Project Versions

stable

Search

Go

Table Of Contents

8. Working with Objects

8.1. Entities and the Identity Map

8.2. Entity Object Graph Traversal

8.3. Persisting entities

8.4. Removing entities

8.5. Detaching entities

8.6. Merging entities

8.7. Synchronization with the Database

8.7.1. Effects of Database and UnitOfWork being Out-Of-Sync

8.7.2. Synchronizing

8. Working with Objects

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

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

> It is very important to understand that only **EntityManager#flush()** ever causes write operations against the database to be executed. Any other methods such as**EntityManager#persist($entity)** or **EntityManager#remove($entity)** only notify th UnitOfWork to perform these operations during flush.
>
> Not calling **EntityManager#flush()** will lead to all changes during that request being lost.

8.1. Entities and the Identity Map

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

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

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

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

In this case the Article is accessed from the entity manager twice, but modified in between.

what kind of Query method you are using (find, Repository Finder or DQL). This is called "Identity Map" pattern, which means Doctrine keeps a map of each entity and ids that have been retrieved per PHP request and keeps returning you the same instances.

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

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

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

8.2. Entity Object Graph Traversal

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

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

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

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

    /** @ManyToOne(targetEntity="User") */
```

```
    /** @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 statemen against the articles table in the database. You can still access the associated properties author and comments and the associated objects they contain.

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

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

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

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

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

// accessing the comments as an iterator triggers the lazy-load
// retrieving ALL the comments of this article from the database
// using a single SELECT statement
foreach ($article->getComments() as $comment) {
```

```
}

// 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\Collecti
 {
    echo "This will always be true!";
}
```

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

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

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

> Traversing the object graph for parts that are lazy-loaded will easily trigger lots 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.

8.3. 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

subsequently be properly synchronized with the database when **`EntityManager#flush()`** is invoked.

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

Example:

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

> Generated entity identifiers / primary keys are guaranteed to be available after the next successful flush operation that involves the entity in question. You can not rely on a generated identifier to be available directly after invoking **`persist`**. The inverse is also true. You can not rely on a generated identifier being not available aft 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 databas result of the flush operation.
- If X is a preexisting managed entity, it is ignored by the persist operation. However, th persist operation is cascaded to entities referenced by X, if the relationships from X to t 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.

8.4 Removing entities

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

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

Example:

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

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

- If X is a new entity, it is ignored by the remove operation. However, the remove operat 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 remo operation is cascaded to entities referenced by X, if the relationships from X to these ot 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 operatio

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”.

different performance impacts.

If an association is marked as **CASCADE=REMOVE** Doctrine 2 will fetch this association. If its a Singl association it will pass this entity to ´EntityManager#remove()``. If the association is a collectic Doctrine will loop over all its elements and pass them to``EntityManager#remove()`. In both ca the cascade remove semantics are applied recursively. For large object graphs this removal stra can be very costly.
Using a DQL **DELETE** statement allows you to delete multiple entities of a type with a single comr and without hydrating these entities. This can be very efficient to delete large object graphs fron database.
Using foreign key semantics **onDelete="CASCADE"** can force the database to remove all associate 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 fore key **onDelete=CASCADE** option, because Doctrine will fetch and remove all associated entities exp nevertheless.

8.5. Detaching entities

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

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

Example:

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

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

- If X is a managed entity, the detach operation causes it to become detached. The detac operation is cascaded to entities referenced by X, if the relationships from X to these ot 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 removed entity, the detach operation is cascaded to entities referenced by X, if 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 when using the Query Result Cache).

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

8.6. Merging entities

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

Example:

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

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

- 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 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 relationship 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=MER or cascade=ALL is not specified, then navigation of the same association from X’ yields reference to a managed object Y’ with the same persistent identity as Y.

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

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

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

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

8.7. 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.

8.7.1. 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. Th 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.

8.7.2. 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 i least one persistent field has changed.

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 b 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, ar exception will be thrown as this indicates a programming error (X would be re-persisted the cascade).
- If X is detached and persist operations are configured to cascade on the relationship, a exception will be thrown (This is semantically the same as passing X to persist()).

8.7.3. 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)`**.

8.7.4. 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.

8.7.5. The cost of flushing

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

- The size of the EntityManager’s current UnitOfWork.

You can get the size of a UnitOfWork as follows:

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

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

> Do not invoke **flush** after every change to an entity or every single invocation o 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.

8.7.6. Direct access to a Unit of Work

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

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

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

8.7.7. Entity State

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

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

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

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

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

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

8.8. 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.

8.8.1. 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
```

```
$user = $em->find('MyProject\Domain\User', $id);
```

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

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

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

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

8.8.2. By Simple Conditions

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

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

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

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

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

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

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

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

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

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

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

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

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

8.8.3. 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**.

8.8.4. 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.

8.8.5. 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.

8.8.6. By DQL

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

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

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

// All users with an age between 20 and 30 (inclusive).
```

```
$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**.

8.8.7. 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**.

8.8.8. Custom Repositories

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

```
<?php
namespace MyDomain\Model;

use Doctrine\ORM\EntityRepository;

/**
 * @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
RE u.status = "admin"')
                         ->getResult();
    }
}
```

You can access your repository now by calling:

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

$admins = $em->getRepository('MyDomain\Model\User')->getAllAdminUsers();
```