuery annotation can be applied to an entity or mapped superclass.

Required attributes:

- **name**: The name used to refer to the query with the EntityManager methods that create query objects.
- **query**: The SQL query string.

Optional attributes:

- **resultClass**: The class of the result.
- **resultSetMapping**: The name of a SqlResultSetMapping, as defined in metadata.

Example:

```php
<?php
/**
 * @NamedNativeQueries({
 *      @NamedNativeQuery(
 *          name            = "fetchJoinedAddress",
 *          resultSetMapping= "mappingJoinedAddress",
 *          query           = "SELECT u.id, u.name, u.status, a.id AS a_id, a.country, a.zip, a.city
 *      ),
 * })
 * @SqlResultSetMappings({
 *      @SqlResultSetMapping(
 *          name    = "mappingJoinedAddress",
 *          entities= {
 *              @EntityResult(
 *                  entityClass = "__CLASS__",
 *                  fields      = {
 *                      @FieldResult(name = "id"),
 *                      @FieldResult(name = "name"),
 *                      @FieldResult(name = "status"),
 *                      @FieldResult(name = "address.zip"),
 *                      @FieldResult(name = "address.city"),
 *                      @FieldResult(name = "address.country"),
 *                      @FieldResult(name = "address.id", column = "a_id"),
 *                  }
 *              )
 *          }
 *      )
 * })
 */
class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;

    /** @Column(type="string", length=50, nullable=true) */
    public $status;

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

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

    /** @OneToOne(targetEntity="Address") */
    public $address;

    // ....
}
```

@OneToOne

The @OneToOne annotation works almost exactly as the *@ManyToOne* with one additional option which can be specified. The configuration defaults for *@JoinColumn* using the target entity table and primary key column names apply here too.

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

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

- **orphanRemoval**: Boolean that specifies if orphans, inverse OneToOne entities that are not connected to any owning instance, should be removed by Doctrine. Defaults to false.
- **inversedBy**: The inversedBy attribute designates the field in the entity that is the inverse side of the relationship.

Example:

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

@OneToMany

Required attributes:

- **targetEntity**: FQCN of the referenced target entity. Can be the unqualified class name if both classes are in the same namespace. *IMPORTANT:* No leading backslash!

Optional attributes:

- **cascade**: Cascade Option
- **orphanRemoval**: Boolean that specifies if orphans, inverse OneToOne entities that are not connected to any owning instance, should be removed by Doctrine. Defaults to false.
- **mappedBy**: This option specifies the property name on the targetEntity that is the owning side of this relation. Its a required attribute for the inverse side of a relationship.
- **fetch**: One of LAZY, EXTRA_LAZY or EAGER.
- **indexBy**: Index the collection by a field on the target entity.

Example:

```
<?php
/**
 * @OneToMany(targetEntity="Phonenumber", mappedBy="user", cascade={"persist", "remove", "merge"}, o
 */
public $phonenumbers;
```

@OrderBy

Optional annotation that can be specified with a *@ManyToMany* or *@OneToMany* annotation to specify by which criteria the collection should be retrieved from the database by using an ORDER BY clause.

This annotation requires a single non-attributed value with an DQL snippet:

Example:

```
<?php
/**
 * @ManyToMany(targetEntity="Group")
 * @OrderBy({"name" = "ASC"})
 */
private $groups;
```

The DQL Snippet in OrderBy is only allowed to consist of unqualified, unquoted field names and of an optional ASC/DESC positional statement. Multiple Fields are separated by a comma (,). The referenced field names have to exist on the `targetEntity` class of the `@ManyToMany` or `@OneToMany` annotation.

@PostLoad

Marks a method on the entity to be called as a @PostLoad event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostPersist

Marks a method on the entity to be called as a @PostPersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostRemove

Marks a method on the entity to be called as a @PostRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PostUpdate

Marks a method on the entity to be called as a @PostUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PrePersist

Marks a method on the entity to be called as a @PrePersist event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PreRemove

Marks a method on the entity to be called as a @PreRemove event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@PreUpdate

Marks a method on the entity to be called as a @PreUpdate event. Only works with @HasLifecycleCallbacks in the entity class PHP DocBlock.

@SequenceGenerator

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

Required attributes:

- **sequenceName**: Name of the sequence

Optional attributes:

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

Example:

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

@SqlResultSetMapping

The SqlResultSetMapping annotation is used to specify the mapping of the result of a native SQL query. The SqlResultSetMapping annotation can be applied to an entity or mapped superclass.

Required attributes:

- **name**: The name given to the result set mapping, and used to refer to it in the methods of the Query API.

Optional attributes:

- **entities**: Array of @EntityResult, Specifies the result set mapping to entities.
- **columns**: Array of @ColumnResult, Specifies the result set mapping to scalar values.

Example:

```
<?php
/**
 * @NamedNativeQueries({
 *      @NamedNativeQuery(
 *          name            = "fetchUserPhonenumberCount",
 *          resultSetMapping= "mappingUserPhonenumberCount",
 *          query           = "SELECT id, name, status, COUNT(phonenumber) AS numphones FROM cms_use
 *      ),
 *      @NamedNativeQuery(
 *          name            = "fetchMultipleJoinsEntityResults",
 *          resultSetMapping= "mappingMultipleJoinsEntityResults",
 *          query           = "SELECT u.id AS u_id, u.name AS u_name, u.status AS u_status, a.id AS 
 *      ),
 * })
 * @SqlResultSetMappings({
 *      @SqlResultSetMapping(
 *          name    = "mappingUserPhonenumberCount",
 *          entities= {
 *              @EntityResult(
 *                  entityClass = "User",
 *                  fields      = {
 *                      @FieldResult(name = "id"),
 *                      @FieldResult(name = "name"),
 *                      @FieldResult(name = "status"),
 *                  }
 *               )
 *           },
 *           columns = {
 *               @ColumnResult("numphones")
 *           }
 *       ),
 *       @SqlResultSetMapping(
 *           name    = "mappingMultipleJoinsEntityResults",
 *           entities= {
 *               @EntityResult(
 *                   entityClass = "__CLASS__",
 *                   fields      = {
 *                       @FieldResult(name = "id",       column="u_id"),
 *                       @FieldResult(name = "name",     column="u_name"),
 *                       @FieldResult(name = "status",   column="u_status"),
 *                   }
 *               ),
 *               @EntityResult(
 *                   entityClass = "Address",
 *                   fields      = {
 *                       @FieldResult(name = "id",       column="a_id"),
 *                       @FieldResult(name = "zip",      column="a_zip"),
 *                       @FieldResult(name = "country",  column="a_country"),
 *                   }
 *               )
 *           },
 *           columns = {
 *               @ColumnResult("numphones")
 *           }
 *       )
 *})
 */
 class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    public $id;

    /** @Column(type="string", length=50, nullable=true) */
    public $status;

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

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

    /** @OneToMany(targetEntity="Phonenumber") */
    public $phonenumbers;

    /** @OneToOne(targetEntity="Address") */
    public $address;

    // ....
}
```

@Table

Annotation describes the table an entity is persisted in. It is placed on the entity-class PHP DocBlock and is optional. If it is not specified the table name will default to the entity's unqualified classname.

Required attributes:

- **name**: Name of the table

Optional attributes:

- **indexes**: Array of @Index annotations
- **uniqueConstraints**: Array of @UniqueConstraint annotations.
- **schema**: (>= 2.5) Name of the schema the table lies in.

Example:

```
<?php
/**
 * @Entity
 * @Table(name="user",
 *      uniqueConstraints={@UniqueConstraint(name="user_unique",columns={"username"})},
 *      indexes={@Index(name="user_idx", columns={"email"})}
 *      schema="schema_name"
 * )
 */
class User { }
```

@UniqueConstraint

Annotation is used inside the *@Table* annotation on the entity-class level. It allows to hint the SchemaTool to generate a database unique constraint on the specified table columns. It only has meaning in the SchemaTool schema generation context.

Required attributes:

- **name**: Name of the Index
- **columns**: Array of columns.

Optional attributes:

- **options**: Array of platform specific options:
  - `where`: SQL WHERE condition to be used for partial indexes. It will only have effect on supported platforms.

Basic example:

```
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx", columns=
 */
class ECommerceProduct
{
}
```

Example with partial indexes:

```php
<?php
/**
 * @Entity
 * @Table(name="ecommerce_products",uniqueConstraints={@UniqueConstraint(name="search_idx", columns=
 */
class ECommerceProduct
{
}
```

@Version

Marker annotation that defines a specified column as version attribute used in an optimistic locking scenario. It only works on *@Column* annotations that have the type integer or datetime. Combining @Version with *@Id* is not supported.

Example:

```php
<?php
/**
 * @Column(type="integer")
 * @Version
 */
protected $version;
```

9.2.23 PHP Mapping

Doctrine 2 also allows you to provide the ORM metadata in the form of plain PHP code using the `ClassMetadata` API. You can write the code in PHP files or inside of a static function named `loadMetadata($class)` on the entity class itself.

PHP Files

If you wish to write your mapping information inside PHP files that are named after the entity and included to populate the metadata for an entity you can do so by using the `PHPDriver`:

```php
<?php
$driver = new PHPDriver('/path/to/php/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

Now imagine we had an entity named `Entities\User` and we wanted to write a mapping file for it using the above configured `PHPDriver` instance:

```php
<?php
namespace Entities;

class User
{
    private $id;
    private $username;
}
```

To write the mapping information you just need to create a file named `Entities.User.php` inside of the `/path/to/php/mapping/files` folder:

```php
<?php
// /path/to/php/mapping/files/Entities.User.php

$metadata->mapField(array(
   'id' => true,
   'fieldName' => 'id',
   'type' => 'integer'
));

$metadata->mapField(array(
   'fieldName' => 'username',
   'type' => 'string',
   'options' => array(
       'fixed' => true,
       'comment' => "User's login name"
   )
));

$metadata->mapField(array(
   'fieldName' => 'login_count',
   'type' => 'integer',
   'nullable' => false,
   'options' => array(
       'unsigned' => true,
       'default' => 0
   )
));
```

Now we can easily retrieve the populated `ClassMetadata` instance where the `PHPDriver` includes the file and the `ClassMetadataFactory` caches it for later retrieval:

```php
<?php
$class = $em->getClassMetadata('Entities\User');
// or
$class = $em->getMetadataFactory()->getMetadataFor('Entities\User');
```

Static Function

In addition to the PHP files you can also specify your mapping information inside of a static function defined on the entity class itself. This is useful for cases where you want to keep your entity and mapping information together but don't want to use annotations. For this you just need to use the `StaticPHPDriver`:

```php
<?php
$driver = new StaticPHPDriver('/path/to/entities');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

Now you just need to define a static function named `loadMetadata($metadata)` on your entity:

```php
<?php
namespace Entities;

use Doctrine\ORM\Mapping\ClassMetadata;

class User
{
    // ...
    public static function loadMetadata(ClassMetadata $metadata)
    {
        $metadata->mapField(array(
           'id' => true,
           'fieldName' => 'id',
           'type' => 'integer'
        ));

        $metadata->mapField(array(
           'fieldName' => 'username',
           'type' => 'string'
        ));
    }
}
```

ClassMetadataBuilder

To ease the use of the ClassMetadata API (which is very raw) there is a `ClassMetadataBuilder` that you can use.

```
<?php
namespace Entities;

use Doctrine\ORM\Mapping\ClassMetadata;
use Doctrine\ORM\Mapping\Builder\ClassMetadataBuilder;

class User
{
    // ...

    public static function loadMetadata(ClassMetadata $metadata)
    {
        $builder = new ClassMetadataBuilder($metadata);
        $builder->createField('id', 'integer')->isPrimaryKey()->generatedValue()->build();
        $builder->addField('username', 'string');
    }
}
```

The API of the ClassMetadataBuilder has the following methods with a fluent interface:

- `addField($name, $type, array $mapping)`
- `setMappedSuperclass()`
- `setReadOnly()`
- `setCustomRepositoryClass($className)`
- `setTable($name)`
- `addIndex(array $columns, $indexName)`
- `addUniqueConstraint(array $columns, $constraintName)`
- `addNamedQuery($name, $dqlQuery)`
- `setJoinedTableInheritance()`
- `setSingleTableInheritance()`
- `setDiscriminatorColumn($name, $type = 'string', $length = 255)`

- `addDiscriminatorMapClass($name, $class)`
- `setChangeTrackingPolicyDeferredExplicit()`
- `setChangeTrackingPolicyNotify()`
- `addLifecycleEvent($methodName, $event)`
- `addManyToOne($name, $targetEntity, $inversedBy = null)`
- `addInverseOneToOne($name, $targetEntity, $mappedBy)`
- `addOwningOneToOne($name, $targetEntity, $inversedBy = null)`
- `addOwningManyToMany($name, $targetEntity, $inversedBy = null)`
- `addInverseManyToMany($name, $targetEntity, $mappedBy)`
- `addOneToMany($name, $targetEntity, $mappedBy)`

It also has several methods that create builders (which are necessary for advanced mappings):

- `createField($name, $type)` returns a `FieldBuilder` instance
- `createManyToOne($name, $targetEntity)` returns an `AssociationBuilder` instance
- `createOneToOne($name, $targetEntity)` returns an `AssociationBuilder` instance
- `createManyToMany($name, $targetEntity)` returns an `ManyToManyAssociationBuilder` instance
- `createOneToMany($name, $targetEntity)` returns an `OneToManyAssociationBuilder` instance

ClassMetadataInfo API

The `ClassMetadataInfo` class is the base data object for storing the mapping metadata for a single entity. It contains all the getters and setters you need populate and retrieve information for an entity.

General Setters

- `setTableName($tableName)`
- `setPrimaryTable(array $primaryTableDefinition)`
- `setCustomRepositoryClass($repositoryClassName)`
- `setIdGeneratorType($generatorType)`
- `setIdGenerator($generator)`
- `setSequenceGeneratorDefinition(array $definition)`
- `setChangeTrackingPolicy($policy)`
- `setIdentifier(array $identifier)`

Inheritance Setters

- `setInheritanceType($type)`
- `setSubclasses(array $subclasses)`

- `setParentClasses(array $classNames)`
- `setDiscriminatorColumn($columnDef)`
- `setDiscriminatorMap(array $map)`

Field Mapping Setters

- `mapField(array $mapping)`
- `mapOneToOne(array $mapping)`
- `mapOneToMany(array $mapping)`
- `mapManyToOne(array $mapping)`
- `mapManyToMany(array $mapping)`

Lifecycle Callback Setters

- `addLifecycleCallback($callback, $event)`
- `setLifecycleCallbacks(array $callbacks)`

Versioning Setters

- `setVersionMapping(array &$mapping)`
- `setVersioned($bool)`
- `setVersionField()`

General Getters

- `getTableName()`
- `getSchemaName()`
- `getTemporaryIdTableName()`

Identifier Getters

- `getIdentifierColumnNames()`
- `usesIdGenerator()`
- `isIdentifier($fieldName)`
- `isIdGeneratorIdentity()`
- `isIdGeneratorSequence()`
- `isIdGeneratorTable()`
- `isIdentifierNatural()`
- `getIdentifierFieldNames()`
- `getSingleIdentifierFieldName()`

- `getSingleIdentifierColumnName()`

Inheritance Getters

- `isInheritanceTypeNone()`
- `isInheritanceTypeJoined()`
- `isInheritanceTypeSingleTable()`
- `isInheritanceTypeTablePerClass()`
- `isInheritedField($fieldName)`
- `isInheritedAssociation($fieldName)`

Change Tracking Getters

- `isChangeTrackingDeferredExplicit()`
- `isChangeTrackingDeferredImplicit()`
- `isChangeTrackingNotify()`

Field & Association Getters

- `isUniqueField($fieldName)`
- `isNullable($fieldName)`
- `getColumnName($fieldName)`
- `getFieldMapping($fieldName)`
- `getAssociationMapping($fieldName)`
- `getAssociationMappings()`
- `getFieldName($columnName)`
- `hasField($fieldName)`
- `getColumnNames(array $fieldNames = null)`
- `getTypeOfField($fieldName)`
- `getTypeOfColumn($columnName)`
- `hasAssociation($fieldName)`
- `isSingleValuedAssociation($fieldName)`
- `isCollectionValuedAssociation($fieldName)`

Lifecycle Callback Getters

- `hasLifecycleCallbacks($lifecycleEvent)`
- `getLifecycleCallbacks($event)`

ClassMetadata API

The `ClassMetadata` class extends `ClassMetadataInfo` and adds the runtime functionality required by Doctrine. It adds a few extra methods related to runtime reflection for working with the entities themselves.

- `getReflectionClass()`
- `getReflectionProperties()`
- `getReflectionProperty($name)`
- `getSingleIdReflectionProperty()`
- `getIdentifierValues($entity)`
- `setIdentifierValues($entity, $id)`
- `setFieldValue($entity, $field, $value)`
- `getFieldValue($entity, $field)`

9.2.24 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, when using `ArrayCache`, the cache does not persist 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 this interface.

The interface defines the following public methods for you to implement:

- 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. use the above protected methods which 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 in the PHP Documentation. 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.

```
<?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 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();
$cacheDriver->setMemcache($memcache);
$cacheDriver->save('cache_id', 'my_data');
```

Memcached

Memcached is a more recent and complete alternative extension to Memcache.

In order to use the Memcached cache driver you must have it compiled and enabled in your php.ini. You can read about Memcached 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 Memcached cache driver by itself.

```
<?php
$memcached = new Memcached();
$memcached->addServer('memcache_host', 11211);

$cacheDriver = new \Doctrine\Common\Cache\MemcachedCache();
$cacheDriver->setMemcached($memcached);
$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. 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');
```

Redis

In order to use the Redis cache driver you must have it compiled and enabled in your php.ini. You can read about what Redis is from here. Also check A PHP extension for Redis for how you can use and install the Redis PHP extension.

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

```
<?php
$redis = new Redis();
$redis->connect('redis_host', 6379);

$cacheDriver = new \Doctrine\Common\Cache\RedisCache();
$cacheDriver->setRedis($redis);
$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 data to a cache, check if some cached data 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

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

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

Checking

Checking whether cached data 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 again 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. You can delete by an individual ID, or you can delete all entries.

By Cache ID

```php
<?php
$cacheDriver->delete('my_array');
```

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

```php
<?php
$deleted = $cacheDriver->deleteAll();
```

Namespaces

If you heavily use caching in your application and use 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
<?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 the performance of various aspects of Doctrine by 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
$config->setResultCacheImpl(new \Doctrine\Common\Cache\ApcCache());
```

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

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

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

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

```
<?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
$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
$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 how you can use the API of the cache drivers to manually delete cache entries. For your convenience we offer a command line task to help you with clearing the query, result and metadata cache.

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

```
$ ./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.

```
$ ./doctrine clear-cache --query
```

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

```
$ ./doctrine clear-cache --metadata
```

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

```
$ ./doctrine clear-cache --result
```

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

Cache Slams

Something to be careful of when using the cache drivers is "cache slams". Imagine 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 to 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 in this blog post.

9.2.25 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 (preferably 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.

Read-Only Entities

Starting with Doctrine 2.1 you can mark entities as read only (See metadata mapping references for details). This means that the entity marked as read only is never considered for updates, which means when you call flush on the EntityManager these entities are skipped even if properties changed. Read-Only allows to persist new entities of a kind and remove existing ones, they are just not considered for updates.

Extra-Lazy Collections

If entities hold references to large collections you will get performance and memory problems initializing them. To solve this issue you can use the EXTRA_LAZY fetch-mode feature for collections. See the *tutorial* for more information on how this fetch mode works.

Temporarily change fetch mode in DQL

See *Doctrine Query Language chapter*

Apply Best Practices

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

Change Tracking policies

See: `Change Tracking Policies`

9.2.26 Tools

Doctrine Console

The Doctrine Console is a Command Line Interface tool for simplifying common administration tasks during the development of a project that uses Doctrine 2.

Take a look at the *Installation and Configuration* chapter for more information how to setup the console command.

Display Help Information

Type `php vendor/bin/doctrine` on the command line and you should see an overview of the available commands or use the –help flag to get information on the available commands. If you want to know more about the use of generate entities for example, you can call:

```
$> php vendor/bin/doctrine orm:generate-entities --help
```

Configuration

Whenever the `doctrine` command line tool is invoked, it can access all Commands that were registered by developer. There is no auto-detection mechanism at work. The Doctrine binary already registers all the commands that currently ship with Doctrine DBAL and ORM. If you want to use additional commands you have to register them yourself.

All the commands of the Doctrine Console require access to the `EntityManager` or `DBAL` Connection. You have to inject them into the console application using so called Helper-Sets. This requires either the `db` or the `em` helpers to be defined in order to work correctly.

Whenever you invoke the Doctrine binary the current folder is searched for a `cli-config.php` file. This file contains the project specific configuration:

```php
<?php
$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($conn)
));
$cli->setHelperSet($helperSet);
```

When dealing with the ORM package, the EntityManagerHelper is required:

```php
<?php
$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em)
));
$cli->setHelperSet($helperSet);
```

The HelperSet instance has to be generated in a separate file (i.e. `cli-config.php`) that contains typical Doctrine bootstrap code and predefines the needed HelperSet attributes mentioned above. A sample `cli-config.php` file looks as follows:

```php
<?php
// cli-config.php
require_once 'my_bootstrap.php';

// Any way to access the EntityManager from  your application
$em = GetMyEntityManager();

$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($em->getConnection()),
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em)
));
```

It is important to define a correct HelperSet that Doctrine binary script will ultimately use. The Doctrine Binary will automatically find the first instance of HelperSet in the global variable namespace and use this.

Note: You have to adjust this snippet for your specific application or framework and use their facilities to access the

Doctrine EntityManager and Connection Resources.

Command Overview

The following Commands are currently available:

- `help` Displays help for a command (?)
- `list` Lists commands
- `dbal:import` Import SQL file(s) directly to Database.
- `dbal:run-sql` Executes arbitrary SQL directly from the command line.
- `orm:clear-cache:metadata` Clear all metadata cache of the various cache drivers.
- `orm:clear-cache:query` Clear all query cache of the various cache drivers.
- `orm:clear-cache:result` Clear result cache of the various cache drivers.
- `orm:convert-d1-schema` Converts Doctrine 1.X schema into a Doctrine 2.X schema.
- `orm:convert-mapping` Convert mapping information between supported formats.
- `orm:ensure-production-settings` Verify that Doctrine is properly configured for a production environment.
- `orm:generate-entities` Generate entity classes and method stubs from your mapping information.
- `orm:generate-proxies` Generates proxy classes for entity classes.
- `orm:generate-repositories` Generate repository classes from your mapping information.
- `orm:run-dql` Executes arbitrary DQL directly from the command line.
- `orm:schema-tool:create` Processes the schema and either create it directly on EntityManager Storage Connection or generate the SQL output.
- `orm:schema-tool:drop` Processes the schema and either drop the database schema of EntityManager Storage Connection or generate the SQL output.
- `orm:schema-tool:update` Processes the schema and either update the database schema of EntityManager Storage Connection or generate the SQL output.

For these commands are also available aliases:

- `orm:convert:d1-schema` is alias for `orm:convert-d1-schema`.
- `orm:convert:mapping` is alias for `orm:convert-mapping`.
- `orm:generate:entities` is alias for `orm:generate-entities`.
- `orm:generate:proxies` is alias for `orm:generate-proxies`.
- `orm:generate:repositories` is alias for `orm:generate-repositories`.

Note: Console also supports auto completion, for example, instead of `orm:clear-cache:query` you can use just `o:c:q`.

Database Schema Generation

Note: SchemaTool can do harm to your database. It will drop or alter tables, indexes, sequences and such. Please use this tool with caution in development and not on a production server. It is meant for helping you develop your Database Schema, but NOT with migrating schema from A to B in production. A safe approach would be generating the SQL on development server and saving it into SQL Migration files that are executed manually on the production server.

SchemaTool assumes your Doctrine Project uses the given database on its own. Update and Drop commands will mess with other tables if they are not related to the current project that is using Doctrine. Please be careful!

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

When using the SchemaTool class directly, create your schema using the `createSchema()` method. First create an instance of the `SchemaTool` and pass it an instance of the `EntityManager` that you want to use to create the schema. This method receives an array of `ClassMetadataInfo` instances.

```php
<?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
<?php
$tool->dropSchema($classes);
```

This drops all the tables that are currently used by your metadata model. When you are changing your metadata a lot during development you might want to drop the complete database instead of only the tables of the current model to clean up with orphaned tables.

```php
<?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
<?php
$tool->updateSchema($classes);
```

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

To create the schema use the `create` command:

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

To drop the schema use the `drop` command:

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

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

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

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

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

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

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

Before using the orm:schema-tool commands, remember to configure your cli-config.php properly.

Note: When using the Annotation Mapping Driver you have to either setup your autoloader in the cli-config.php correctly to find all the entities, or you can use the second argument of the `EntityManagerHelper` to specify all the paths of your entities (or mapping files), i.e. `new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em, $mappingPaths);`

Entity Generation

Generate entity classes and method stubs from your mapping information.

```
$ php doctrine orm:generate-entities
$ php doctrine orm:generate-entities --update-entities
$ php doctrine orm:generate-entities --regenerate-entities
```

This command is not suited for constant usage. It is a little helper and does not support all the mapping edge cases very well. You still have to put work in your entities after using this command.

It is possible to use the EntityGenerator on code that you have already written. It will not be lost. The EntityGenerator will only append new code to your file and will not delete the old code. However this approach may still be prone to error and we suggest you use code repositories such as GIT or SVN to make backups of your code.

It makes sense to generate the entity code if you are using entities as Data Access Objects only and don't put much additional logic on them. If you are however putting much more logic on the entities you should refrain from using the entity-generator and code your entities manually.

Note: Even if you specified Inheritance options in your XML or YAML Mapping files the generator cannot generate the base and child classes for you correctly, because it doesn't know which class is supposed to extend which. You have to adjust the entity code manually for inheritance to work!

Convert Mapping Information

Convert mapping information between supported formats.

This is an **execute one-time** command. It should not be necessary for you to call this method multiple times, especially when using the `--from-database` flag.

Converting an existing database schema into mapping files only solves about 70-80% of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

Note: There is no need to convert YAML or XML mapping files to annotations every time you make changes. All mapping drivers are first class citizens in Doctrine 2 and can be used as runtime mapping for the ORM. See the docs on XML and YAML Mapping for an example how to register this metadata drivers as primary mapping source.

To convert some mapping information between the various supported formats you can use the `ClassMetadataExporter` to get exporter instances for the different formats:

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

Once you have a instance you can use it to get an exporter. For example, the yml exporter:

```php
<?php
$exporter = $cme->getExporter('yml', '/path/to/export/yml');
```

Now you can export some `ClassMetadata` instances:

```php
<?php
$classes = array(
  $em->getClassMetadata('Entities\User'),
  $em->getClassMetadata('Entities\Profile')
);
$exporter->setMetadata($classes);
$exporter->export();
```

This functionality is also available from the command line to convert your loaded mapping information to another format. The `orm:convert-mapping` command accepts two arguments, the type to convert to and the path to generate it:

```
$ php doctrine orm:convert-mapping xml /path/to/mapping-path-converted-to-xml
```

Reverse Engineering

You can use the `DatabaseDriver` to reverse engineer a database to an array of `ClassMetadataInfo` instances and generate YAML, XML, etc. from them.

Note: Reverse Engineering is a **one-time** process that can get you started with a project. Converting an existing database schema into mapping files only detects about 70-80% of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

First you need to retrieve the metadata instances with the `DatabaseDriver`:

```php
<?php
$em->getConfiguration()->setMetadataDriverImpl(
    new \Doctrine\ORM\Mapping\Driver\DatabaseDriver(
        $em->getConnection()->getSchemaManager()
    )
);

$cmf = new DisconnectedClassMetadataFactory();
$cmf->setEntityManager($em);
$metadata = $cmf->getAllMetadata();
```

Now you can get an exporter instance and export the loaded metadata to yml:

```php
<?php
$exporter = $cme->getExporter('yml', '/path/to/export/yml');
$exporter->setMetadata($metadata);
$exporter->export();
```

You can also reverse engineer a database using the `orm:convert-mapping` command:

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

Note: Reverse Engineering is not always working perfectly depending on special cases. It will only detect Many-To-One relations (even if they are One-To-One) and will try to create entities from Many-To-Many tables. It also has problems with naming of foreign keys that have multiple column names. Any Reverse Engineered Database-Schema needs considerable manual work to become a useful domain model.

Runtime vs Development Mapping Validation

For performance reasons Doctrine 2 has to skip some of the necessary validation of metadata mappings. You have to execute this validation in your development workflow to verify the associations are correctly defined.

You can either use the Doctrine Command Line Tool:

```
doctrine orm:validate-schema
```

Or you can trigger the validation manually:

```php
<?php
use Doctrine\ORM\Tools\SchemaValidator;

$validator = new SchemaValidator($entityManager);
$errors = $validator->validateMapping();

if (count($errors) > 0) {
    // Lots of errors!
    echo implode("\n\n", $errors);
}
```

If the mapping is invalid the errors array contains a positive number of elements with error messages.

Warning: One mapping option that is not validated is the use of the referenced column name. It has to point to the equivalent primary key otherwise Doctrine will not work.

Note: One common error is to use a backlash in front of the fully-qualified class-name. Whenever a FQCN is represented inside a string (such as in your mapping definitions) you have to drop the prefix backslash. PHP does this with `get_class()` or Reflection methods for backwards compatibility reasons.

Adding own commands

You can also add your own commands on-top of the Doctrine supported tools if you are using a manually built console script.

To include a new command on Doctrine Console, you need to do modify the `doctrine.php` file a little:

```php
<?php
// doctrine.php
use Symfony\Component\Console\Helper\Application;

// as before ...

// replace the ConsoleRunner::run() statement with:
$cli = new Application('Doctrine Command Line Interface', \Doctrine\ORM\Version::VERSION);
$cli->setCatchExceptions(true);
```

```
$cli->setHelperSet($helperSet);

// Register All Doctrine Commands
ConsoleRunner::addCommands($cli);

// Register your own command
$cli->addCommand(new \MyProject\Tools\Console\Commands\MyCustomCommand);

// Runs console application
$cli->run();
```

Additionally, include multiple commands (and overriding previously defined ones) is possible through the command:

```
<?php

$cli->addCommands(array(
    new \MyProject\Tools\Console\Commands\MyCustomCommand(),
    new \MyProject\Tools\Console\Commands\SomethingCommand(),
    new \MyProject\Tools\Console\Commands\AnotherCommand(),
    new \MyProject\Tools\Console\Commands\OneMoreCommand(),
));
```

Re-use console application

You are also able to retrieve and re-use the default console application. Just call `ConsoleRunner::createApplication(...)` with an appropriate HelperSet, like it is described in the configuration section.

```
<?php

// Retrieve default console application
$cli = ConsoleRunner::createApplication($helperSet);

// Runs console application
$cli->run();
```

9.2.27 Metadata Drivers

The heart of an object relational mapper is the mapping information that glues everything together. It instructs the EntityManager how it should behave when dealing with the different entities.

Core Metadata Drivers

Doctrine provides a few different ways for you to specify your metadata:

- **XML files** (XmlDriver)
- **Class DocBlock Annotations** (AnnotationDriver)
- **YAML files** (YamlDriver)
- **PHP Code in files or static functions** (PhpDriver)

Something important to note about the above drivers is they are all an intermediate step to the same end result. The mapping information is populated to `Doctrine\ORM\Mapping\ClassMetadata` instances. So in the end, Doctrine only ever has to work with the API of the `ClassMetadata` class to get mapping information for an entity.

Note: The populated `ClassMetadata` instances are also cached so in a production environment the parsing and populating only ever happens once. You can configure the metadata cache implementation using the `setMetadataCacheImpl()` method on the `Doctrine\ORM\Configuration` class:

```php
<?php
$em->getConfiguration()->setMetadataCacheImpl(new ApcCache());
```

If you want to use one of the included core metadata drivers you just need to configure it. All the drivers are in the `Doctrine\ORM\Mapping\Driver` namespace:

```php
<?php
$driver = new \Doctrine\ORM\Mapping\Driver\XmlDriver('/path/to/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

Implementing Metadata Drivers

In addition to the included metadata drivers you can very easily implement your own. All you need to do is define a class which implements the `Driver` interface:

```php
<?php
namespace Doctrine\ORM\Mapping\Driver;

use Doctrine\ORM\Mapping\ClassMetadataInfo;

interface Driver
{
    /**
     * Loads the metadata for the specified class into the provided container.
     *
     * @param string $className
     * @param ClassMetadataInfo $metadata
     */
    function loadMetadataForClass($className, ClassMetadataInfo $metadata);

    /**
     * Gets the names of all mapped classes known to this driver.
     *
     * @return array The names of all mapped classes known to this driver.
     */
    function getAllClassNames();

    /**
     * Whether the class with the specified name should have its metadata loaded.
     * This is only the case if it is either mapped as an Entity or a
     * MappedSuperclass.
     *
     * @param string $className
     * @return boolean
     */
    function isTransient($className);
}
```

If you want to write a metadata driver to parse information from some file format we've made your life a little easier by providing the `AbstractFileDriver` implementation for you to extend from:

```
<?php
class MyMetadataDriver extends AbstractFileDriver
{
    /**
     * {@inheritdoc}
     */
    protected $_fileExtension = '.dcm.ext';

    /**
     * {@inheritdoc}
     */
    public function loadMetadataForClass($className, ClassMetadataInfo $metadata)
    {
        $data = $this->_loadMappingFile($file);

        // populate ClassMetadataInfo instance from $data
    }

    /**
     * {@inheritdoc}
     */
    protected function _loadMappingFile($file)
    {
        // parse contents of $file and return php data structure
    }
}
```

Note: When using the `AbstractFileDriver` it requires that you only have one entity defined per file and the file named after the class described inside where namespace separators are replaced by periods. So if you have an entity named `Entities\User` and you wanted to write a mapping file for your driver above you would need to name the file `Entities.User.dcm.ext` for it to be recognized.

Now you can use your `MyMetadataDriver` implementation by setting it with the `setMetadataDriverImpl()` method:

```
<?php
$driver = new MyMetadataDriver('/path/to/mapping/files');
$em->getConfiguration()->setMetadataDriverImpl($driver);
```

ClassMetadata

The last piece you need to know and understand about metadata in Doctrine 2 is the API of the `ClassMetadata` classes. You need to be familiar with them in order to implement your own drivers but more importantly to retrieve mapping information for a certain entity when needed.

You have all the methods you need to manually specify the mapping information instead of using some mapping file to populate it from. The base `ClassMetadataInfo` class is responsible for only data storage and is not meant for runtime use. It does not require that the class actually exists yet so it is useful for describing some entity before it exists and using that information to generate for example the entities themselves. The class `ClassMetadata` extends `ClassMetadataInfo` and adds some functionality required for runtime usage and requires that the PHP class is present and can be autoloaded.

You can read more about the API of the `ClassMetadata` classes in the PHP Mapping chapter.

Getting ClassMetadata Instances

If you want to get the `ClassMetadata` instance for an entity in your project to programmatically use some mapping information to generate some HTML or something similar you can retrieve it through the `ClassMetadataFactory`:

```php
<?php
$cmf = $em->getMetadataFactory();
$class = $cmf->getMetadataFor('MyEntityName');
```

Now you can learn about the entity and use the data stored in the `ClassMetadata` instance to get all mapped fields for example and iterate over them:

```php
<?php
foreach ($class->fieldMappings as $fieldMapping) {
    echo $fieldMapping['fieldName'] . "\n";
}
```

9.2.28 Best Practices

The best practices mentioned here that affect database design generally refer to best practices when working with Doctrine and do not necessarily reflect best practices for database design in general.

Constrain relationships as much as possible

It is important to constrain relationships as much as possible. This means:

- Impose a traversal direction (avoid bidirectional associations if possible)
- Eliminate nonessential associations

This has several benefits:

- Reduced coupling in your domain model
- Simpler code in your domain model (no need to maintain bidirectionality properly)
- Less work for Doctrine

Avoid composite keys

Even though Doctrine fully supports composite keys it is best not to use them if possible. Composite keys require additional work by Doctrine and thus have a higher probability of errors.

Use events judiciously

The event system of Doctrine is great and fast. Even though making heavy use of events, especially lifecycle events, can have a negative impact on the performance of your application. Thus you should use events judiciously.

Use cascades judiciously

Automatic cascades of the persist/remove/merge/etc. operations are very handy but should be used wisely. Do NOT simply add all cascades to all associations. Think about which cascades actually do make sense for you for a particular association, given the scenarios it is most likely used in.

Don't use special characters

Avoid using any non-ASCII characters in class, field, table or column names. Doctrine itself is not unicode-safe in many places and will not be until PHP itself is fully unicode-aware (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.

Use explicit transaction demarcation

While Doctrine will automatically wrap all DML operations in a transaction on flush(), it is considered best practice to explicitly set the transaction boundaries yourself. Otherwise every single query is wrapped in a small transaction (Yes, SELECT queries, too) since you can not talk to your database outside of a transaction. While such short transactions for read-only (SELECT) queries generally don't have any noticeable performance impact, it is still preferable to use fewer, well-defined transactions that are established through explicit transaction boundaries.

9.2.29 Limitations and Known Issues

We try to make using Doctrine2 a very pleasant experience. Therefore we think it is very important to be honest about the current limitations to our users. Much like every other piece of software Doctrine2 is not perfect and far from feature complete. This section should give you an overview of current limitations of Doctrine 2 as well as critical known issues that you should know about.

Current Limitations

There is a set of limitations that exist currently which might be solved in the future. Any of this limitations now stated has at least one ticket in the Tracker and is discussed for future releases.

Join-Columns with non-primary keys

It is not possible to use join columns pointing to non-primary keys. Doctrine will think these are the primary keys and create lazy-loading proxies with the data, which can lead to unexpected results. Doctrine can for performance reasons not validate the correctness of this settings at runtime but only through the Validate Schema command.

Mapping Arrays to a Join Table

Related to the previous limitation with "Foreign Keys as Identifier" you might be interested in mapping the same table structure as given above to an array. However this is not yet possible either. See the following example:

```
CREATE TABLE product (
    id INTEGER,
    name VARCHAR,
    PRIMARY KEY(id)
);

CREATE TABLE product_attributes (
    product_id INTEGER,
    attribute_name VARCHAR,
    attribute_value VARCHAR,
    PRIMARY KEY (product_id, attribute_name)
);
```

This schema should be mapped to a Product Entity as follows:

```
class Product
{
    private $id;
    private $name;
    private $attributes = array();
}
```

Where the `attribute_name` column contains the key and `attribute_value` contains the value of each array element in `$attributes`.

The feature request for persistence of primitive value arrays is described in the DDC-298 ticket.

Cascade Merge with Bi-directional Associations

There are two bugs now that concern the use of cascade merge in combination with bi-directional associations. Make sure to study the behavior of cascade merge if you are using it:

- DDC-875 Merge can sometimes add the same entity twice into a collection
- DDC-763 Cascade merge on associated entities can insert too many rows through "Persistence by Reachability"

Custom Persisters

A Persister in Doctrine is an object that is responsible for the hydration and write operations of an entity against the database. Currently there is no way to overwrite the persister implementation for a given entity, however there are several use-cases that can benefit from custom persister implementations:

- Add Upsert Support
- Evaluate possible ways in which stored-procedures can be used
- The previous Filter Rules Feature Request

Persist Keys of Collections

PHP Arrays are ordered hash-maps and so should be the `Doctrine\Common\Collections\Collection` interface. We plan to evaluate a feature that optionally persists and hydrates the keys of a Collection instance.

Ticket DDC-213

Mapping many tables to one entity

It is not possible to map several equally looking tables onto one entity. For example if you have a production and an archive table of a certain business concept then you cannot have both tables map to the same entity.

Behaviors

Doctrine 2 will **never** include a behavior system like Doctrine 1 in the core library. We don't think behaviors add more value than they cost pain and debugging hell. Please see the many different blog posts we have written on this topics:

- Doctrine2 "Behaviors" in a Nutshell
- A re-usable Versionable behavior for Doctrine2
- Write your own ORM on top of Doctrine2
- Doctrine 2 Behavioral Extensions
- *Doctrator <https://github.com/pablodip/doctrator>*_

Doctrine 2 has enough hooks and extension points so that **you** can add whatever you want on top of it. None of this will ever become core functionality of Doctrine2 however, you will have to rely on third party extensions for magical behaviors.

Nested Set

NestedSet was offered as a behavior in Doctrine 1 and will not be included in the core of Doctrine 2. However there are already two extensions out there that offer support for Nested Set with Doctrine 2:

- Doctrine2 Hierarchical-Structural Behavior
- Doctrine2 NestedSet

Known Issues

The Known Issues section describes critical/blocker bugs and other issues that are either complicated to fix, not fixable due to backwards compatibility issues or where no simple fix exists (yet). We don't plan to add every bug in the tracker there, just those issues that can potentially cause nightmares or pain of any sort.

See the Open Bugs on Jira for more details on bugs, improvement and feature requests.

Identifier Quoting and Legacy Databases

For compatibility reasons between all the supported vendors and edge case problems Doctrine 2 does **NOT** do automatic identifier quoting. This can lead to problems when trying to get legacy-databases to work with Doctrine 2.

- You can quote column-names as described in the *Basic-Mapping* section.
- You cannot quote join column names.
- You cannot use non [a-zA-Z0-9_]+ characters, they will break several SQL statements.

Having problems with these kind of column names? Many databases support all CRUD operations on views that semantically map to certain tables. You can create views for all your problematic tables and column names to avoid the legacy quoting nightmare.

Microsoft SQL Server and Doctrine "datetime"

Doctrine assumes that you use `DateTime2` data-types. If your legacy database contains DateTime datatypes then you have to add your own data-type (see Basic Mapping for an example).

MySQL with MyISAM tables

Doctrine cannot provide atomic operations when calling `EntityManager#flush()` if one of the tables involved uses the storage engine MyISAM. You must use InnoDB or other storage engines that support transactions if you need integrity.

9.2.30 Pagination

New in version 2.2.

Starting with version 2.2 Doctrine ships with a Paginator for DQL queries. It has a very simple API and implements the SPL interfaces `Countable` and `IteratorAggregate`.

```php
<?php
use Doctrine\ORM\Tools\Pagination\Paginator;

$dql = "SELECT p, c FROM BlogPost p JOIN p.comments c";
$query = $entityManager->createQuery($dql)
                       ->setFirstResult(0)
                       ->setMaxResults(100);

$paginator = new Paginator($query, $fetchJoinCollection = true);

$c = count($paginator);
foreach ($paginator as $post) {
    echo $post->getHeadline() . "\n";
}
```

Paginating Doctrine queries is not as simple as you might think in the beginning. If you have complex fetch-join scenarios with one-to-many or many-to-many associations using the "default" LIMIT functionality of database vendors is not sufficient to get the correct results.

By default the pagination extension does the following steps to compute the correct result:

- Perform a Count query using *DISTINCT* keyword.
- Perform a Limit Subquery with *DISTINCT* to find all ids of the entity in from on the current page.
- Perform a WHERE IN query to get all results for the current page.

This behavior is only necessary if you actually fetch join a to-many collection. You can disable this behavior by setting the `$fetchJoinCollection` flag to `false`; in that case only 2 instead of the 3 queries described are executed. We hope to automate the detection for this in the future.

9.2.31 Filters

New in version 2.2.

Doctrine 2.2 features a filter system that allows the developer to add SQL to the conditional clauses of queries, regardless the place where the SQL is generated (e.g. from a DQL query, or by loading associated entities).

The filter functionality works on SQL level. Whether a SQL query is generated in a Persister, during lazy loading, in extra lazy collections or from DQL. Each time the system iterates over all the enabled filters, adding a new SQL part as a filter returns.

By adding SQL to the conditional clauses of queries, the filter system filters out rows belonging to the entities at the level of the SQL result set. This means that the filtered entities are never hydrated (which can be expensive).

Example filter class

Throughout this document the example `MyLocaleFilter` class will be used to illustrate how the filter feature works. A filter class must extend the base `Doctrine\ORM\Query\Filter\SQLFilter` class and implement the `addFilterConstraint` method. The method receives the `ClassMetadata` of the filtered entity and the table alias of the SQL table of the entity.

Note: In the case of joined or single table inheritance, you always get passed the ClassMetadata of the inheritance root. This is necessary to avoid edge cases that would break the SQL when applying the filters.

Parameters for the query should be set on the filter object by `SQLFilter#setParameter()`. Only parameters set via this function can be used in filters. The `SQLFilter#getParameter()` function takes care of the proper quoting of parameters.

```php
<?php
namespace Example;
use Doctrine\ORM\Mapping\ClassMetaData,
    Doctrine\ORM\Query\Filter\SQLFilter;

class MyLocaleFilter extends SQLFilter
{
    public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias)
    {
        // Check if the entity implements the LocalAware interface
        if (!$targetEntity->reflClass->implementsInterface('LocaleAware')) {
            return "";
        }
        return $targetTableAlias.'.locale = ' . $this->getParameter('locale'); // getParameter applie
    }
}
```

Configuration

Filter classes are added to the configuration as following:

```
<?php
$config->addFilter("locale", "\Doctrine\Tests\ORM\Functional\MyLocaleFilter");
```

The `Configuration#addFilter()` method takes a name for the filter and the name of the class responsible for the actual filtering.

Disabling/Enabling Filters and Setting Parameters

Filters can be disabled and enabled via the `FilterCollection` which is stored in the `EntityManager`. The `FilterCollection#enable($name)` method will retrieve the filter object. You can set the filter parameters on that object.

```
<?php
$filter = $em->getFilters()->enable("locale");
$filter->setParameter('locale', 'en');

// Disable it
$filter = $em->getFilters()->disable("locale");
```

Warning: Disabling and enabling filters has no effect on managed entities. If you want to refresh or reload an object after having modified a filter or the FilterCollection, then you should clear the EntityManager and re-fetch your entities, having the new rules for filtering applied.

9.2.32 Implementing a NamingStrategy

New in version 2.3.

Using a naming strategy you can provide rules for automatically generating database identifiers, columns and tables names when the table/column name is not given. This feature helps reduce the verbosity of the mapping document, eliminating repetitive noise (eg: `TABLE_`).

Configuring a naming strategy

The default strategy used by Doctrine is quite minimal.

By default the `Doctrine\ORM\Mapping\DefaultNamingStrategy` uses the simple class name and the attributes names to generate tables and columns

You can specify a different strategy by calling `Doctrine\ORM\Configuration#setNamingStrategy()` :

```
<?php
$namingStrategy = new MyNamingStrategy();
$configuration()->setNamingStrategy($namingStrategy);
```

Underscore naming strategy

\Doctrine\ORM\Mapping\UnderscoreNamingStrategy is a built-in strategy that might be a useful if you want to use a underlying convention.

```php
<?php
$namingStrategy = new \Doctrine\ORM\Mapping\UnderscoreNamingStrategy(CASE_UPPER);
$configuration()->setNamingStrategy($namingStrategy);
```

Then SomeEntityName will generate the table SOME_ENTITY_NAME when CASE_UPPER or some_entity_name using CASE_LOWER is given.

Naming strategy interface

The interface Doctrine\ORM\Mapping\NamingStrategy allows you to specify a "naming standard" for database tables and columns.

```php
<?php
/**
 * Return a table name for an entity class
 *
 * @param string $className The fully-qualified class name
 * @return string A table name
 */
function classToTableName($className);

/**
 * Return a column name for a property
 *
 * @param string $propertyName A property
 * @return string A column name
 */
function propertyToColumnName($propertyName);

/**
 * Return the default reference column name
 *
 * @return string A column name
 */
function referenceColumnName();

/**
 * Return a join column name for a property
 *
 * @param string $propertyName A property
 * @return string A join column name
 */
function joinColumnName($propertyName, $className = null);

/**
 * Return a join table name
 *
 * @param string $sourceEntity The source entity
 * @param string $targetEntity The target entity
 * @param string $propertyName A property
 * @return string A join table name
 */
function joinTableName($sourceEntity, $targetEntity, $propertyName = null);
```

```
/**
 * Return the foreign key column name for the given parameters
 *
 * @param string $entityName A entity
 * @param string $referencedColumnName A property
 * @return string A join column name
 */
function joinKeyColumnName($entityName, $referencedColumnName = null);
```

Implementing a naming strategy

If you have database naming standards like all tables names should be prefixed by the application prefix, all column names should be upper case, you can easily achieve such standards by implementing a naming strategy. You need to implements NamingStrategy first. Following is an example

```
<?php
class MyAppNamingStrategy implements NamingStrategy
{
    public function classToTableName($className)
    {
        return 'MyApp_' . substr($className, strrpos($className, '\\') + 1);
    }
    public function propertyToColumnName($propertyName)
    {
        return $propertyName;
    }
    public function referenceColumnName()
    {
        return 'id';
    }
    public function joinColumnName($propertyName, $className = null)
    {
        return $propertyName . '_' . $this->referenceColumnName();
    }
    public function joinTableName($sourceEntity, $targetEntity, $propertyName = null)
    {
        return strtolower($this->classToTableName($sourceEntity) . '_' .
                $this->classToTableName($targetEntity));
    }
    public function joinKeyColumnName($entityName, $referencedColumnName = null)
    {
        return strtolower($this->classToTableName($entityName) . '_' .
                ($referencedColumnName ?: $this->referenceColumnName()));
    }
}
```

Configuring the namingstrategy is easy if. Just set your naming strategy calling `Doctrine\ORM\Configuration#setNamingStrategy()` :.

```
<?php
$namingStrategy = new MyAppNamingStrategy();
$configuration()->setNamingStrategy($namingStrategy);
```

9.2.33 Installation

The installation chapter has moved to Installation and Configuration.

9.2.34 Advanced Configuration

The configuration of the EntityManager requires a `Doctrine\ORM\Configuration` instance as well as some database connection parameters. This example shows all the potential steps of configuration.

```php
<?php
use Doctrine\ORM\EntityManager,
    Doctrine\ORM\Configuration;

// ...

if ($applicationMode == "development") {
    $cache = new \Doctrine\Common\Cache\ArrayCache;
} else {
    $cache = new \Doctrine\Common\Cache\ApcCache;
}

$config = new Configuration;
$config->setMetadataCacheImpl($cache);
$driverImpl = $config->newDefaultAnnotationDriver('/path/to/lib/MyProject/Entities');
$config->setMetadataDriverImpl($driverImpl);
$config->setQueryCacheImpl($cache);
$config->setProxyDir('/path/to/myproject/lib/MyProject/Proxies');
$config->setProxyNamespace('MyProject\Proxies');

if ($applicationMode == "development") {
    $config->setAutoGenerateProxyClasses(true);
} else {
    $config->setAutoGenerateProxyClasses(false);
}

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

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

Note: Do not use Doctrine without a metadata and query cache! Doctrine is 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. 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.

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();
```

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

There are currently 4 available implementations:

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

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`.

The annotation driver can be configured with a factory method on the `Doctrine\ORM\Configuration`:

```
<?php
$driverImpl = $config->newDefaultAnnotationDriver('/path/to/lib/MyProject/Entities');
$config->setMetadataDriverImpl($driverImpl);
```

The path information to the entities is required for the annotation driver, because otherwise mass-operations on all entities through the console could not work correctly. All of metadata drivers accept either a single directory as a string or an array of directories. With this feature a single driver can support multiple directories of Entities.

Metadata Cache (*RECOMMENDED*)

```
<?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 for production are:

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

For development you should use the `Doctrine\Common\Cache\ArrayCache` which only caches data on a per-request basis.

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 for production are:

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

For development you should use the `Doctrine\Common\Cache\ArrayCache` which only caches data on a per-request basis.

SQL Logger (*Optional*)

```php
<?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*)

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

```php
<?php
$config->setAutoGenerateProxyClasses($mode);
```

Possible values for `$mode` are:

- `Doctrine\Common\Proxy\AbstractProxyFactory::AUTOGENERATE_NEVER`

Never autogenerate a proxy. You will need to generate the proxies manually, for this use the Doctrine Console like so:

```
$ ./doctrine orm:generate-proxies
```

When you do this in a development environment, be aware that you may get class/file not found errors if certain proxies are not yet generated. You may also get 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 Console to (re)generate the proxy classes.

- `Doctrine\Common\Proxy\AbstractProxyFactory::AUTOGENERATE_ALWAYS`

Always generates a new proxy in every request and writes it to disk.

- `Doctrine\Common\Proxy\AbstractProxyFactory::AUTOGENERATE_FILE_NOT_EXISTS`

Generate the proxy class when the proxy file does not exist. This strategy causes a file exists call whenever any proxy is used the first time in a request.

- `Doctrine\Common\Proxy\AbstractProxyFactory::AUTOGENERATE_EVAL`

Generate the proxy classes and evaluate them on the fly via eval(), avoiding writing the proxies to disk. This strategy is only sane for development.

In a production environment, it is highly recommended to use AUTOGENERATE_NEVER to allow for optimal performances. The other options are interesting in development environment.

Before v2.4, `setAutoGenerateProxyClasses` would accept a boolean value. This is still possible, `FALSE` being equivalent to AUTOGENERATE_NEVER and `TRUE` to AUTOGENERATE_ALWAYS.

Development vs Production Configuration

You should code your Doctrine2 bootstrapping with two different runtime models in mind. There are some serious benefits of using APC or Memcache in production. In development however this will frequently give you fatal errors, when you change your entities and the cache still keeps the outdated metadata. That is why we recommend the `ArrayCache` for development.

Furthermore you should have the Auto-generating Proxy Classes option to true in development and to false in production. If this option is set to `TRUE` it can seriously hurt your script performance if several proxy classes are re-generated during script execution. Filesystem calls of that magnitude can even slower than all the database queries Doctrine issues. Additionally writing a proxy sets an exclusive file lock which can cause serious performance bottlenecks in systems with regular concurrent requests.

Connection Options

The `$connectionOptions` passed as the first argument to `EntityManager::create()` has to be either an array or an instance of `Doctrine\DBAL\Connection`. If an array is passed it is directly passed along to the DBAL Factory `Doctrine\DBAL\DriverManager::getConnection()`. The DBAL configuration is explained in the DBAL section.

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
<?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.

Note: 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

In a production environment, it is highly recommended to use `AUTOGENERATE_NEVER` to allow for optimal performances. However you will be required to generate the proxies manually using the Doctrine Console:

```
$ ./doctrine orm:generate-proxies
```

The other options are interesting in development environment:

- `AUTOGENERATE_ALWAYS` will require you to create and configure a proxy directory. Proxies will be generated and written to file on each request, so any modification to your code will be acknowledged.

- AUTOGENERATE_FILE_NOT_EXISTS will not overwrite an existing proxy file. If your code changes, you will need to regenerate the proxies manually.
- AUTOGENERATE_EVAL will regenerate each proxy on each request, but without writing them to disk.

Autoloading Proxies

When you deserialize proxy objects from the session or any other storage it is necessary to have an autoloading mechanism in place for these classes. For implementation reasons Proxy class names are not PSR-0 compliant. This means that you have to register a special autoloader for these classes:

```php
<?php
use Doctrine\ORM\Proxy\Autoloader;

$proxyDir = "/path/to/proxies";
$proxyNamespace = "MyProxies";

Autoloader::register($proxyDir, $proxyNamespace);
```

If you want to execute additional logic to intercept the proxy file not found state you can pass a closure as the third argument. It will be called with the arguments proxydir, namespace and className when the proxy file could not be found.

Multiple Metadata Sources

When using different components using Doctrine 2 you may end up with them using two different metadata drivers, for example XML and YAML. You can use the DriverChain Metadata implementations to aggregate these drivers based on namespaces:

```php
<?php
use Doctrine\ORM\Mapping\Driver\DriverChain;

$chain = new DriverChain();
$chain->addDriver($xmlDriver, 'Doctrine\Tests\Models\Company');
$chain->addDriver($yamlDriver, 'Doctrine\Tests\ORM\Mapping');
```

Based on the namespace of the entity the loading of entities is delegated to the appropriate driver. The chain semantics come from the fact that the driver loops through all namespaces and matches the entity class name against the namespace using a `strpos() === 0` call. This means you need to order the drivers correctly if sub-namespaces use different metadata driver implementations.

Default Repository (*OPTIONAL*)

Specifies the FQCN of a subclass of the EntityRepository. That will be available for all entities without a custom repository class.

```php
<?php
$config->setDefaultRepositoryClassName($fqcn);
$config->getDefaultRepositoryClassName();
```

The default value is `Doctrine\ORM\EntityRepository`. Any repository class must be a subclass of EntityRepository otherwise you got an ORMException

Setting up the Console

Doctrine uses the Symfony Console component for generating the command line interface. You can take a look at the `vendor/bin/doctrine.php` script and the `Doctrine\ORM\Tools\Console\ConsoleRunner` command for inspiration how to setup the cli.

In general the required code looks like this:

```
<?php
$cli = new Application('Doctrine Command Line Interface', \Doctrine\ORM\Version::VERSION);
$cli->setCatchExceptions(true);
$cli->setHelperSet($helperSet);
Doctrine\ORM\Tools\Console\ConsoleRunner::addCommands($cli);
$cli->run();
```

9.2.35 The Second Level Cache

Note: The second level cache functionality is marked as experimental for now. It is a very complex feature and we cannot guarantee yet that it works stable in all cases.

The Second Level Cache is designed to reduce the amount of necessary database access. It sits between your application and the database to avoid the number of database hits as much as possible.

When turned on, entities will be first searched in cache and if they are not found, a database query will be fired an then the entity result will be stored in a cache provider.

There are some flavors of caching available, but is better to cache read-only data.

Be aware that caches are not aware of changes made to the persistent store by another application. They can, however, be configured to regularly expire cached data.

Caching Regions

Second level cache does not store instances of an entity, instead it caches only entity identifier and values. Each entity class, collection association and query has its region, where values of each instance are stored.

Caching Regions are specific region into the cache provider that might store entities, collection or queries. Each cache region resides in a specific cache namespace and has its own lifetime configuration.

Notice that when caching collection and queries only identifiers are stored. The entity values will be stored in its own region

Something like below for an entity region :

```
<?php
[
  'region_name:entity_1_hash' => ['id'=> 1, 'name' => 'FooBar', 'associationName'=>null],
  'region_name:entity_2_hash' => ['id'=> 2, 'name' => 'Foo', 'associationName'=>['id'=>11]],
  'region_name:entity_3_hash' => ['id'=> 3, 'name' => 'Bar', 'associationName'=>['id'=>22]]
];
```

If the entity holds a collection that also needs to be cached. An collection region could look something like :

```
<?php
[
  'region_name:entity_1_coll_assoc_name_hash' => ['ownerId'=> 1, 'list' => [1, 2, 3]],
  'region_name:entity_2_coll_assoc_name_hash' => ['ownerId'=> 2, 'list' => [2, 3]],
  'region_name:entity_3_coll_assoc_name_hash' => ['ownerId'=> 3, 'list' => [2, 4]]
];
```

A query region might be something like :

```
<?php
[
  'region_name:query_1_hash' => ['list' => [1, 2, 3]],
  'region_name:query_2_hash' => ['list' => [2, 3]],
  'region_name:query_3_hash' => ['list' => [2, 4]]
];
```

Note: The following data structures represents now the cache will looks like, this is not actual cached data.

Cache Regions

`Doctrine\ORM\Cache\Region\DefaultRegion` It's the default implementation. A simplest cache region compatible with all doctrine-cache drivers but does not support locking.

`Doctrine\ORM\Cache\Region` and `Doctrine\ORM\Cache\ConcurrentRegion` Defines contracts that should be implemented by a cache provider.

It allows you to provide your own cache implementation that might take advantage of specific cache driver.

If you want to support locking for `READ_WRITE` strategies you should implement `ConcurrentRegion`; `CacheRegion` otherwise.

Cache region

Defines a contract for accessing a particular region.

`Doctrine\ORM\Cache\Region`

Defines a contract for accessing a particular cache region.

See API Doc.

Concurrent cache region

A `Doctrine\ORM\Cache\ConcurrentRegion` is designed to store concurrently managed data region. By default, Doctrine provides a very simple implementation based on file locks `Doctrine\ORM\Cache\Region\FileLockRegion`.

If you want to use an `READ_WRITE` cache, you should consider providing your own cache region.

`Doctrine\ORM\Cache\ConcurrentRegion`

Defines contract for concurrently managed data region.

See API Doc.

Timestamp region

`Doctrine\ORM\Cache\TimestampRegion`

Tracks the timestamps of the most recent updates to particular entity.

See API Doc.

Caching mode

- `READ_ONLY` (DEFAULT)
  - Can do reads, inserts and deletes, cannot perform updates or employ any locks.
  - Useful for data that is read frequently but never updated.
  - Best performer.
  - It is Simple.
- `NONSTRICT_READ_WRITE`
  - Read Write Cache doesn't employ any locks but can do reads, inserts, updates and deletes.
  - Good if the application needs to update data rarely.
- `READ_WRITE`
  - Read Write cache employs locks before update/delete.
  - Use if data needs to be updated.
  - Slowest strategy.
  - To use it a the cache region implementation must support locking.

Built-in cached persisters

Cached persisters are responsible to access cache regions.

Cache Usage	Persister
READ_ONLY	Doctrine\ORM\Cache\Persister\ReadOnlyCachedEntityPersister
READ_WRITE	Doctrine\ORM\Cache\Persister\ReadWriteCachedEntityPersister
NON-STRICT_READ_WRITE	Doc-trine\ORM\Cache\Persister\NonStrictReadWriteCachedEntityPersister
READ_ONLY	Doctrine\ORM\Cache\Persister\ReadOnlyCachedCollectionPersister
READ_WRITE	Doctrine\ORM\Cache\Persister\ReadWriteCachedCollectionPersister
NON-STRICT_READ_WRITE	Doc-trine\ORM\Cache\Persister\NonStrictReadWriteCacheCollectionPersister

Configuration

Doctrine allows you to specify configurations and some points of extension for the second-level-cache

Enable Second Level Cache

To enable the second-level-cache, you should provide a cache factory `\Doctrine\ORM\Cache\DefaultCacheFactory` is the default implementation.

```php
<?php
/* @var $config \Doctrine\ORM\Cache\RegionsConfiguration */
/* @var $cache \Doctrine\Common\Cache\Cache */

$factory = new \Doctrine\ORM\Cache\DefaultCacheFactory($config, $cache);
```

```
// Enable second-level-cache
$config->setSecondLevelCacheEnabled();

// Cache factory
$config->getSecondLevelCacheConfiguration()
    ->setCacheFactory($factory);
```

Cache Factory

Cache Factory is the main point of extension.

It allows you to provide a specific implementation of the following components :

- `QueryCache` Store and retrieve query cache results.
- `CachedEntityPersister` Store and retrieve entity results.
- `CachedCollectionPersister` Store and retrieve query results.
- `EntityHydrator` Transform an entity into a cache entry and cache entry into entities
- `CollectionHydrator` Transform a collection into a cache entry and cache entry into collection

See API Doc.

Region Lifetime

To specify a default lifetime for all regions or specify a different lifetime for a specific region.

```
<?php
/* @var $config \Doctrine\ORM\Configuration */
/* @var $cacheConfig \Doctrine\ORM\Configuration */
$cacheConfig  =  $config->getSecondLevelCacheConfiguration();
$regionConfig =  $cacheConfig->getRegionsConfiguration();

// Cache Region lifetime
$regionConfig->setLifetime('my_entity_region', 3600);   // Time to live for a specific region; In sec
$regionConfig->setDefaultLifetime(7200);                // Default time to live; In seconds
```

Cache Log

By providing a cache logger you should be able to get information about all cache operations such as hits, misses and puts.

`\Doctrine\ORM\Cache\Logging\StatisticsCacheLogger` is a built-in implementation that provides basic statistics.

```
<?php
/* @var $config \Doctrine\ORM\Configuration */
$logger = \Doctrine\ORM\Cache\Logging\StatisticsCacheLogger();

// Cache logger
$config->setSecondLevelCacheEnabled(true);
$config->getSecondLevelCacheConfiguration()
    ->setCacheLogger($logger);
```

```
// Collect cache statistics

// Get the number of entries successfully retrieved from a specific region.
$logger->getRegionHitCount('my_entity_region');

// Get the number of cached entries *not* found in a specific region.
$logger->getRegionMissCount('my_entity_region');

// Get the number of cacheable entries put in cache.
$logger->getRegionPutCount('my_entity_region');

// Get the total number of put in all regions.
$logger->getPutCount();

// Get the total number of entries successfully retrieved from all regions.
$logger->getHitCount();

// Get the total number of cached entries *not* found in all regions.
$logger->getMissCount();
```

If you want to get more information you should implement `\Doctrine\ORM\Cache\Logging\CacheLogger`. and collect all information you want.

See API Doc.

Entity cache definition

- Entity cache configuration allows you to define the caching strategy and region for an entity.
  - `usage` Specifies the caching strategy: `READ_ONLY`, `NONSTRICT_READ_WRITE`, `READ_WRITE`. see *Caching mode*
  - `region` Optional value that specifies the name of the second level cache region.
- *PHP*

```
<?php
/**
 * @Entity
 * @Cache(usage="READ_ONLY", region="my_entity_region")
 */
class Country
{
    /**
     * @Id
     * @GeneratedValue
     * @Column(type="integer")
     */
    protected $id;

    /**
     * @Column(unique=true)
     */
    protected $name;

    // other properties and methods
}
```

- *XML*

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping" xmlns:xsi="ht
  <entity name="Country">
    <cache usage="READ_ONLY" region="my_entity_region" />
    <id name="id" type="integer" column="id">
      <generator strategy="IDENTITY"/>
    </id>
    <field name="name" type="string" column="name"/>
  </entity>
</doctrine-mapping>
```

- *YAML*

```
Country:
  type: entity
  cache:
    usage : READ_ONLY
    region : my_entity_region
  id:
    id:
      type: integer
      id: true
      generator:
        strategy: IDENTITY
  fields:
    name:
      type: string
```

Association cache definition

The most common use case is to cache entities. But we can also cache relationships. It caches the primary keys of association and cache each element will be cached into its region.

- *PHP*

```
<?php
/**
 * @Entity
 * @Cache("NONSTRICT_READ_WRITE")
 */
class State
{
    /**
     * @Id
     * @GeneratedValue
     * @Column(type="integer")
     */
    protected $id;

    /**
     * @Column(unique=true)
     */
    protected $name;

    /**
     * @Cache("NONSTRICT_READ_WRITE")
     * @ManyToOne(targetEntity="Country")
     * @JoinColumn(name="country_id", referencedColumnName="id")
     */
    protected $country;

    /**
     * @Cache("NONSTRICT_READ_WRITE")
     * @OneToMany(targetEntity="City", mappedBy="state")
     */
    protected $cities;

    // other properties and methods
}
```

- *XML*

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping" xmlns:xsi="ht
  <entity name="State">

    <cache usage="NONSTRICT_READ_WRITE" />

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

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

    <many-to-one field="country" target-entity="Country">
      <cache usage="NONSTRICT_READ_WRITE" />

      <join-columns>
        <join-column name="country_id" referenced-column-name="id"/>
      </join-columns>
    </many-to-one>

    <one-to-many field="cities" target-entity="City" mapped-by="state">
      <cache usage="NONSTRICT_READ_WRITE"/>
    </one-to-many>
  </entity>
</doctrine-mapping>
```

- *YAML*

```
State:
  type: entity
  cache:
    usage : NONSTRICT_READ_WRITE
  id:
    id:
      type: integer
      id: true
      generator:
        strategy: IDENTITY
  fields:
    name:
      type: string

  manyToOne:
    state:
      targetEntity: Country
      joinColumns:
        country_id:
          referencedColumnName: id
      cache:
        usage : NONSTRICT_READ_WRITE

  oneToMany:
    cities:
      targetEntity:City
      mappedBy: state
      cache:
        usage : NONSTRICT_READ_WRITE
```

> Note: for this to work, the target entity must also be marked as cacheable.

Cache usage

Basic entity cache

```php
<?php
$em->persist(new Country($name));
$em->flush();                         // Hit database to insert the row and put into cache

$em->clear();                         // Clear entity manager

$country1  = $em->find('Country', 1); // Retrieve item from cache

$country->setName("New Name");
$em->persist($country);
$em->flush();                         // Hit database to update the row and update cache

$em->clear();                         // Clear entity manager

$country2  = $em->find('Country', 1); // Retrieve item from cache
                                      // Notice that $country1 and $country2 are not the same instanc
```

Association cache

```php
<?php
// Hit database to insert the row and put into cache
$em->persist(new State($name, $country));
$em->flush();

// Clear entity manager
$em->clear();

// Retrieve item from cache
$state = $em->find('State', 1);

// Hit database to update the row and update cache entry
$state->setName("New Name");
$em->persist($state);
$em->flush();

// Create a new collection item
$city = new City($name, $state);
```

```
$state->addCity($city);

// Hit database to insert new collection item,
// put entity and collection cache into cache.
$em->persist($city);
$em->persist($state);
$em->flush();

// Clear entity manager
$em->clear();

// Retrieve item from cache
$state = $em->find('State', 1);

// Retrieve association from cache
$country = $state->getCountry();

// Retrieve collection from cache
$cities = $state->getCities();

echo $country->getName();
echo $state->getName();

// Retrieve each collection item from cache
foreach ($cities as $city) {
    echo $city->getName();
}
```

Note: Notice that all entities should be marked as cacheable.

Using the query cache

The second level cache stores the entities, associations and collections. The query cache stores the results of the query but as identifiers, entity values are actually stored in the 2nd level cache.

Note: Query cache should always be used in conjunction with the second-level-cache for those entities which should be cached.

```
<?php
/* @var $em \Doctrine\ORM\EntityManager */

// Execute database query, store query cache and entity cache
$result1 = $em->createQuery('SELECT c FROM Country c ORDER BY c.name')
    ->setCacheable(true)
    ->getResult();

$em->clear()

// Check if query result is valid and load entities from cache
$result2 = $em->createQuery('SELECT c FROM Country c ORDER BY c.name')
    ->setCacheable(true)
    ->getResult();
```

Cache mode

The Cache Mode controls how a particular query interacts with the second-level cache:

- `Cache::MODE_GET` - May read items from the cache, but will not add items.
- `Cache::MODE_PUT` - Will never read items from the cache, but will add items to the cache as it reads them from the database.
- `Cache::MODE_NORMAL` - May read items from the cache, and add items to the cache.
- `Cache::MODE_REFRESH` - The query will never read items from the cache, but will refresh items to the cache as it reads them from the database.

```php
<?php
/* @var $em \Doctrine\ORM\EntityManager */
// Will refresh the query cache and all entities the cache as it reads from the database.
$result1 = $em->createQuery('SELECT c FROM Country c ORDER BY c.name')
    ->setCacheMode(Cache::MODE_GET)
    ->setCacheable(true)
    ->getResult();
```

Note: The the default query cache mode is `` `Cache::MODE_NORMAL` ``

DELETE / UPDATE queries

DQL UPDATE / DELETE statements are ported directly into a database and bypass the second-level cache, Entities that are already cached will NOT be invalidated. However the cached data could be evicted using the cache API or an special query hint.

Execute the `UPDATE` and invalidate `all cache entries` using `Query::HINT_CACHE_EVICT`

```php
<?php
// Execute and invalidate
$this->_em->createQuery("UPDATE Entity\Country u SET u.name = 'unknown' WHERE u.id = 1")
    ->setHint(Query::HINT_CACHE_EVICT, true)
    ->execute();
```

Execute the `UPDATE` and invalidate `all cache entries` using the cache API

```php
<?php
// Execute
$this->_em->createQuery("UPDATE Entity\Country u SET u.name = 'unknown' WHERE u.id = 1")
    ->execute();
// Invoke Cache API
$em->getCache()->evictEntityRegion('Entity\Country');
```

Execute the `UPDATE` and invalidate `a specific cache entry` using the cache API

```php
<?php
// Execute
$this->_em->createQuery("UPDATE Entity\Country u SET u.name = 'unknown' WHERE u.id = 1")
    ->execute();
// Invoke Cache API
$em->getCache()->evictEntity('Entity\Country', 1);
```

Using the repository query cache

As well as `Query Cache` all persister queries store only identifier values for an individual query. All persister use a single timestamps cache region keeps track of the last update for each persister, When a query is loaded from cache, the timestamp region is checked for the last update for that persister. Using the last update timestamps as part of the query key invalidate the cache key when an update occurs.

```php
<?php
// load from database and store cache query key hashing the query + parameters + last timestamp cache
$entities   = $em->getRepository('Entity\Country')->findAll();

// load from query and entities from cache..
$entities   = $em->getRepository('Entity\Country')->findAll();

// update the timestamp cache region for Country
$em->persist(new Country('zombieland'));
$em->flush();
$em->clear();

// Reload from database.
// At this point the query cache key if not logger valid, the select goes straight
$entities   = $em->getRepository('Entity\Country')->findAll();
```

Cache API

Caches are not aware of changes made by another application. However, you can use the cache API to check / invalidate cache entries.

```php
<?php
/* @var $cache \Doctrine\ORM\Cache */
$cache = $em->getCache();

$cache->containsEntity('Entity\State', 1)      // Check if the cache exists
$cache->evictEntity('Entity\State', 1);        // Remove an entity from cache
$cache->evictEntityRegion('Entity\State');     // Remove all entities from cache

$cache->containsCollection('Entity\State', 'cities', 1);   // Check if the cache exists
$cache->evictCollection('Entity\State', 'cities', 1);      // Remove an entity collection from cache
$cache->evictCollectionRegion('Entity\State', 'cities');   // Remove all collections from cache
```

Limitations

Composite primary key

Composite primary key are supported by second level cache, however when one of the keys is an association the cached entity should always be retrieved using the association identifier. For performance reasons the cache API does not extract from composite primary key.

```php
<?php
/**
 * @Entity
 */
class Reference
{
    /**
```

```
     * @Id
     * @ManyToOne(targetEntity="Article", inversedBy="references")
     * @JoinColumn(name="source_id", referencedColumnName="article_id")
     */
    private $source;

    /**
     * @Id
     * @ManyToOne(targetEntity="Article")
     * @JoinColumn(name="target_id", referencedColumnName="article_id")
     */
    private $target;
}

// Supported
/* @var $article Article */
$article = $em->find('Article', 1);

// Supported
/* @var $article Article */
$article = $em->find('Article', $article);

// Supported
$id        = array('source' => 1, 'target' => 2);
$reference = $em->find('Reference', $id);

// NOT Supported
$id        = array('source' => new Article(1), 'target' => new Article(2));
$reference = $em->find('Reference', $id);
```

Distributed environments

Some cache driver are not meant to be used in a distributed environment. Load-balancer for distributing workloads across multiple computing resources should be used in conjunction with distributed caching system such as memcached, redis, riak ...

Caches should be used with care when using a load-balancer if you don't share the cache. While using APC or any file based cache update occurred in a specific machine would not reflect to the cache in other machines.

Paginator

Count queries generated by `Doctrine\ORM\Tools\Pagination\Paginator` are not cached by second-level cache. Although entities and query result are cached count queries will hit the database every time.

9.2.36 Security

The Doctrine library is operating very close to your database and as such needs to handle and make assumptions about SQL injection vulnerabilities.

It is vital that you understand how Doctrine approaches security, because we cannot protect you from SQL injection.

Please also read the documentation chapter on Security in Doctrine DBAL. This page only handles Security issues in the ORM.

- [DBAL Security Page](https://github.com/doctrine/dbal/blob/master/docs/en/reference/security.rst)

If you find a Security bug in Doctrine, please report it on Jira and change the Security Level to “Security Issues”. It will be visible to Doctrine Core developers and you only.

User input and Doctrine ORM

The ORM is much better at protecting against SQL injection than the DBAL alone. You can consider the following APIs to be safe from SQL injection:

- `\Doctrine\ORM\EntityManager#find()` and `getReference()`.
- All values on Objects inserted and updated through `Doctrine\ORM\EntityManager#persist()`
- All find methods on `Doctrine\ORM\EntityRepository`.
- **User Input set to DQL Queries or QueryBuilder methods through**
  - `setParameter()` or variants
  - `setMaxResults()`
  - `setFirstResult()`
- Queries through the Criteria API on `Doctrine\ORM\PersistentCollection` and `Doctrine\ORM\EntityRepository`.

You are **NOT** save from SQL injection when using user input with:

- Expression API of `Doctrine\ORM\QueryBuilder`
- Concatenating user input into DQL SELECT, UPDATE or DELETE statements or Native SQL.

This means SQL injections can only occur with Doctrine ORM when working with Query Objects of any kind. The safe rule is to always use prepared statement parameters for user objects when using a Query object.

Warning: Insecure code follows, don’t copy paste this.

The following example shows insecure DQL usage:

```
<?php

// INSECURE
$dql = "SELECT u
          FROM MyProject\Entity\User u
         WHERE u.status = '" .  $_GET['status'] . "'
     ORDER BY " . $_GET['orderField'] . " ASC";
```

For Doctrine there is absolutely no way to find out which parts of `$dql` are from user input and which are not, even if we have our own parsing process this is technically impossible. The correct way is:

```
<?php

$orderFieldWhitelist = array('email', 'username');
$orderField = "email";

if (in_array($_GET['orderField'], $orderFieldWhitelist)) {
    $orderField = $_GET['orderField'];
}

$dql = "SELECT u
          FROM MyProject\Entity\User u
         WHERE u.status = ?1
```

```
     ORDER BY u." . $orderField . " ASC";

$query = $entityManager->createQuery($dql);
$query->setParameter(1, $_GET['status']);
```

Preventing Mass Assignment Vulnerabilities

ORMs are very convenient for CRUD applications and Doctrine is no exception. However CRUD apps are often vulnerable to mass assignment security problems when implemented naively.

Doctrine is not vulnerable to this problem out of the box, but you can easily make your entities vulnerable to mass assignment when you add methods of the kind `updateFromArray()` or `updateFromJson()` to them. A vulnerable entity might look like this:

```php
<?php

/**
 * @Entity
 */
class InsecureEntity
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;
    /** @Column */
    private $email;
    /** @Column(type="boolean") */
    private $isAdmin;

    public function fromArray(array $userInput)
    {
        foreach ($userInput as $key => $value) {
            $this->$key = $value;
        }
    }
}
```

Now the possiblity of mass-asignment exists on this entity and can be exploitet by attackers to set the “isAdmin” flag to true on any object when you pass the whole request data to this method like:

```php
<?php
$entity = new InsecureEntity();
$entity->fromArray($_POST);

$entityManager->persist($entity);
$entityManager->flush();
```

You can spot this problem in this very simple example easily. However in combination with frameworks and form libraries it might not be so obvious when this issue arises. Be careful to avoid this kind of mistake.

How to fix this problem? You should always have a whitelist of allowed key to set via mass assignment functions.

```
public function fromArray(array $userInput, $allowedFields = array())
{
    foreach ($userInput as $key => $value) {
        if (in_array($key, $allowedFields)) {
            $this->$key = $value;
        }
    }
}
```

9.3 Cookbook

9.3.1 Aggregate Fields

Section author: Benjamin Eberlei <kontakt@beberlei.de>

You will often come across the requirement to display aggregate values of data that can be computed by using the MIN, MAX, COUNT or SUM SQL functions. For any ORM this is a tricky issue traditionally. Doctrine 2 offers several ways to get access to these values and this article will describe all of them from different perspectives.

You will see that aggregate fields can become very explicit features in your domain model and how this potentially complex business rules can be easily tested.

An example model

Say you want to model a bank account and all their entries. Entries into the account can either be of positive or negative money values. Each account has a credit limit and the account is never allowed to have a balance below that value.

For simplicity we live in a world were money is composed of integers only. Also we omit the receiver/sender name, stated reason for transfer and the execution date. These all would have to be added on the `Entry` object.

Our entities look like:

```php
<?php
namespace Bank\Entities;

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

    /** @Column(type="string", unique=true) */
    private $no;

    /**
     * @OneToMany(targetEntity="Entry", mappedBy="account", cascade={"persist"})
     */
    private $entries;

    /**
     * @Column(type="integer")
     */
    private $maxCredit = 0;

    public function __construct($no, $maxCredit = 0)
    {
        $this->no = $no;
        $this->maxCredit = $maxCredit;
        $this->entries = new \Doctrine\Common\Collections\ArrayCollection();
    }
}

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

    /**
     * @ManyToOne(targetEntity="Account", inversedBy="entries")
     */
    private $account;

    /**
     * @Column(type="integer")
     */
    private $amount;

    public function __construct($account, $amount)
    {
        $this->account = $account;
        $this->amount = $amount;
        // more stuff here, from/to whom, stated reason, execution date and such
    }

    public function getAmount()
    {
        return $this->amount;
    }
}
```

Using DQL

The Doctrine Query Language allows you to select for aggregate values computed from fields of your Domain Model. You can select the current balance of your account by calling:

```
<?php
$dql = "SELECT SUM(e.amount) AS balance FROM Bank\Entities\Entry e " .
       "WHERE e.account = ?1";
$balance = $em->createQuery($dql)
              ->setParameter(1, $myAccountId)
              ->getSingleScalarResult();
```

The `$em` variable in this (and forthcoming) example holds the Doctrine `EntityManager`. We create a query for the SUM of all amounts (negative amounts are withdraws) and retrieve them as a single scalar result, essentially return only the first column of the first row.

This approach is simple and powerful, however it has a serious drawback. We have to execute a specific query for the balance whenever we need it.

To implement a powerful domain model we would rather have access to the balance from our `Account` entity during all times (even if the Account was not persisted in the database before!).

Also an additional requirement is the max credit per `Account` rule.

We cannot reliably enforce this rule in our `Account` entity with the DQL retrieval of the balance. There are many different ways to retrieve accounts. We cannot guarantee that we can execute the aggregation query for all these use-cases, let alone that a userland programmer checks this balance against newly added entries.

Using your Domain Model

`Account` and all the `Entry` instances are connected through a collection, which means we can compute this value at runtime:

```php
<?php
class Account
{
    // .. previous code
    public function getBalance()
    {
        $balance = 0;
        foreach ($this->entries as $entry) {
            $balance += $entry->getAmount();
        }
        return $balance;
    }
}
```

Now we can always call `Account::getBalance()` to access the current account balance.

To enforce the max credit rule we have to implement the "Aggregate Root" pattern as described in Eric Evans book on Domain Driven Design. Described with one sentence, an aggregate root controls the instance creation, access and manipulation of its children.

In our case we want to enforce that new entries can only added to the `Account` by using a designated method. The `Account` is the aggregate root of this relation. We can also enforce the correctness of the bi-directional `Account` <-> `Entry` relation with this method:

```php
<?php
class Account
{
    public function addEntry($amount)
    {
        $this->assertAcceptEntryAllowed($amount);

        $e = new Entry($this, $amount);
        $this->entries[] = $e;
        return $e;
    }
}
```

Now look at the following test-code for our entities:

```php
<?php
class AccountTest extends \PHPUnit_Framework_TestCase
{
    public function testAddEntry()
    {
        $account = new Account("123456", $maxCredit = 200);
        $this->assertEquals(0, $account->getBalance());

        $account->addEntry(500);
        $this->assertEquals(500, $account->getBalance());
        $account->addEntry(-700);
        $this->assertEquals(-200, $account->getBalance());
    }

    public function testExceedMaxLimit()
    {
        $account = new Account("123456", $maxCredit = 200);

        $this->setExpectedException("Exception");
        $account->addEntry(-1000);
    }
}
```

To enforce our rule we can now implement the assertion in `Account::addEntry`:

```
<?php
class Account
{
    private function assertAcceptEntryAllowed($amount)
    {
        $futureBalance = $this->getBalance() + $amount;
        $allowedMinimalBalance = ($this->maxCredit * -1);
        if ($futureBalance < $allowedMinimalBalance) {
            throw new Exception("Credit Limit exceeded, entry is not allowed!");
        }
    }
}
```

We haven't talked to the entity manager for persistence of our account example before. You can call `EntityManager::persist($account)` and then `EntityManager::flush()` at any point to save the account to the database. All the nested `Entry` objects are automatically flushed to the database also.

```
<?php
$account = new Account("123456", 200);
$account->addEntry(500);
$account->addEntry(-200);
$em->persist($account);
$em->flush();
```

The current implementation has a considerable drawback. To get the balance, we have to initialize the complete `Account::$entries` collection, possibly a very large one. This can considerably hurt the performance of your application.

Using an Aggregate Field

To overcome the previously mentioned issue (initializing the whole entries collection) we want to add an aggregate field called "balance" on the Account and adjust the code in `Account::getBalance()` and `Account:addEntry()`:

```
<?php
class Account
{
    /**
     * @Column(type="integer")
     */
    private $balance = 0;
    public function getBalance()
    {
        return $this->balance;
    }

    public function addEntry($amount)
    {
        $this->assertAcceptEntryAllowed($amount);

        $e = new Entry($this, $amount);
        $this->entries[] = $e;
        $this->balance += $amount;
        return $e;
    }
}
```

This is a very simple change, but all the tests still pass. Our account entities return the correct balance. Now calling the `Account::getBalance()` method will not occur the overhead of loading all entries anymore. Adding a new Entry to the `Account::$entities` will also not initialize the collection internally.

Adding a new entry is therefore very performant and explicitly hooked into the domain model. It will only update the account with the current balance and insert the new entry into the database.

Tackling Race Conditions with Aggregate Fields

Whenever you denormalize your database schema race-conditions can potentially lead to inconsistent state. See this example:

```
<?php
// The Account $accId has a balance of 0 and a max credit limit of 200:
// request 1 account
$account1 = $em->find('Bank\Entities\Account', $accId);

// request 2 account
$account2 = $em->find('Bank\Entities\Account', $accId);

$account1->addEntry(-200);
$account2->addEntry(-200);

// now request 1 and 2 both flush the changes.
```

The aggregate field `Account::$balance` is now -200, however the SUM over all entries amounts yields -400. A violation of our max credit rule.

You can use both optimistic or pessimistic locking to save-guard your aggregate fields against this kind of race-conditions. Reading Eric Evans DDD carefully he mentions that the "Aggregate Root" (Account in our example) needs a locking mechanism.

Optimistic locking is as easy as adding a version column:

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

The previous example would then throw an exception in the face of whatever request saves the entity last (and would create the inconsistent state).

Pessimistic locking requires an additional flag set on the `EntityManager::find()` call, enabling write locking directly in the database using a FOR UPDATE.

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

$account = $em->find('Bank\Entities\Account', $accId, LockMode::PESSIMISTIC_READ);
```

Keeping Updates and Deletes in Sync

The example shown in this article does not allow changes to the value in `Entry`, which considerably simplifies the effort to keep `Account::$balance` in sync. If your use-case allows fields to be updated or related entities to be removed you have to encapsulate this logic in your “Aggregate Root” entity and adjust the aggregate field accordingly.

Conclusion

This article described how to obtain aggregate values using DQL or your domain model. It showed how you can easily add an aggregate field that offers serious performance benefits over iterating all the related objects that make up an aggregate value. Finally I showed how you can ensure that your aggregate fields do not get out of sync due to race-conditions and concurrent access.

9.3.2 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
{
    const MYTYPE = 'mytype'; // modify to match your type name

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        // return the SQL used to create your column type. To create a portable column type, use the
    }

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

    public function getName()
    {
        return self::MYTYPE; // modify to match your constant name
    }
}
```

The following assumptions are applied to mapping types by the ORM:

- If the value of the field is *NULL* the method `convertToDatabaseValue()` is not called.
- The `UnitOfWork` never passes values to the database convert method that did not change in the request.

When you have implemented the type you still need to let Doctrine know about it. This can be achieved through the `Doctrine\DBAL\Types\Type#addType($name, $className)` method. See the following example:

```
<?php
// in bootstrapping code

// ...

use Doctrine\DBAL\Types\Type;

// ...

// Register my type
Type::addType('mytype', 'My\Project\Types\MyType');
```

To convert the underlying database type of your new "mytype" directly into an instance of `MyType` when performing schema operations, the type has to be registered with the database platform as well:

```
<?php
$conn = $em->getConnection();
$conn->getDatabasePlatform()->registerDoctrineTypeMapping('db_mytype', 'mytype');
```

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 the new type can be used when mapping columns:

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

9.3.3 Persisting the Decorator Pattern

Section author: Chris Woodford <chris.woodford@gmail.com>

This recipe will show you a simple example of how you can use Doctrine 2 to persist an implementation of the Decorator Pattern

Component

The `Component` class needs to be persisted, so it's going to be an `Entity`. As the top of the inheritance hierarchy, it's going to have to define the persistent inheritance. For this example, we will use Single Table Inheritance, but Class Table Inheritance would work as well. In the discriminator map, we will define two concrete subclasses, `ConcreteComponent` and `ConcreteDecorator`.

```php
<?php

namespace Test;

/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"cc" = "Test\Component\ConcreteComponent",
    "cd" = "Test\Decorator\ConcreteDecorator"})
 */
abstract class Component
{

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

    /** @Column(type="string", nullable=true) */
    protected $name;

    /**
     * Get id
     * @return integer $id
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set name
     * @param string $name
     */
    public function setName($name)
    {
        $this->name = $name;
    }

    /**
     * Get name
     * @return string $name
     */
    public function getName()
    {
        return $this->name;
    }

}
```

ConcreteComponent

The `ConcreteComponent` class is pretty simple and doesn't do much more than extend the abstract `Component` class (only for the purpose of keeping this example simple).

```php
<?php

namespace Test\Component;

use Test\Component;

/** @Entity */
class ConcreteComponent extends Component
{}
```

Decorator

The `Decorator` class doesn't need to be persisted, but it does need to define an association with a persisted `Entity`. We can use a `MappedSuperclass` for this.

```php
<?php

namespace Test;

/** @MappedSuperclass */
abstract class Decorator extends Component
{

    /**
     * @OneToOne(targetEntity="Test\Component", cascade={"all"})
     * @JoinColumn(name="decorates", referencedColumnName="id")
     */
    protected $decorates;

    /**
     * initialize the decorator
     * @param Component $c
     */
    public function __construct(Component $c)
    {
        $this->setDecorates($c);
    }

    /**
     * (non-PHPdoc)
     * @see Test.Component::getName()
     */
    public function getName()
    {
        return 'Decorated ' . $this->getDecorates()->getName();
    }

    /**
     * the component being decorated
     * @return Component
     */
    protected function getDecorates()
    {
        return $this->decorates;
    }

    /**
     * sets the component being decorated
     * @param Component $c
     */
    protected function setDecorates(Component $c)
    {
        $this->decorates = $c;
    }

}
```

All operations on the `Decorator` (i.e. persist, remove, etc) will cascade from the `Decorator` to the `Component`. This means that when we persist a `Decorator`, Doctrine will take care of persisting the chain of decorated objects for us. A `Decorator` can be treated exactly as a `Component` when it comes time to persisting it.

The `Decorator's` constructor accepts an instance of a `Component`, as defined by the `Decorator` pattern. The setDecorates/getDecorates methods have been defined as protected to hide the fact that a `Decorator` is decorating a `Component` and keeps the `Component` interface and the `Decorator` interface identical.

To illustrate the intended result of the `Decorator` pattern, the getName() method has been overridden to append a string to the `Component's` getName() method.

ConcreteDecorator

The final class required to complete a simple implementation of the Decorator pattern is the `ConcreteDecorator`. In order to further illustrate how the `Decorator` can alter data as it moves through the chain of decoration, a new field, "special", has been added to this class. The getName() has been overridden and appends the value of the getSpecial() method to its return value.

```
<?php

namespace Test\Decorator;

use Test\Decorator;

/** @Entity */
class ConcreteDecorator extends Decorator
{

    /** @Column(type="string", nullable=true) */
    protected $special;

    /**
     * Set special
     * @param string $special
     */
    public function setSpecial($special)
    {
        $this->special = $special;
    }

    /**
     * Get special
     * @return string $special
     */
    public function getSpecial()
    {
        return $this->special;
    }

    /**
     * (non-PHPdoc)
     * @see Test.Component::getName()
     */
    public function getName()
    {
        return '[' . $this->getSpecial()
            . '] ' . parent::getName();
    }

}
```

Examples

Here is an example of how to persist and retrieve your decorated objects

```
<?php

use Test\Component\ConcreteComponent,
    Test\Decorator\ConcreteDecorator;

// assumes Doctrine 2 is configured and an instance of
// an EntityManager is available as $em

// create a new concrete component
$c = new ConcreteComponent();
$c->setName('Test Component 1');
$em->persist($c); // assigned unique ID = 1

// create a new concrete decorator
$c = new ConcreteComponent();
$c->setName('Test Component 2');

$d = new ConcreteDecorator($c);
$d->setSpecial('Really');
$em->persist($d);
// assigns c as unique ID = 2, and d as unique ID = 3

$em->flush();

$c = $em->find('Test\Component', 1);
$d = $em->find('Test\Component', 3);

echo get_class($c);
// prints: Test\Component\ConcreteComponent

echo $c->getName();
// prints: Test Component 1

echo get_class($d)
```

```
// prints: Test\Component\ConcreteDecorator

echo $d->getName();
// prints: [Really] Decorated Test Component 2
```

9.3.4 Extending DQL in Doctrine 2: Custom AST Walkers

Section author: Benjamin Eberlei <kontakt@beberlei.de>

The Doctrine Query Language (DQL) is a proprietary sql-dialect that substitutes tables and columns for Entity names and their fields. Using DQL you write a query against the database using your entities. With the help of the metadata you can write very concise, compact and powerful queries that are then translated into SQL by the Doctrine ORM.

In Doctrine 1 the DQL language was not implemented using a real parser. This made modifications of the DQL by the user impossible. Doctrine 2 in contrast has a real parser for the DQL language, which transforms the DQL statement into an Abstract Syntax Tree and generates the appropriate SQL statement for it. Since this process is deterministic Doctrine heavily caches the SQL that is generated from any given DQL query, which reduces the performance overhead of the parsing process to zero.

You can modify the Abstract syntax tree by hooking into DQL parsing process by adding a Custom Tree Walker. A walker is an interface that walks each node of the Abstract syntax tree, thereby generating the SQL statement.

There are two types of custom tree walkers that you can hook into the DQL parser:

- An output walker. This one actually generates the SQL, and there is only ever one of them. We implemented the default SqlWalker implementation for it.
- A tree walker. There can be many tree walkers, they cannot generate the sql, however they can modify the AST before its rendered to sql.

Now this is all awfully technical, so let me come to some use-cases fast to keep you motivated. Using walker implementation you can for example:

- Modify the AST to generate a Count Query to be used with a paginator for any given DQL query.
- Modify the Output Walker to generate vendor-specific SQL (instead of ANSI).
- Modify the AST to add additional where clauses for specific entities (example ACL, country-specific content...)
- Modify the Output walker to pretty print the SQL for debugging purposes.

In this cookbook-entry I will show examples on the first two points. There are probably much more use-cases.

Generic count query for pagination

Say you have a blog and posts all with one category and one author. A query for the front-page or any archive page might look something like:

```
SELECT p, c, a FROM BlogPost p JOIN p.category c JOIN p.author a WHERE ...
```

Now in this query the blog post is the root entity, meaning its the one that is hydrated directly from the query and returned as an array of blog posts. In contrast the comment and author are loaded for deeper use in the object tree.

A pagination for this query would want to approximate the number of posts that match the WHERE clause of this query to be able to predict the number of pages to show to the user. A draft of the DQL query for pagination would look like:

```
SELECT count(DISTINCT p.id) FROM BlogPost p JOIN p.category c JOIN p.author a WHERE ...
```

Now you could go and write each of these queries by hand, or you can use a tree walker to modify the AST for you. Lets see how the API would look for this use-case:

```php
<?php
$pageNum = 1;
$query = $em->createQuery($dql);
$query->setFirstResult( ($pageNum-1) * 20)->setMaxResults(20);

$totalResults = Paginate::count($query);
$results = $query->getResult();
```

The `Paginate::count(Query $query)` looks like:

```php
<?php
class Paginate
{
    static public function count(Query $query)
    {
        /* @var $countQuery Query */
        $countQuery = clone $query;

        $countQuery->setHint(Query::HINT_CUSTOM_TREE_WALKERS, array('DoctrineExtensions\Paginate\Cou
        $countQuery->setFirstResult(null)->setMaxResults(null);

        return $countQuery->getSingleScalarResult();
    }
}
```

It clones the query, resets the limit clause first and max results and registers the `CountSqlWalker` custom tree walker which will modify the AST to execute a count query. The walkers implementation is:

```php
<?php
class CountSqlWalker extends TreeWalkerAdapter
{
    /**
     * Walks down a SelectStatement AST node, thereby generating the appropriate SQL.
     *
     * @return string The SQL.
     */
    public function walkSelectStatement(SelectStatement $AST)
    {
        $parent = null;
        $parentName = null;
        foreach ($this->_getQueryComponents() as $dqlAlias => $qComp) {
            if ($qComp['parent'] === null && $qComp['nestingLevel'] == 0) {
                $parent = $qComp;
                $parentName = $dqlAlias;
                break;
            }
        }

        $pathExpression = new PathExpression(
            PathExpression::TYPE_STATE_FIELD | PathExpression::TYPE_SINGLE_VALUED_ASSOCIATION, $pare
            $parent['metadata']->getSingleIdentifierFieldName()
        );
        $pathExpression->type = PathExpression::TYPE_STATE_FIELD;

        $AST->selectClause->selectExpressions = array(
            new SelectExpression(
                new AggregateExpression('count', $pathExpression, true), null
            )
        );
    }
}
```

This will delete any given select expressions and replace them with a distinct count query for the root entities primary key. This will only work if your entity has only one identifier field (composite keys won't work).

Modify the Output Walker to generate Vendor specific SQL

Most RMDBS have vendor-specific features for optimizing select query execution plans. You can write your own output walker to introduce certain keywords using the Query Hint API. A query hint can be set via `Query::setHint($name, $value)` as shown in the previous example with the `HINT_CUSTOM_TREE_WALKERS` query hint.

We will implement a custom Output Walker that allows to specify the SQL_NO_CACHE query hint.

```
<?php
$dql = "SELECT p, c, a FROM BlogPost p JOIN p.category c JOIN p.author a WHERE ...";
$query = $m->createQuery($dql);
$query->setHint(Query::HINT_CUSTOM_OUTPUT_WALKER, 'DoctrineExtensions\Query\MysqlWalker');
$query->setHint("mysqlWalker.sqlNoCache", true);
$results = $query->getResult();
```

Our `MysqlWalker` will extend the default `SqlWalker`. We will modify the generation of the SELECT clause, adding the SQL_NO_CACHE on those queries that need it:

```
<?php
class MysqlWalker extends SqlWalker
{
     /**
      * Walks down a SelectClause AST node, thereby generating the appropriate SQL.
      *
      * @param $selectClause
      * @return string The SQL.
      */
    public function walkSelectClause($selectClause)
    {
        $sql = parent::walkSelectClause($selectClause);

        if ($this->getQuery()->getHint('mysqlWalker.sqlNoCache') === true) {
            if ($selectClause->isDistinct) {
                $sql = str_replace('SELECT DISTINCT', 'SELECT DISTINCT SQL_NO_CACHE', $sql);
            } else {
                $sql = str_replace('SELECT', 'SELECT SQL_NO_CACHE', $sql);
            }
        }

        return $sql;
    }
}
```

Writing extensions to the Output Walker requires a very deep understanding of the DQL Parser and Walkers, but may offer your huge benefits with using vendor specific features. This would still allow you write DQL queries instead of NativeQueries to make use of vendor specific features.

9.3.5 DQL User Defined Functions

Section author: Benjamin Eberlei <kontakt@beberlei.de>

By default DQL supports a limited subset of all the vendor-specific SQL functions common between all the vendors. However in many cases once you have decided on a specific database vendor, you will never change it during the life of your project. This decision for a specific vendor potentially allows you to make use of powerful SQL features that are unique to the vendor.

It is worth to mention that Doctrine 2 also allows you to handwrite your SQL instead of extending the DQL parser. Extending DQL is sort of an advanced extension point. You can map arbitrary SQL to your objects and gain access to vendor specific functionalities using the `EntityManager#createNativeQuery()` API as described in the *Native Query* chapter.

The DQL Parser has hooks to register functions that can then be used in your DQL queries and transformed into SQL, allowing to extend Doctrines Query capabilities to the vendors strength. This post explains the Used-Defined Functions API (UDF) of the Dql Parser and shows some examples to give you some hints how you would extend DQL.

There are three types of functions in DQL, those that return a numerical value, those that return a string and those that return a Date. Your custom method has to be registered as either one of those. The return type information is used by the DQL parser to check possible syntax errors during the parsing process, for example using a string function return value in a math expression.

Registering your own DQL functions

You can register your functions adding them to the ORM configuration:

```php
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($name, $class);
$config->addCustomNumericFunction($name, $class);
$config->addCustomDatetimeFunction($name, $class);

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

The `$name` is the name the function will be referred to in the DQL query. `$class` is a string of a class-name which has to extend `Doctrine\ORM\Query\Node\FunctionNode`. This is a class that offers all the necessary API and methods to implement a UDF.

Instead of providing the function class name, you can also provide a callable that returns the function object:

```php
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction($name, function () {
    return new MyCustomFunction();
});
```

In this post we will implement some MySql specific Date calculation methods, which are quite handy in my opinion:

Date Diff

Mysql's DateDiff function takes two dates as argument and calculates the difference in days with `date1-date2`.

The DQL parser is a top-down recursive descent parser to generate the Abstract-Syntax Tree (AST) and uses a TreeWalker approach to generate the appropriate SQL from the AST. This makes reading the Parser/TreeWalker code manageable in a finite amount of time.

The `FunctionNode` class I referred to earlier requires you to implement two methods, one for the parsing process (obviously) called `parse` and one for the TreeWalker process called `getSql()`. I show you the code for the DateDiff method and discuss it step by step:

```
<?php
/**
 * DateDiffFunction ::= "DATEDIFF" "(" ArithmeticPrimary "," ArithmeticPrimary ")"
 */
class DateDiff extends FunctionNode
{
    // (1)
    public $firstDateExpression = null;
    public $secondDateExpression = null;

    public function parse(\Doctrine\ORM\Query\Parser $parser)
    {
        $parser->match(Lexer::T_IDENTIFIER); // (2)
        $parser->match(Lexer::T_OPEN_PARENTHESIS); // (3)
        $this->firstDateExpression = $parser->ArithmeticPrimary(); // (4)
        $parser->match(Lexer::T_COMMA); // (5)
        $this->secondDateExpression = $parser->ArithmeticPrimary(); // (6)
        $parser->match(Lexer::T_CLOSE_PARENTHESIS); // (3)
    }

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'DATEDIFF(' .
            $this->firstDateExpression->dispatch($sqlWalker) . ', ' .
            $this->secondDateExpression->dispatch($sqlWalker) .
        ')'; // (7)
    }
}
```

The Parsing process of the DATEDIFF function is going to find two expressions the date1 and the date2 values, whose AST Node representations will be saved in the variables of the DateDiff FunctionNode instance at (1).

The parse() method has to cut the function call "DATEDIFF" and its argument into pieces. Since the parser detects the function using a lookahead the T_IDENTIFIER of the function name has to be taken from the stack (2), followed by a detection of the arguments in (4)-(6). The opening and closing parenthesis have to be detected also. This happens during the Parsing process and leads to the generation of a DateDiff FunctionNode somewhere in the AST of the dql statement.

The `ArithmeticPrimary` method call is the most common denominator of valid EBNF tokens taken from the DQL EBNF grammar that matches our requirements for valid input into the DateDiff Dql function. Picking the right tokens for your methods is a tricky business, but the EBNF grammar is pretty helpful finding it, as is looking at the Parser source code.

Now in the TreeWalker process we have to pick up this node and generate SQL from it, which apparently is quite easy looking at the code in (7). Since we don't know which type of AST Node the first and second Date expression are we are just dispatching them back to the SQL Walker to generate SQL from and then wrap our DATEDIFF function call around this output.

Now registering this DateDiff FunctionNode with the ORM using:

```
<?php
$config = new \Doctrine\ORM\Configuration();
$config->addCustomStringFunction('DATEDIFF', 'DoctrineExtensions\Query\MySql\DateDiff');
```

We can do fancy stuff like:

```
SELECT p FROM DoctrineExtensions\Query\BlogPost p WHERE DATEDIFF(CURRENT_TIME(), p.created) < 7
```

Date Add

Often useful it the ability to do some simple date calculations in your DQL query using MySql's DATE_ADD function.

I'll skip the blah and show the code for this function:

```php
<?php
/**
 * DateAddFunction ::=
 *     "DATE_ADD" "(" ArithmeticPrimary ", INTERVAL" ArithmeticPrimary Identifier ")"
 */
class DateAdd extends FunctionNode
{
    public $firstDateExpression = null;
    public $intervalExpression = null;
    public $unit = null;

    public function parse(\Doctrine\ORM\Query\Parser $parser)
    {
        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);

        $this->firstDateExpression = $parser->ArithmeticPrimary();

        $parser->match(Lexer::T_COMMA);
        $parser->match(Lexer::T_IDENTIFIER);

        $this->intervalExpression = $parser->ArithmeticPrimary();

        $parser->match(Lexer::T_IDENTIFIER);

        /* @var $lexer Lexer */
        $lexer = $parser->getLexer();
        $this->unit = $lexer->token['value'];

        $parser->match(Lexer::T_CLOSE_PARENTHESIS);
    }

    public function getSql(\Doctrine\ORM\Query\SqlWalker $sqlWalker)
    {
        return 'DATE_ADD(' .
            $this->firstDateExpression->dispatch($sqlWalker) . ', INTERVAL ' .
            $this->intervalExpression->dispatch($sqlWalker) . ' ' . $this->unit .
        ')';
    }
}
```

The only difference compared to the DATEDIFF here is, we additionally need the `Lexer` to access the value of the `T_IDENTIFIER` token for the Date Interval unit, for example the MONTH in:

```
SELECT p FROM DoctrineExtensions\Query\BlogPost p WHERE DATE_ADD(CURRENT_TIME(), INTERVAL 4 MONTH) >
```

The above method now only supports the specification using `INTERVAL`, to also allow a real date in DATE_ADD we need to add some decision logic to the parsing process (makes up for a nice exercise).

Now as you see, the Parsing process doesn't catch all the possible SQL errors, here we don't match for all the valid inputs for the interval unit. However where necessary we rely on the database vendors SQL parser to show us further errors in the parsing process, for example if the Unit would not be one of the supported values by MySql.

Conclusion

Now that you all know how you can implement vendor specific SQL functionalities in DQL, we would be excited to see user extensions that add vendor specific function packages, for example more math functions, XML + GIS Support, Hashing functions and so on.

For 2.0 we will come with the current set of functions, however for a future version we will re-evaluate if we can abstract even more vendor sql functions and extend the DQL languages scope.

Code for this Extension to DQL and other Doctrine Extensions can be found in my Github DoctrineExtensions repository.

9.3.6 Implementing ArrayAccess for Domain Objects

Section author: Roman Borschel (roman@code-factory.org)

This recipe will show you how to implement ArrayAccess for your domain objects in order to allow more uniform access, for example in templates. In these examples we will implement ArrayAccess on a Layer Supertype for all our domain objects.

Option 1

In this implementation we will make use of PHPs highly dynamic nature to dynamically access properties of a subtype in a supertype at runtime. Note that this implementation has 2 main caveats:

- It will not work with private fields
- It will not go through any getters/setters

```php
<?php
abstract class DomainObject implements ArrayAccess
{
    public function offsetExists($offset) {
        return isset($this->$offset);
    }

    public function offsetSet($offset, $value) {
        $this->$offset = $value;
    }

    public function offsetGet($offset) {
        return $this->$offset;
    }

    public function offsetUnset($offset) {
        $this->$offset = null;
    }
}
```

Option 2

In this implementation we will dynamically invoke getters/setters. Again we use PHPs dynamic nature to invoke methods on a subtype from a supertype at runtime. This implementation has the following caveats:

- It relies on a naming convention
- The semantics of offsetExists can differ
- offsetUnset will not work with typehinted setters

```php
<?php
abstract class DomainObject implements ArrayAccess
{
    public function offsetExists($offset) {
        // In this example we say that exists means it is not null
        $value = $this->{"get$offset"}();
        return $value !== null;
    }

    public function offsetSet($offset, $value) {
        $this->{"set$offset"}($value);
    }

    public function offsetGet($offset) {
        return $this->{"get$offset"}();
    }

    public function offsetUnset($offset) {
        $this->{"set$offset"}(null);
    }
}
```

Read-only

You can slightly tweak option 1 or option 2 in order to make array access read-only. This will also circumvent some of the caveats of each option. Simply make offsetSet and offsetUnset throw an exception (i.e. BadMethodCallException).

```php
<?php
abstract class DomainObject implements ArrayAccess
{
    public function offsetExists($offset) {
        // option 1 or option 2
    }

    public function offsetSet($offset, $value) {
        throw new BadMethodCallException("Array access of class " . get_class($this) . " is read-only
    }

    public function offsetGet($offset) {
        // option 1 or option 2
    }

    public function offsetUnset($offset) {
        throw new BadMethodCallException("Array access of class " . get_class($this) . " is read-only
    }
}
```

9.3.7 Implementing the Notify ChangeTracking Policy

Section author: Roman Borschel (roman@code-factory.org)

The NOTIFY change-tracking policy is the most effective change-tracking policy provided by Doctrine but it requires some boilerplate code. This recipe will show you how this boilerplate code should look like. We will implement it on a Layer Supertype for all our domain objects.

Implementing NotifyPropertyChanged

The NOTIFY 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.

```php
<?php
use Doctrine\Common\NotifyPropertyChanged;
use Doctrine\Common\PropertyChangedListener;

abstract class DomainObject implements NotifyPropertyChanged
{
    private $listeners = array();

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

    /** Notifies listeners of a change. */
    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 concrete, derived domain classes, you need to invoke onPropertyChanged as follows to notify listeners:

```php
<?php
// Mapping not shown, either in annotations, xml or yaml as usual
class MyEntity extends DomainObject
{
    private $data;
    // ... other fields as usual

    public function setData($data) {
        if ($data != $this->data) { // check: is it actually modified?
            $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 can avoid unnecessary updates and also have full control over when you consider a property changed.

9.3.8 Implementing Wakeup or Clone

Section author: Roman Borschel (roman@code-factory.org)

As explained in the restrictions for entity classes in the manual, it is usually not allowed for an entity to implement `__wakeup` or `__clone`, because Doctrine makes special use of them. However, it is quite easy to make use of these methods in a safe way by guarding the custom wakeup or clone code with an entity identity check, as demonstrated in the following sections.

Safely implementing __wakeup

To safely implement `__wakeup`, simply enclose your implementation code in an identity check as follows:

```php
<?php
class MyEntity
{
    private $id; // This is the identifier of the entity.
    //...

    public function __wakeup()
    {
        // If the entity has an identity, proceed as normal.
        if ($this->id) {
            // ... Your code here as normal ...
        }
        // otherwise do nothing, do NOT throw an exception!
    }

    //...
}
```

Safely implementing __clone

Safely implementing `__clone` is pretty much the same:

```php
<?php
class MyEntity
{
    private $id; // This is the identifier of the entity.
    //...

    public function __clone()
    {
        // If the entity has an identity, proceed as normal.
        if ($this->id) {
            // ... Your code here as normal ...
        }
        // otherwise do nothing, do NOT throw an exception!
    }

    //...
}
```

Summary

As you have seen, it is quite easy to safely make use of `__wakeup` and `__clone` in your entities without adding any really Doctrine-specific or Doctrine-dependant code.

These implementations are possible and safe because when Doctrine invokes these methods, the entities never have an identity (yet). Furthermore, it is possibly a good idea to check for the identity in your code anyway, since it's rarely the case that you want to unserialize or clone an entity with no identity.

9.3.9 Integrating with CodeIgniter

This is recipe for using Doctrine 2 in your CodeIgniter framework.

Note: This might not work for all CodeIgniter versions and may require slight adjustments.

Here is how to set it up:

Make a CodeIgniter library that is both a wrapper and a bootstrap for Doctrine 2.

Setting up the file structure

Here are the steps:

- Add a php file to your system/application/libraries folder called Doctrine.php. This is going to be your wrapper/bootstrap for the D2 entity manager.
- Put the Doctrine folder (the one that contains Common, DBAL, and ORM) inside that same libraries folder.
- Your system/application/libraries folder now looks like this:

  system/applications/libraries -Doctrine -Doctrine.php -index.html
- If you want, open your config/autoload.php file and autoload your Doctrine library.

  <?php $autoload['libraries'] = array('doctrine');

Creating your Doctrine CodeIgniter library

Now, here is what your Doctrine.php file should look like. Customize it to your needs.

```php
<?php
use Doctrine\Common\ClassLoader,
    Doctrine\ORM\Configuration,
    Doctrine\ORM\EntityManager,
    Doctrine\Common\Cache\ArrayCache,
    Doctrine\DBAL\Logging\EchoSQLLogger;

class Doctrine {

  public $em = null;

  public function __construct()
  {
    // load database configuration from CodeIgniter
    require_once APPPATH.'config/database.php';

    // Set up class loading. You could use different autoloaders, provided by your favorite framewor
    // if you want to.
    require_once APPPATH.'libraries/Doctrine/Common/ClassLoader.php';

    $doctrineClassLoader = new ClassLoader('Doctrine',  APPPATH.'libraries');
    $doctrineClassLoader->register();
    $entitiesClassLoader = new ClassLoader('models', rtrim(APPPATH, "/" ));
    $entitiesClassLoader->register();
    $proxiesClassLoader = new ClassLoader('Proxies', APPPATH.'models/proxies');
    $proxiesClassLoader->register();

    // Set up caches
    $config = new Configuration;
    $cache = new ArrayCache;
    $config->setMetadataCacheImpl($cache);
    $driverImpl = $config->newDefaultAnnotationDriver(array(APPPATH.'models/Entities'));
    $config->setMetadataDriverImpl($driverImpl);
    $config->setQueryCacheImpl($cache);

    $config->setQueryCacheImpl($cache);

    // Proxy configuration
    $config->setProxyDir(APPPATH.'/models/proxies');
    $config->setProxyNamespace('Proxies');

    // Set up logger
    $logger = new EchoSQLLogger;
    $config->setSQLLogger($logger);

    $config->setAutoGenerateProxyClasses( TRUE );

    // Database connection information
    $connectionOptions = array(
        'driver' => 'pdo_mysql',
        'user' =>     $db['default']['username'],
        'password' => $db['default']['password'],
        'host' =>     $db['default']['hostname'],
        'dbname' =>   $db['default']['database']
    );

    // Create EntityManager
    $this->em = EntityManager::create($connectionOptions, $config);
  }
}
```

Please note that this is a development configuration; for a production system you'll want to use a real caching system like APC, get rid of EchoSqlLogger, and turn off autoGenerateProxyClasses.

For more details, consult the Doctrine 2 Configuration documentation.

Now to use it

Whenever you need a reference to the entity manager inside one of your controllers, views, or models you can do this:

```
<?php
$em = $this->doctrine->em;
```

That's all there is to it. Once you get the reference to your EntityManager do your Doctrine 2.0 voodoo as normal.

Note: If you do not choose to autoload the Doctrine library, you will need to put this line before you get a reference to it:

```
<?php
$this->load->library('doctrine');
```

Good luck!

9.3.10 Keeping your Modules independent

New in version 2.2.

One of the goals of using modules is to create discrete units of functionality that do not have many (if any) dependencies, allowing you to use that functionality in other applications without including unnecessary items.

Doctrine 2.2 includes a new utility called the `ResolveTargetEntityListener`, that functions by intercepting certain calls inside Doctrine and rewrite targetEntity parameters in your metadata mapping at runtime. It means that in your bundle you are able to use an interface or abstract class in your mappings and expect correct mapping to a concrete entity at runtime.

This functionality allows you to define relationships between different entities but not making them hard dependencies.

Background

In the following example, the situation is we have an *InvoiceModule* which provides invoicing functionality, and a *CustomerModule* that contains customer management tools. We want to keep these separated, because they can be used in other systems without each other, but for our application we want to use them together.

In this case, we have an `Invoice` entity with a relationship to a non-existent object, an `InvoiceSubjectInterface`. The goal is to get the `ResolveTargetEntityListener` to replace any mention of the interface with a real object that implements that interface.

Set up

We're going to use the following basic entities (which are incomplete for brevity) to explain how to set up and use the RTEL.

A Customer entity

```
<?php
// src/Acme/AppModule/Entity/Customer.php

namespace Acme\AppModule\Entity;

use Doctrine\ORM\Mapping as ORM;
use Acme\CustomerModule\Entity\Customer as BaseCustomer;
use Acme\InvoiceModule\Model\InvoiceSubjectInterface;

/**
 * @ORM\Entity
 * @ORM\Table(name="customer")
 */
class Customer extends BaseCustomer implements InvoiceSubjectInterface
{
    // In our example, any methods defined in the InvoiceSubjectInterface
    // are already implemented in the BaseCustomer
}
```

An Invoice entity

```php
<?php
// src/Acme/InvoiceModule/Entity/Invoice.php

namespace Acme\InvoiceModule\Entity;

use Doctrine\ORM\Mapping AS ORM;
use Acme\InvoiceModule\Model\InvoiceSubjectInterface;

/**
 * Represents an Invoice.
 *
 * @ORM\Entity
 * @ORM\Table(name="invoice")
 */
class Invoice
{
    /**
     * @ORM\ManyToOne(targetEntity="Acme\InvoiceModule\Model\InvoiceSubjectInterface")
     * @var InvoiceSubjectInterface
     */
    protected $subject;
}
```

An InvoiceSubjectInterface

```php
<?php
// src/Acme/InvoiceModule/Model/InvoiceSubjectInterface.php

namespace Acme\InvoiceModule\Model;

/**
 * An interface that the invoice Subject object should implement.
 * In most circumstances, only a single object should implement
 * this interface as the ResolveTargetEntityListener can only
 * change the target to a single object.
 */
interface InvoiceSubjectInterface
{
    // List any additional methods that your InvoiceModule
    // will need to access on the subject so that you can
    // be sure that you have access to those methods.

    /**
     * @return string
     */
    public function getName();
}
```

Next, we need to configure the listener. Add this to the area you set up Doctrine. You must set this up in the way outlined below, otherwise you can not be guaranteed that the targetEntity resolution will occur reliably:

```php
<?php
$evm  = new \Doctrine\Common\EventManager;
$rtel = new \Doctrine\ORM\Tools\ResolveTargetEntityListener;

// Adds a target-entity class
$rtel->addResolveTargetEntity('Acme\\InvoiceModule\\Model\\InvoiceSubjectInterface', 'Acme\\Customer
// Add the ResolveTargetEntityListener
$evm->addEventListener(Doctrine\ORM\Events::loadClassMetadata, $rtel);

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

Final Thoughts

With the `ResolveTargetEntityListener`, we are able to decouple our bundles, keeping them usable by themselves, but still being able to define relationships between different objects. By using this method, I've found my bundles end up being easier to maintain independently.

9.3.11 SQL-Table Prefixes

This recipe is intended as an example of implementing a loadClassMetadata listener to provide a Table Prefix option for your application. The method used below is not a hack, but fully integrates into the Doctrine system, all SQL generated will include the appropriate table prefix.

In most circumstances it is desirable to separate different applications into individual databases, but in certain cases, it may be beneficial to have a table prefix for your Entities to separate them from other vendor products in the same database.

Implementing the listener

The listener in this example has been set up with the DoctrineExtensions namespace. You create this file in your library/DoctrineExtensions directory, but will need to set up appropriate autoloaders.

```
<?php

namespace DoctrineExtensions;
use \Doctrine\ORM\Event\LoadClassMetadataEventArgs;

class TablePrefix
{
    protected $prefix = '';

    public function __construct($prefix)
    {
        $this->prefix = (string) $prefix;
    }

    public function loadClassMetadata(LoadClassMetadataEventArgs $eventArgs)
    {
        $classMetadata = $eventArgs->getClassMetadata();
        $classMetadata->setTableName($this->prefix . $classMetadata->getTableName());
        foreach ($classMetadata->getAssociationMappings() as $fieldName => $mapping) {
            if ($mapping['type'] == \Doctrine\ORM\Mapping\ClassMetadataInfo::MANY_TO_MANY) {
                $mappedTableName = $classMetadata->associationMappings[$fieldName]['joinTable']['name
                $classMetadata->associationMappings[$fieldName]['joinTable']['name'] = $this->prefix
            }
        }
    }

}
```

Telling the EntityManager about our listener

A listener of this type must be set up before the EntityManager has been initialised, otherwise an Entity might be created or cached before the prefix has been set.

Note: If you set this listener up, be aware that you will need to clear your caches and drop then recreate your database schema.

```php
<?php

// $connectionOptions and $config set earlier

$evm = new \Doctrine\Common\EventManager;

// Table Prefix
$tablePrefix = new \DoctrineExtensions\TablePrefix('prefix_');
$evm->addEventListener(\Doctrine\ORM\Events::loadClassMetadata, $tablePrefix);

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

9.3.12 Strategy-Pattern

This recipe will give you a short introduction on how to design similar entities without using expensive (i.e. slow) inheritance but with not more than * the well-known strategy pattern * event listeners

Scenario / Problem

Given a Content-Management-System, we probably want to add / edit some so-called "blocks" and "panels". What are they for?

- A block might be a registration form, some text content, a table with information. A good example might also be a small calendar.
- A panel is by definition a block that can itself contain blocks. A good example for a panel might be a sidebar box: You could easily add a small calendar into it.

So, in this scenario, when building your CMS, you will surely add lots of blocks and panels to your pages and you will find yourself highly uncomfortable because of the following:

- Every existing page needs to know about the panels it contains - therefore, you'll have an association to your panels. But if you've got several types of panels - what do you do? Add an association to every panel-type? This wouldn't be flexible. You might be tempted to add an AbstractPanelEntity and an AbstractBlockEntity that use class inheritance. Your page could then only confer to the AbstractPanelType and Doctrine 2 would do the rest for you, i.e. load the right entities. But - you'll for sure have lots of panels and blocks, and even worse, you'd have to edit the discriminator map *manually* every time you or another developer implements a new block / entity. This would tear down any effort of modular programming.

Therefore, we need something that's far more flexible.

Solution

The solution itself is pretty easy. We will have one base class that will be loaded via the page and that has specific behaviour - a Block class might render the front-end and even the backend, for example. Now, every block that you'll write might look different or need different data - therefore, we'll offer an API to these methods but internally, we use a strategy that exactly knows what to do.

First of all, we need to make sure that we have an interface that contains every needed action. Such actions would be rendering the front-end or the backend, solving dependencies (blocks that are supposed to be placed in the sidebar could refuse to be placed in the middle of your page, for example).

Such an interface could look like this:

```php
<?php
/**
 * This interface defines the basic actions that a block / panel needs to support.
 *
 * Every blockstrategy is *only* responsible for rendering a block and declaring some basic
 * support, but *not* for updating its configuration etc. For this purpose, use controllers
 * and models.
 */
interface BlockStrategyInterface {
    /**
     * This could configure your entity
     */
    public function setConfig(Config\EntityConfig $config);

    /**
     * Returns the config this strategy is configured with.
     * @return Core\Model\Config\EntityConfig
     */
    public function getConfig();

    /**
     * Set the view object.
     * @param  \Zend_View_Interface $view
     * @return \Zend_View_Helper_Interface
     */
    public function setView(\Zend_View_Interface $view);

    /**
     * @return \Zend_View_Interface
     */
    public function getView();

    /**
     * Renders this strategy. This method will be called when the user
     * displays the site.
     *
     * @return string
     */
    public function renderFrontend();

    /**
     * Renders the backend of this block. This method will be called when
     * a user tries to reconfigure this block instance.
     *
     * Most of the time, this method will return / output a simple form which in turn
     * calls some controllers.
     *
     * @return string
     */
    public function renderBackend();

    /**
     * Returns all possible types of panels this block can be stacked onto
     *
     * @return array
     */
    public function getRequiredPanelTypes();

    /**
     * Determines whether a Block is able to use a given type or not
     * @param string $typeName The typename
     * @return boolean
     */
    public function canUsePanelType($typeName);

    public function setBlockEntity(AbstractBlock $block);

    public function getBlockEntity();
}
```

As you can see, we have a method "setBlockEntity" which ties a potential strategy to an object of type AbstractBlock. This type will simply define the basic behaviour of our blocks and could potentially look something like this:

```
<?php
/**
 * This is the base class for both Panels and Blocks.
 * It shouldn't be extended by your own blocks - simply write a strategy!
 */
abstract class AbstractBlock {
    /**
     * The id of the block item instance
     * This is a doctrine field, so you need to setup generation for it
     * @var integer
     */
    private $id;

    // Add code for relation to the parent panel, configuration objects, ....

    /**
     * This var contains the classname of the strategy
     * that is used for this blockitem. (This string (!) value will be persisted by Doctrine 2)
     *
     * This is a doctrine field, so make sure that you use an @column annotation or setup your
     * yaml or xml files correctly
     * @var string
     */
    protected $strategyClassName;

    /**
     * This var contains an instance of $this->blockStrategy. Will not be persisted by Doctrine 2.
     *
     * @var BlockStrategyInterface
     */
    protected $strategyInstance;

    /**
     * Returns the strategy that is used for this blockitem.
     *
     * The strategy itself defines how this block can be rendered etc.
     *
     * @return string
     */
    public function getStrategyClassName() {
        return $this->strategyClassName;
    }

    /**
     * Returns the instantiated strategy
     *
     * @return BlockStrategyInterface
     */
    public function getStrategyInstance() {
        return $this->strategyInstance;
    }

    /**
     * Sets the strategy this block / panel should work as. Make sure that you've used
     * this method before persisting the block!
     *
     * @param BlockStrategyInterface $strategy
     */
    public function setStrategy(BlockStrategyInterface $strategy) {
        $this->strategyInstance  = $strategy;
        $this->strategyClassName = get_class($strategy);
        $strategy->setBlockEntity($this);
    }
```

Now, the important point is that $strategyClassName is a Doctrine 2 field, i.e. Doctrine will persist this value. This is only the class name of your strategy and not an instance!

Finishing your strategy pattern, we hook into the Doctrine postLoad event and check whether a block has been loaded. If so, you will initialize it - i.e. get the strategies classname, create an instance of it and set it via setStrategyBlock().

This might look like this:

```
<?php
use \Doctrine\ORM,
    \Doctrine\Common;

/**
 * The BlockStrategyEventListener will initialize a strategy after the
 * block itself was loaded.
 */
class BlockStrategyEventListener implements Common\EventSubscriber {

    protected $view;

    public function __construct(\Zend_View_Interface $view) {
        $this->view = $view;
    }

    public function getSubscribedEvents() {
       return array(ORM\Events::postLoad);
    }

    public function postLoad(ORM\Event\LifecycleEventArgs $args) {
        $blockItem = $args->getEntity();

        // Both blocks and panels are instances of Block\AbstractBlock
        if ($blockItem instanceof Block\AbstractBlock) {
            $strategy  = $blockItem->getStrategyClassName();
```

```
                $strategyInstance = new $strategy();
                if (null !== $blockItem->getConfig()) {
                    $strategyInstance->setConfig($blockItem->getConfig());
                }
                $strategyInstance->setView($this->view);
                $blockItem->setStrategy($strategyInstance);
            }
        }
}
```

In this example, even some variables are set - like a view object or a specific configuration object.

9.3.13 Validation of Entities

Section author: Benjamin Eberlei <kontakt@beberlei.de>

Doctrine 2 does not ship with any internal validators, the reason being that we think all the frameworks out there already ship with quite decent ones that can be integrated into your Domain easily. What we offer are hooks to execute any kind of validation.

Note: You don't need to validate your entities in the lifecycle events. Its only one of many options. Of course you can also perform validations in value setters or any other method of your entities that are used in your code.

Entities can register lifecycle event methods with Doctrine that are called on different occasions. For validation we would need to hook into the events called before persisting and updating. Even though we don't support validation out of the box, the implementation is even simpler than in Doctrine 1 and you will get the additional benefit of being able to re-use your validation in any other part of your domain.

Say we have an `Order` with several `OrderLine` instances. We never want to allow any customer to order for a larger sum than he is allowed to:

```
<?php
class Order
{
    public function assertCustomerAllowedBuying()
    {
        $orderLimit = $this->customer->getOrderLimit();

        $amount = 0;
        foreach ($this->orderLines as $line) {
            $amount += $line->getAmount();
        }

        if ($amount > $orderLimit) {
            throw new CustomerOrderLimitExceededException();
        }
    }
}
```

Now this is some pretty important piece of business logic in your code, enforcing it at any time is important so that customers with a unknown reputation don't owe your business too much money.

We can enforce this constraint in any of the metadata drivers. First Annotations:

```
<?php
/**
 * @Entity
 * @HasLifecycleCallbacks
 */
class Order
{
    /**
     * @PrePersist @PreUpdate
     */
    public function assertCustomerAllowedBuying() {}
}
```

In XML Mappings:

```
<doctrine-mapping>
    <entity name="Order">
        <lifecycle-callbacks>
            <lifecycle-callback type="prePersist" method="assertCustomerallowedBuying" />
            <lifecycle-callback type="preUpdate" method="assertCustomerallowedBuying" />
        </lifecycle-callbacks>
    </entity>
</doctrine-mapping>
```

YAML needs some little change yet, to allow multiple lifecycle events for one method, this will happen before Beta 1 though.

Now validation is performed whenever you call `EntityManager#persist($order)` or when you call `EntityManager#flush()` and an order is about to be updated. Any Exception that happens in the lifecycle callbacks will be cached by the EntityManager and the current transaction is rolled back.

Of course you can do any type of primitive checks, not null, email-validation, string size, integer and date ranges in your validation callbacks.

```
<?php
class Order
{
    /**
     * @PrePersist @PreUpdate
     */
    public function validate()
    {
        if (!($this->plannedShipDate instanceof DateTime)) {
            throw new ValidateException();
        }

        if ($this->plannedShipDate->format('U') < time()) {
            throw new ValidateException();
        }

        if ($this->customer == null) {
            throw new OrderRequiresCustomerException();
        }
    }
}
```

What is nice about lifecycle events is, you can also re-use the methods at other places in your domain, for example in combination with your form library. Additionally there is no limitation in the number of methods you register on one particular event, i.e. you can register multiple methods for validation in “PrePersist” or “PreUpdate” or mix and share them in any combinations between those two events.

There is no limit to what you can and can’t validate in “PrePersist” and “PreUpdate” as long as you don’t create new entity instances. This was already discussed in the previous blog post on the Versionable extension, which requires another type of event called "onFlush".

Further readings: *Lifecycle Events*

9.3.14 Working with DateTime Instances

There are many nitty gritty details when working with PHPs DateTime instances. You have know their inner workings pretty well not to make mistakes with date handling. This cookbook entry holds several interesting pieces of information on how to work with PHP DateTime instances in Doctrine 2.

DateTime changes are detected by Reference

When calling `EntityManager#flush()` Doctrine computes the changesets of all the currently managed entities and saves the differences to the database. In case of object properties (@Column(type="datetime") or @Column(type="object")) these comparisons are always made **BY REFERENCE**. That means the following change will **NOT** be saved into the database:

```php
<?php
/** @Entity */
class Article
{
    /** @Column(type="datetime") */
    private $updated;

    public function setUpdated()
    {
        // will NOT be saved in the database
        $this->updated->modify("now");
    }
}
```

The way to go would be:

```php
<?php
class Article
{
    public function setUpdated()
    {
        // WILL be saved in the database
        $this->updated = new \DateTime("now");
    }
}
```

Default Timezone Gotcha

By default Doctrine assumes that you are working with a default timezone. Each DateTime instance that is created by Doctrine will be assigned the timezone that is currently the default, either through the `date.timezone` ini setting or by calling `date_default_timezone_set()`.

This is very important to handle correctly if your application runs on different serves or is moved from one to another server (with different timezone settings). You have to make sure that the timezone is the correct one on all this systems.

Handling different Timezones with the DateTime Type

If you first come across the requirement to save different you are still optimistic to manage this mess, however let me crush your expectations fast. There is not a single database out there (supported by Doctrine 2) that supports timezones correctly. Correctly here means that you can cover all the use-cases that can come up with timezones. If you don't believe me you should read up on Storing DateTime in Databases.

The problem is simple. Not a single database vendor saves the timezone, only the differences to UTC. However with frequent daylight saving and political timezone changes you can have a UTC offset that moves in different offset directions depending on the real location.

The solution for this dilemma is simple. Don't use timezones with DateTime and Doctrine 2. However there is a workaround that even allows correct date-time handling with timezones:

1. Always convert any DateTime instance to UTC.
2. Only set Timezones for displaying purposes
3. Save the Timezone in the Entity for persistence.

Say we have an application for an international postal company and employees insert events regarding postal-package around the world, in their current timezones. To determine the exact time an event occurred means to save both the UTC time at the time of the booking and the timezone the event happened in.

```php
<?php

namespace DoctrineExtensions\DBAL\Types;

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

class UTCDateTimeType extends DateTimeType
{
    static private $utc = null;

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if ($value === null) {
            return null;
        }


        return $value->format($platform->getDateTimeFormatString(),
            (self::$utc) ? self::$utc : (self::$utc = new \DateTimeZone('UTC'))
        );
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        if ($value === null) {
            return null;
        }

        $val = \DateTime::createFromFormat(
            $platform->getDateTimeFormatString(),
            $value,
            (self::$utc) ? self::$utc : (self::$utc = new \DateTimeZone('UTC'))
        );
        if (!$val) {
            throw ConversionException::conversionFailed($value, $this->getName());
        }
        return $val;
    }
}
```

This database type makes sure that every DateTime instance is always saved in UTC, relative to the current timezone that the passed DateTime instance has. To be able to transform these values back into their real timezone you have to save the timezone in a separate field of the entity requiring timezoned datetimes:

```
<?php
namespace Shipping;

/**
 * @Entity
 */
class Event
{
    /** @Column(type="datetime") */
    private $created;

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

    /**
     * @var bool
     */
    private $localized = false;

    public function __construct(\DateTime $createDate)
    {
        $this->localized = true;
        $this->created = $createDate;
        $this->timezone = $createDate->getTimeZone()->getName();
    }

    public function getCreated()
    {
        if (!$this->localized) {
            $this->created->setTimeZone(new \DateTimeZone($this->timezone));
        }
        return $this->created;
    }
}
```

This snippet makes use of the previously discussed "changeset by reference only" property of objects. That means a new DateTime will only be used during updating if the reference changes between retrieval and flush operation. This means we can easily go and modify the instance by setting the previous local timezone.

9.3.15 Mysql Enums

The type system of Doctrine 2 consists of flyweights, which means there is only one instance of any given type. Additionally types do not contain state. Both assumptions make it rather complicated to work with the Enum Type of MySQL that is used quite a lot by developers.

When using Enums with a non-tweaked Doctrine 2 application you will get errors from the Schema-Tool commands due to the unknown database type "enum". By default Doctrine does not map the MySQL enum type to a Doctrine type. This is because Enums contain state (their allowed values) and Doctrine types don't.

This cookbook entry shows two possible solutions to work with MySQL enums. But first a word of warning. The MySQL Enum type has considerable downsides:

- Adding new values requires to rebuild the whole table, which can take hours depending on the size.
- Enums are ordered in the way the values are specified, not in their "natural" order.
- Enums validation mechanism for allowed values is not necessarily good, specifying invalid values leads to an empty enum for the default MySQL error settings. You can easily replicate the "allow only some values" requirement in your Doctrine entities.

Solution 1: Mapping to Varchars

You can map ENUMs to varchars. You can register MySQL ENUMs to map to Doctrine varchars. This way Doctrine always resolves ENUMs to Doctrine varchars. It will even detect this match correctly when using SchemaTool update commands.

```
<?php
$conn = $em->getConnection();
$conn->getDatabasePlatform()->registerDoctrineTypeMapping('enum', 'string');
```

In this case you have to ensure that each varchar field that is an enum in the database only gets passed the allowed values. You can easily enforce this in your entities:

```
<?php
/** @Entity */
class Article
{
    const STATUS_VISIBLE = 'visible';
    const STATUS_INVISIBLE = 'invisible';

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

    public function setStatus($status)
    {
        if (!in_array($status, array(self::STATUS_VISIBLE, self::STATUS_INVISIBLE))) {
            throw new \InvalidArgumentException("Invalid status");
        }
        $this->status = $status;
    }
}
```

If you want to actively create enums through the Doctrine Schema-Tool by using the **columnDefinition** attribute.

```
<?php
/** @Entity */
class Article
{
    /** @Column(type="string", columnDefinition="ENUM('visible', 'invisible')") */
    private $status;
}
```

In this case however Schema-Tool update will have a hard time not to request changes for this column on each call.

Solution 2: Defining a Type

You can make a stateless ENUM type by creating a type class for each unique set of ENUM values. For example for the previous enum type:

```php
<?php
namespace MyProject\DBAL;

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

class EnumVisibilityType extends Type
{
    const ENUM_VISIBILITY = 'enumvisibility';
    const STATUS_VISIBLE = 'visible';
    const STATUS_INVISIBLE = 'invisible';

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        return "ENUM('visible', 'invisible') COMMENT '(DC2Type:enumvisibility)'";
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return $value;
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if (!in_array($value, array(self::STATUS_VISIBLE, self::STATUS_INVISIBLE))) {
            throw new \InvalidArgumentException("Invalid status");
        }
        return $value;
    }

    public function getName()
    {
        return self::ENUM_VISIBILITY;
    }
}
```

You can register this type with `Type::addType('enumvisibility', 'MyProject\DBAL\EnumVisibilityType');`. Then in your entity you can just use this type:

```php
<?php
/** @Entity */
class Article
{
    /** @Column(type="enumvisibility") */
    private $status;
}
```

You can generalize this approach easily to create a base class for enums:

```php
<?php
namespace MyProject\DBAL;

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

```
abstract class EnumType extends Type
{
    protected $name;
    protected $values = array();

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        $values = array_map(function($val) { return "'".$val."'"; }, $this->values);

        return "ENUM(".implode(", ", $values).") COMMENT '(DC2Type:".$this->name.")'";
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        return $value;
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if (!in_array($value, $this->values)) {
            throw new \InvalidArgumentException("Invalid '".$this->name."' value.");
        }
        return $value;
    }

    public function getName()
    {
        return $this->name;
    }
}
```

With this base class you can define an enum as easily as:

```
<?php
namespace MyProject\DBAL;

class EnumVisibilityType extends EnumType
{
    protected $name = 'enumvisibility';
    protected $values = array('visible', 'invisible');
}
```

9.3.16 Advanced field value conversion using custom mapping types

Section author: Jan Sorgalla <jsorgalla@googlemail.com>

When creating entities, you sometimes have the need to transform field values before they are saved to the database. In Doctrine you can use Custom Mapping Types to solve this (see: *reference-basic-mapping-custom-mapping-types*).

There are several ways to achieve this: converting the value inside the Type class, converting the value on the database-level or a combination of both.

This article describes the third way by implementing the MySQL specific column type Point.

The `Point` type is part of the Spatial extension of MySQL and enables you to store a single location in a coordinate space by using x and y coordinates. You can use the Point type to store a longitude/latitude pair to represent a geographic location.

The entity

We create a simple entity with a field `$point` which holds a value object `Point` representing the latitude and longitude of the position.

The entity class:

```php
<?php

namespace Geo\Entity;

/**
 * @Entity
 */
class Location
{
    /**
     * @Column(type="point")
     *
     * @var \Geo\ValueObject\Point
     */
    private $point;

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

    /**
     * @param \Geo\ValueObject\Point $point
     */
    public function setPoint(\Geo\ValueObject\Point $point)
    {
        $this->point = $point;
    }

    /**
     * @return \Geo\ValueObject\Point
     */
    public function getPoint()
    {
        return $this->point;
    }

    /**
     * @param string $address
     */
    public function setAddress($address)
    {
        $this->address = $address;
    }

    /**
     * @return string
     */
    public function getAddress()
    {
        return $this->address;
    }
}
```

We use the custom type `point` in the `@Column` docblock annotation of the `$point` field. We will create this custom mapping type in the next chapter.

The point class:

```
<?php

namespace Geo\ValueObject;

class Point
{

    /**
     * @param float $latitude
     * @param float $longitude
     */
    public function __construct($latitude, $longitude)
    {
        $this->latitude  = $latitude;
        $this->longitude = $longitude;
    }

    /**
     * @return float
     */
    public function getLatitude()
    {
        return $this->latitude;
    }

    /**
     * @return float
     */
    public function getLongitude()
    {
        return $this->longitude;
    }
}
```

The mapping type

Now we're going to create the `point` type and implement all required methods.

```
<?php

namespace Geo\Types;

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

use Geo\ValueObject\Point;

class PointType extends Type
{
    const POINT = 'point';

    public function getName()
    {
        return self::POINT;
    }

    public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
    {
        return 'POINT';
    }

    public function convertToPHPValue($value, AbstractPlatform $platform)
    {
        list($longitude, $latitude) = sscanf($value, 'POINT(%f %f)');

        return new Point($latitude, $longitude);
    }

    public function convertToDatabaseValue($value, AbstractPlatform $platform)
    {
        if ($value instanceof Point) {
            $value = sprintf('POINT(%F %F)', $value->getLongitude(), $value->getLatitude());
        }

        return $value;
    }

    public function canRequireSQLConversion()
    {
        return true;
    }

    public function convertToPHPValueSQL($sqlExpr, AbstractPlatform $platform)
    {
        return sprintf('AsText(%s)', $sqlExpr);
    }

    public function convertToDatabaseValueSQL($sqlExpr, AbstractPlatform $platform)
    {
        return sprintf('PointFromText(%s)', $sqlExpr);
    }
}
```

We do a 2-step conversion here. In the first step, we convert the `Point` object into a string representation before saving to the database (in the `convertToDatabaseValue` method) and back into an object after fetching the value from the database (in the `convertToPHPValue` method).

The format of the string representation format is called Well-known text (WKT). The advantage of this format is, that it is both human readable and parsable by MySQL.

Internally, MySQL stores geometry values in a binary format that is not identical to the WKT format. So, we need to let MySQL transform the WKT representation into its internal format.

This is where the `convertToPHPValueSQL` and `convertToDatabaseValueSQL` methods come into play.

This methods wrap a sql expression (the WKT representation of the Point) into MySQL functions PointFromText and AsText which convert WKT strings to and from the internal format of MySQL.

Note: When using DQL queries, the `convertToPHPValueSQL` and `convertToDatabaseValueSQL` methods only apply to identification variables and path expressions in SELECT clauses. Expressions in WHERE clauses are **not** wrapped!

If you want to use Point values in WHERE clauses, you have to implement a *user defined function* for `PointFromText`.

Example usage

```php
<?php

// Bootstrapping stuff...
// $em = \Doctrine\ORM\EntityManager::create($connectionOptions, $config);

// Setup custom mapping type
use Doctrine\DBAL\Types\Type;

Type::addType('point', 'Geo\Types\PointType');
$em->getConnection()->getDatabasePlatform()->registerDoctrineTypeMapping('point', 'point');

// Store a Location object
use Geo\Entity\Location;
use Geo\ValueObject\Point;

$location = new Location();

$location->setAddress('1600 Amphitheatre Parkway, Mountain View, CA');
$location->setPoint(new Point(37.4220761, -122.0845187));

$em->persist($location);
$em->flush();
$em->clear();

// Fetch the Location object
$query = $em->createQuery("SELECT l FROM Geo\Entity\Location WHERE l.address = '1600 Amphitheatre Pa
$location = $query->getSingleResult();

/* @var Geo\ValueObject\Point */
$point = $location->getPoint();
```

9.3.17 Entities in the Session

There are several use-cases to save entities in the session, for example:

1. User object
2. Multi-step forms

To achieve this with Doctrine you have to pay attention to some details to get this working.

Merging entity into an EntityManager

In Doctrine an entity objects has to be "managed" by an EntityManager to be updateable. Entities saved into the session are not managed in the next request anymore. This means that you have to register these entities with an

EntityManager again if you want to change them or use them as part of references between other entities. You can achieve this by calling `EntityManager#merge()`.

For a representative User object the code to get turn an instance from the session into a managed Doctrine object looks like this:

```php
<?php
require_once 'bootstrap.php';
$em = GetEntityManager(); // creates an EntityManager

session_start();
if (isset($_SESSION['user']) && $_SESSION['user'] instanceof User) {
    $user = $_SESSION['user'];
    $user = $em->merge($user);
}
```

Note: A frequent mistake is not to get the merged user object from the return value of `EntityManager#merge()`. The entity object passed to merge is not necessarily the same object that is returned from the method.

Serializing entity into the session

Entities that are serialized into the session normally contain references to other entities as well. Think of the user entity has a reference to his articles, groups, photos or many other different entities. If you serialize this object into the session then you don't want to serialize the related entities as well. This is why you should call `EntityManager#detach()` on this object or implement the __sleep() magic method on your entity.

```php
<?php
require_once 'bootstrap.php';
$em = GetEntityManager(); // creates an EntityManager

$user = $em->find("User", 1);
$em->detach($user);
$_SESSION['user'] = $user;
```

Note: When you called detach on your objects they get "unmanaged" with that entity manager. This means you cannot use them as part of write operations during `EntityManager#flush()` anymore in this request.