CHAPTER 7

Managing Entities

Creating a New Entity

Once the domain model has been constructed, it is time to use it. A new entity can be created based simply on a new object of an entity class:

```
1   <?php
2   $newPost = new \Entity\Post();
3   $newPost->setTitle('A new post!');
4   $newPost->setContent('This is the body of the new post.');
5   $em->persist($newPost);
6   $em->flush();
```

The code shown above anticipates that `$em` references a ready-to-use entity manager. First, a new `Post` is created, then data is assigned, and the object is passed to the entity manager for persistence.

One must remember that the `persist()` method call does not yet cause an SQL `INSERT` statement to be issued. The entity is only scheduled for persistence with the next flushing. As long as no flushing has taken place, the entity is in a state called `MANAGED`, meaning that the entity manager recognizes the new entity.

Only when the `flush()` method is invoked on the entity manager is a new record written to the database. Otherwise, the entity will be lost after the script has finished.

Loading an Existing Entity

There are two main ways to load an existing entity: either by querying and retrieving it from its corresponding repository or by accessing it through an association given by another, already loaded entity.

Using a Repository

We already learned that a repository is a container for all entities of a specific type. A repository provides finder methods to search for entities based on a query. While several finder methods are available out-of-the-box, custom finder methods can also be added later on. A custom finder method can be imagined as a "quick access" to a typical query. With the help of finder methods, you can look up an entity, for example by its ID, like this:

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_7

```
<?php
$post = $em->getRepository('Entity\Post')->findOneById($id);
```

First we request the repository from the entity manager and then we execute a finder method on it. By default, four finder methods are available. To find a single entity based on its ID, use the `find()` method:

```
<?php
public function find($id, $lockMode = LockMode::NONE, $lockVersion =
null);
```

To find a single entity based on criteria, use the `findOneBy()` method:

```
<?php
public function findOneBy(array $criteria, array $orderBy = null);
```

To find all entities of a specific type, use the `findAll()` method:

```
<?php
public function findAll();
```

To find multiple entities based on criteria, use the `findBy()` method:

```
<?php
public function findBy(
        array $criteria,
        array $orderBy = null,
        $limit = null,
        $offset = null
);
```

You can define the order, limit, and offset values when using `findBy()`.

Also helpful are the so-called *magic finders*. They allow you to include search criteria directly in a finder method's name. The next two statements produce the same result:

```
<?php
$tag = $em->getRepository('Entity\Tag')->findOneBy(array('label'=>$label));
$tag = $em->getRepository('Entity\Tag')->findOneByLabel($label);
```

Adding a custom repository with individual finder methods is a two-step process. First, a new repository class needs to be set up:

```
<?php
namespace Repository;

use Doctrine\ORM\EntityRepository;

```

```
 6   class Post extends EntityRepository
 7   {
 8           public function findAllPostsWithTag($tag)
 9           {
10                   // DQL statement goes here
11           }
12   }
```

Next, Doctrine 2 needs to know about the new repository. This is done through the corresponding entity:

```
 1   <?php
 2   namespace Entity;
 3
 4   /**
 5   * @Entity(repositoryClass="Repository\Post")
 6   * @Table(name="posts")
 7   */
 8   class Post
 9   {
10           // [..]
11   }
```

That's all! The `findAllPostsWithTag()` finder method can now be easily invoked:

```
1   <?php
2   $posts = $em->getRepository('Entity\Post')->findAllPostsWithTag($tag);
```

We will learn more about DQL, the *Doctrine Query Language*, later in the book. It is used to phrase a query.

Using an Association

Let's assume we already have a loaded `User` entity available. Instead of loading the `User's` contact data by using its repository, we can also reach this entity from the given `User` entity:

```
 1   <?php
 2   namespace Entity;
 3
 4   /**
 5   * @Entity
 6   * @Table(name="users")
 7   */
 8   class User
 9   {
10          // [..]
11
```

```
12          /**
13          * @OneToOne(targetEntity="Entity\ContactData")
14          */
15          private $contactData;
16
17          // [..]
18
19          public function getContactData()
20          {
21                  return $this->contactData;
22          }
23   }
```

In this case, when calling `getContactData()`, the referenced `ContactData` entity is loaded on demand by Doctrine's proxy mechanism.

We could even make sure that the referenced `ContactData` entity is already loaded when loading the `User` itself, and save an additional database query:

```
1   <?php
2   // [..]
3   /**
4   * @OneToOne(targetEntity="Entity\ContactData", fetch="EAGER")
5   */
6   private $contactData;
```

By using `fetch="EAGER,"` we tell Doctrine 2 to always load the referenced `ContactData` entity when the `User` itself is loaded.

Loading eagerly sometimes has its advantages, especially when it's likely to access a referenced entity later in the process. If the `fetch` attribute is omitted, Doctrine 2 fetches `LAZY` by default. This means it loads the entity on first access. The third option is `EXTRA_LAZY`, which is helpful for huge datasets. Even if one decides to lazy load references, the referenced entities are still all loaded fully into RAM. Depending on the amount and size of the entities referenced, this could be a serious performance issue. When `EXTRA_LAZY` is used, several methods can be executed on the collection of referenced entities without fully loading them into the RAM right away. This is true for:

- `contains()`
- `count()`
- `offsetSet()`
- `add()`
- `count()`
- `slice()`

In this way, a pagination feature, for example, can be built without performance issues.

Changing an Existing Entity

Modifying an existing, already loaded entity is easy. All changes made to such an entity are auto-detected by Doctrine 2 when flushing the entity manager:

```
1   <?php
2   // [..]
3   $post = $em->getRepository('Entity\Post')->find(1);
4   $post->setTitle("New title");
5   $em->flush();
```

There is no need for explicitly telling Doctrine 2 again about the fact that this entity has been changed. The method `persist($entity)` does not need to be called on the entity manager again.

Removing an Entity

Removing an existing entity can easily be done through the entity manager, if a handle to a loaded entity is available:

```
1   <?php
2   $em->remove($post);
3   $em->flush();
```

The SQL statement needed to physically delete the record in the database is not issued as long as `flush()` has not yet been invoked.

Sorting an Association

When accessing entities via an association, the order of the entities retrieved is not defined. As a part of the entity mapping definitions, one can define the order of entities using the `@OrderBy` annotation. If we get back to our demo application, we could define the order of a `User's Posts` in such a way that the most recent `Post` is shown first in its list of `Posts`, simply by modifying the mapping configuration in the `User` entity:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="users")
7   */
8   class User
9   {
10          // [..]
```

```
11
12          /**
13          * @OneToMany(targetEntity="Entity\Post", mappedBy="user")
14          * @OrderBy({"id" = "DESC"})
15          */
16          private $posts;
17
18          // [..]
19      }
```

Alternatively, you can sort a collection usng PHP after retrieving the entities from the database.

Removing an Association

Removing an association is as straightforward as removing an entity:

```
1   <?php
2   $newPost = new \Entity\Post();
3   $newPost->setTitle('A new post!');
4   $newPost->setContent('This is the body of the new post.');
5   $user = $em->getRepository('Entity\User')->findOneById(1);
6   $newPost->setUser($user);
7   $em->flush();
8
9   $newPost->setUser(null);
10  $em->flush();
```

In the example above, a new `Post` entity is created. Then, an existing `User` entity is loaded and associated with the new post. After flushing, the reference has been persisted to the database. Next, we remove the association again by setting `User` to `null`. After the next flushing, the reference is gone in the database.

We need to keep in mind here that we have established a bidirectional relationship between the `User` and its `Posts`, and the `Post` entity is the owning side of the association. If, for example, we would take the `User` entity and remove the `Post` from its collection `$posts`, nothing would happen on flushing:

```
1   <?php
2   // [..]
3   $user->getPosts()->removeElement($newPost);
4   $em->flush();
```

The `removeElement()` method, which is used to remove an element from a given Doctrine 2 collection based on an entity loaded, is without the desired effect. However, there *is* an effect. While the change won't be persisted, the element has been removed from the collection in RAM. One won't find the element anymore when looking it up in the collection.

Lifecycle Events

When working with entities, several events are triggered by Doctrine 2:

`preRemove`: Occurs for a given entity before the respective `EntityManager remove` operation for that entity is executed.

`postRemove`: Occurs for an entity after the entity has been deleted. It will be invoked after the database `delete` operations.

`prePersist`: Occurs for a given entity before the respective `EntityManager persist` operation for that entity is executed.

`postPersist`: Occurs for an entity after the entity has been made persistent. It will be invoked after the database `insert` operations.

`preUpdate`: Occurs before the database `update` operations to entity data.

`postUpdate`: Occurs after the database `update` operations to entity data.

`postLoad`: Occurs for an entity after the entity has been loaded from the database.

With `loadClassMetadata, onFlush,` and `onClear`, additional events are triggered that are persistence-related, but are not specific to a single entity.

With these events available, you can hook into persistence processing, with a so-called *lifecycle callback*, which is implemented as a method of an entity class.

Let's assume we want to add login data for each `User` to our demo application. While the username can be picked by the user, the password is auto-generated on signup. This can be achieved by adding a lifecycle callback to the `User` class:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @HasLifecycleCallbacks
7   * @Table(name="users")
8   */
9   class User
10  {
11          // [..]
12
13          const GENERATED_PASSWORD_LENGTH = 6;
14
15          // [..]
16
```

```
17          /** @PrePersist */
18          public function generatePassword()
19          {
20                  for($i = 1; $i <= self::GENERATED_PASSWORD_LENGTH;
                    $i++) {
21                          $this->password .= chr(rand(65, 90)); // 65 ->
                            A, 90 -> Z
22                  }
23          }
24  }
```

First, we need to declare that lifecycle callbacks are present by using the `@HasLifecycleCallbacks` annotation. Then we add the lifecycle annotation `@PrePersist` to the `generatePassword()` method.

That's it! Now, before persisting a new entity, this method is called automatically, and the `User`'s password is auto-generated.

Cascading Operations

When creating a new entity or modifying an existing one, all operations by default affect only a single entity. A powerful, but somewhat dangerous, feature is the option to define "operation cascades." Let's consider the following example. If we delete a `User` in our demo application, we also want all of its `Posts` to be deleted. This can be achieved by adding the `cascade` attribute to the association definition:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @HasLifecycleCallbacks
7   * @Table(name="users")
8   */
9   class User
10  {
11          // [..]
12
13          /**
14          * @OneToMany(targetEntity="Entity\Post", mappedBy="user",
              cascade={"remove"})
15          */
16          private $posts;
17
18          // [..]
19  }
```

Now, when removing a user via

```
<?php
$user = $em->getRepository('Entity\User')->find($id);
$em->remove($user);
$em->flush();
```

the User is gone, and so are all its Posts. When setting cascade to value all, the cascade will be applied on other operations, such as persist, as well.

When adding the cascade attribute, the side matters. In the code shown above, all referenced Post entities are removed when a User is removed. When adding cascade to the Post entity as shown in the following code, the User entity will be removed if one of its referenced Post entities is deleted:

```
<?php
namespace Entity;

/**
* @Entity(repositoryClass="Repository\Post")
* @Table(name="posts")
*/
class Post
{
        // [..]

        /**
        * @ManyToOne(targetEntity="Entity\User", inversedBy="posts",
          cascade={"remove"})
        * @JoinColumn(name="user_id", referencedColumnName="id")
        */
        protected $user;

        // [..]
}
```

Assuming that this is not the desired behavior, it is absolutely crucial to verify the cascade configuration to prevent data loss.

Another way to achieve automatic deletion of referenced entities as shown above is *orphan removal* for one-to-one and one-to-many associations. Orphan removal means Doctrine 2 will automatically remove referenced entities without a parent entity:

```
/**
* @Entity
* @HasLifecycleCallbacks
* @Table(name="users")
*/
class User
{
        // [..]

```

```
10          /**
11          * @OneToMany(targetEntity="Entity\Post", mappedBy="user",
              orphanRemoval=true)
12          private $posts;
13
14          // [..]
15  }
```

Again, when the `User` is gone, all of its `Posts` are also gone.

While cascading operation are useful, they can be expensive. The reason is that all operations on the referenced entities happen in RAM. The entities must first be loaded and reconstructed from the database, and then modified. Depending on the size of the collection, this could be resource intensive.

Luckily, Doctrine 2 also offers "database level" cascading operations for updates and deletes via the `@JoinColumn` annotation:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity(repositoryClass="Repository\Post")
6   * @Table(name="posts")
7   */
8   class Post
9   {
10          /**
11          * @ManyToOne(targetEntity="Entity\User", inversedBy="posts")
12          * @JoinColumn(name="user_id", referencedColumnName="id",
              onDelete="CASCADE")
13          */
14          protected $user;
15
16          // [..]
17  }
```

Transactions

A transaction is an atomic unit of one or more database statements. All insert, update, or delete operations done through the entity manager are queued, as long as the `flush()` method has been called on the entity manager. Technically speaking, the queue is an implementation of the so-called *unit of work*[1] pattern. When calling `flush()`, all queued operations in the unit of work are fired against the database as a single transaction. If one of these operations fails Doctrine 2 automatically rolls back the entire transaction—that is, all operations queued—and then quits, itself, to prevent data loss due to inconsistencies.

[1] http://martinfowler.com/eaaCatalog/unitOfWork.html

Doctrine 2 offers a convenient way to wrap several database operations into a single transaction. The following code demonstrates how to "reset" a `User` in the demo application. First, the existing `User` is deleted, then a new one is created by using its current first and last name:

```
<?php
// [..]
$em->transactional(function($em) {
        $oldUser = $em->getRepository('Entity\User')->find(1);
        $newUser = new Entity\User();
        $newUser->setFirstName($oldUser->getFirstname());
        $newUser->setLastName($oldUser->getLastname());
        $em->persist($newUser);
        $em->remove($oldUser);
});
```

Both operations take effect only if no exception was thrown for either of them. Otherwise, both operations are rolled back.

Another issue may arise when two or more persons simultaneously work on the same sets of data, which is not unlikely for web applications. Doctrine 2 fully supports a strategy called *optimistic locking*. The core idea behind optimistic locking is that multiple users can all read data sets, however, whenever changing data, only the first person writing is privileged to persist its changes. All other users, now working with an outdated version of the entity, will get an exception when trying to persist their individual changes. As this strategy allows for concurrent reading operations and controls only write operations, read-intensive applications won't be slowed down, compared to a pessimistic locking strategy.

To make optimistic locking happen, Doctrine 2 allows us to add a special `integer` or `datetime` version field to an entity. The current value of this field is compared to the value loaded before, and if it doesn't match on write, an `OptimisticLockException` is thrown. If this happens, another user must have already modified the entity. A version field can be set up by adding the `@Version` annotation:

```
<?php
namespace Entity;

/**
* @Entity(repositoryClass="Repository\Post")
* @Table(name="posts")
*/
class Post
{
        // [..]

        /** @Version @Column(type="integer") */
        protected $version;

        // [..]
}
```

When creating the corresponding data schema via Doctrine 2, a column called `version` is added to the `posts` table. For every new `Post` entity, Doctrine 2 automatically assigns a value of `1`. The value is increased by one with each subsequent modification:

```
1   $post = $em->getRepository('Entity\Post')->find(1);
2   $post->setTitle("New title");
3   $em->flush();
4   $em->clear();
5   $post = $em->getRepository('Entity\Post')->find(1);
6   $post->setTitle("Again, a new title");
7   $em->flush();
```

The code shown above modifies the `Post` entity two times with an ID value of `1`. After both changes have been applied, the `value` column has a value of `3`. The `$em->clear()` statement is important here: if we omit it, we would produce an `OptimisticLockException` ourselves. The reason is that changing the title to "New title" and flushing the entity manager increments the version value for this entity by one in the database. However, the value stored in the `Post` entity object in RAM still has the old value of `1`, and therefore is now outdated—it is not being updated to the latest version value automatically. Therefore, trying to modifying the title again will fail with an `OptimisticLockException`. The same will be true if we try to persist changes to an entity which has been modified in the meantime by somebody else. How an `OptimisticLockException` situation is handled is fully up to the application developer.

Summary

In this chapter, we got our hands dirty creating and manipulating entities programatically. We also looked at the entity lifecycle and cascading features making our code even more compact, but also introduces some „magic“ and should be used with care. Transactions are another major aspect of data persistence and is covered well by Doctrine. For the most part, Doctrine already takes care of proper transaction demarcation for you: All the write operations are queued until EntityManager#flush() is invoked which wraps all of these changes in a single transaction. However, Doctrine also allows (and encourages) you to take over and control transaction demarcation yourself.