CHAPTER 5

Defining Entities

We have already learned how easy it is to map member variables to fields in a database table. Let's take a look at exactly how this works.

Mapping Formats

In general, mappings can be defined using annotations within the entities themselves, or with external XML or YAML files. In terms of performance, there is no difference between the different ways of implementing the mapping when using a cache. Therefore, using annotations is often preferred, simply because they make coding easier, as all the persistence metadata is located in the same file. In the following examples, annotations are used exclusively.

Annotations are meta information used to tell Doctrine how to persist data. Syntactically, Doctrine annotations follow the PHPDoc standard which is also used to auto-generate code documentation using compatible tools. PHPDoc uses individual DocBlocks to provide meta information on certain code elements. In several IDEs, annotations are also used to provide improved code completion, type hinting and to support debugging.

Mapping Objects to Tables

First of all, a class that acts as a template for persistent objects needs to have the `@Entity` annotation:

```
1   <?php
2   /**
3   * @Entity
4   */
5   class User
6   {
7           // [..]
8   }
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_5

A DocBlock is an extended PHP comment that begins with "/**" and has an "*" at the beginning of every line. Simply adding `@Entity` without additional configuration means that Doctrine 2 looks for a database table named `user`, in this case. If the table has another name, the table name can be configured like this:

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

An optional attribute, "`indexes`," of the `@Table` annotation can be used to create indexes automatically using Doctrine's schema tool, which we will discuss later in the book. For now, one should remember that this attribute has no meaning during runtime. Some persistence metadata is important only in conjunction with offline tools such as the schema tool, while other metadata is evaluated during runtime. An index definition to the `users` table goes like this:

```
 1   <?php
 2   /**
 3   * @Entity
 4   * @Table(name="users",
 5   *        indexes={@Index(name="search_idx", columns={"name", "email"})}
 6   * )
 7   */
 8   class User
 9   {
10          // [..]
11   }
```

The `indexes` attribute of annotation `@Table` itself has an annotation with attributes as its value. The annotation `@Index` needs the `name` attribute (name of the index) and the list of `columns` to be considered. Multiple values may be given using curly braces.

The same applies to another optional attribute called `uniqueConstraints` which defines special unique value indexes:

```
.1   <?php
 2   /**
 3   * @Entity
 4   * @Table(name="users",
 5   *         uniqueConstraints={@UniqueConstraint(name="user_unique",
 6   *                 columns={"username"})},
```

```
 7   *        indexes={@Index(name="user_idx", columns={"email"})}
 8   * )
 9   */
10   class User
11   {
12           // [..]
13   }
```

Mapping Scalar Member Variables to Fields

Mapping scalar-valued member variables (numbers or strings, for example) to a database table is super simple:

```
1   <?php
2
3   / [..]
4
5   /** @Column(type="string") */
6   private $lastName;
```

First, it's important that all member variables mapped to database fields are declared as `private` or `protected`. Doctrine 2 often applies a technique called *lazy loading*, which means that data is loaded just in the moment it is needed, but not earlier. To make this happen, Doctrine 2 implements so-called proxy objects which rely on the entities' getter methods. Therefore, one will want to make sure they are set-up.

The main annotation here is `@Column`. Its attribute type is the only mandatory attribute and is important to make persistence work correctly. If a wrong type is defined, you may lose information while saving to or reading from the database. In addition, there are many other optional attributes, such as `name`, which allows you to configure a table column name. It defaults to the name of the member variable. The `length` attribute is needed only for strings, where it defines the maximum length of a string value. It defaults to 255 if not defined differently. If one encounters truncated values in the database, a wrong length configuration often is the root cause.

The attributes `precision` and `scale` are valid only for values of type `decimal`, while `unique` ensures uniqueness, and `nullable` tells Doctrine 2 whether a `NULL` value is allowed (`true`) or not (`false`):

```
1   <?php
2
3   // [..]
4
5   /**
6   * @Column(type="string",
7   *         name="last_name", length=32, unique=true, nullable=false)
8   */
9   protected $lastName;
```

Data Types

Doctrine 2 ships with a set of data types that map PHP data types to SQL data types. Doctrine 2 data types are compatible with most common database systems:

`string`: Type that maps an SQL `VARCHAR` to a PHP `string`

`integer`: Type that maps an SQL `INT` to a PHP `integer`

`smallint`: Type that maps a database `SMALLINT` to a PHP `integer`

`bigint`: Type that maps a database `BIGINT` to a PHP `string`

`boolean`: Type that maps an SQL `boolean` to a PHP `boolean`

`decimal`: Type that maps an SQL `DECIMAL` to a PHP `double`

`date`: Type that maps an SQL `DATETIME` to a PHP `DateTime` object

`time`: Type that maps an SQL `TIME` to a PHP `DateTime` object

`datetime`: Type that maps an SQL `DATETIME/TIMESTAMP` to a PHP `DateTime` object

`text`: Type that maps an SQL `CLOB` to a PHP `string`

`object`: Type that maps an SQL `CLOB` to a PHP `object` using `serialize()` and `unserialize()`

`array`: Type that maps an SQL `CLOB` to a PHP `object` using `serialize()` and `unserialize()`

`float`: Type that maps an SQL Float (Double Precision) to a PHP `double`. IMPORTANT: Type `float` works only with locale settings that use decimal points as separators.

Custom data types If needed, you can define custom data types, mappings between PHP data types and SQL data types. The official documentation covers this topic in depth.[1]

[1] http://docs.doctrine-project.org/en/latest/reference/basic-mapping.html#custom-mapping-types

Entity Identifier

An entity class must define a unique identifier. The `@Id` annotation is used for this:

```
1   <?php
2
3   // [..]
4
5   /**
6   * @Id @Column(type="integer")
7   */
8   private $id;
```

If the `@GeneratedValue` annotation is used as well, the unique identifier doesn't need to be given by the application developer but is assigned automatically by the database system. The way this value is generated depends on the strategy defined. By default, Doctrine 2 identifies the best approach on its own. When using MySQL, for instance, Doctrine 2 uses the `auto_increment` function. There are additional strategies that can be used; however, they usually don't work across platforms and therefore should be avoided.

By the way, the `@Id` annotation can be used with multiple member variables to define composite keys. Clearly, the `@GeneratedValue` annotation then cannot be used anymore. In this case, the application developer needs to take care in assigning unique keys.

Inheritance

Doctrine 2 supports persistence of hereditary structures with three different strategies:

- Single Table Inheritance
- Class Table Inheritance
- Mapped Superclass

Single Table Inheritance

The *single table inheritance* strategy uses a single database table with a so-called "discriminator column" to differentiate between the different types. All classes of a hierarchy go into the same table. Let's say our demo application differentiates between different types of posts. The supported types are:

`Post`: A simple, text based post

`ImagePost`: A post with text and image

`VideoPost`: A post with text and video

Both ImagePost and VideoPost extend Post and add an image or video URL. To set up this hierarchy, the topmost Post class holds the configuration needed as well as all member variables and associations that are shared between the three types:

```
<?php
namespace Entity;

/**
* @Entity(repositoryClass="Repository\Post")
* @Table(name="posts")
* @InheritanceType("SINGLE_TABLE")
* @DiscriminatorColumn(name="discr", type="string")
* @DiscriminatorMap({"text"="Post","video"="VideoPost","image"=
"ImagePost"})
*/
class Post
{
        // member variables and associations shared between all types
}
```

Since the Post itself can be an entity, it has the typical entity-related annotations. In addition, via the @InheritanceType annotation, it configures single table inheritance. The @DiscriminatorColumn annotation is used to label the column which indicates the type persisted. Lastly, @DiscriminatorMap defines the values that can be present in the discriminator column. In our case, Doctrine 2 writes "text" into the discriminator column if the entity is a Post. If the entity is a VideoPost, Doctrine 2 writes "video," and if the entity is an ImagePost, the value used is "image."

The ImagePost itself is straightforward:

```
<?php
namespace Entity;

/**
* @Entity
*/
class ImagePost extends Post
{
        /** @Column(type="string") */
        protected $imageUrl;

        /**
        * @param mixed $imageUrl
        */
        public function setImageUrl($imageUrl)
        {
                $this->imageUrl = $imageUrl;
        }

```

```
20          /**
21          * @return mixed
22          */
23          public function getImageUrl()
24          {
25                  return $this->imageUrl;
26          }
27  }
```

The same is true for the VideoPost:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   */
7   class VideoPost extends Post
8   {
9           /** @Column(type="string") */
10          protected $videoUrl;
11
12          /**
13          * @param mixed $videoUrl
14          */
15          public function setVideoUrl($videoUrl)
16          {
17                  $this->videoUrl = $videoUrl;
18          }
19
20          /**
21          * @return mixed
22          */
23          public function getVideoUrl()
24          {
25                  return $this->videoUrl;
26          }
27  }
```

The data structure used by Doctrine 2 looks like this:

Col	Type	Length	NULL?	KEY	Extras
Id	INT	11		PRI	auto_ increment
user_id	INT	11	YES	MUL	
category_id	INT	11	YES	MUL	
title	VARCHAR	255			
content	LONGTEXT				
discr	VARCHAR	255			
videoUrl	VARCHAR	255	YES		
imageUrl	VARCHAR	255	YES		

Now all three types of entities can easily be persisted. The downside of this approach is that, depending on how complex the hierarchy is and how different the types involved are, the table grows and grows, with many fields never used by certain types. In our case, the `videoUrl` won't ever be used by entities of type `ImagePost`. The `imageUrl` won't be used by `VideoPost` entities, and for `Post` entities, both fields will remain blank.

The upside is that this strategy—due to its very nature—allows for very efficient queueing across all types in the hierarchy without any joins.

Class Table Inheritance

While single table inheritance uses only one table for all types, with *class table inheritance* each class in the hierarchy is mapped to several tables. While this gives the best flexibility of the three strategies, this also means that reconstructing entities will require joins in the database. Since joins are usually expensive operations, the performance with class table inheritance is not as good as with single table inheritance.

The main difference in configuration is in `Post`:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity(repositoryClass="Repository\Post")
6  * @Table(name="posts")
7  * @InheritanceType("JOINED")
8  * @DiscriminatorColumn(name="discr", type="string")
9  * @DiscriminatorMap({"text"="Post","video"="VideoPost","image"=
   "ImagePost"})
10 */
```

```
11  class Post
12  {
13          // member variables and associations shared between all types
14  }
```

Instead of inheritance type "SINGLE_TABLE," now "JOINED" is used. This tells Doctrine 2 to apply the class table inheritance strategy. Since tables will be used for each type, configuring the table names may be useful for subclasses:

```
 1  <?php
 2  namespace Entity;
 3
 4  /**
 5  * @Entity
 6  * @Table(name="posts_image")
 7  */
 8  class ImagePost extends Post
 9  {
10          /** @Column(type="string") */
11          protected $imageUrl;
12
13          /**
14          * @param mixed $imageUrl
15          */
16          public function setImageUrl($imageUrl)
17          {
18                  $this->imageUrl = $imageUrl;
19          }
20
21          /**
22          * @return mixed
23          */
24          public function getImageUrl()
25          {
26                  return $this->imageUrl;
27          }
28  }
```

The table name `posts_image` is handy for `ImagesPost` entities. We add the same style of configuration to `VideoPost` entities as well:

```
1  <?php
2  namespace Entity;
3
4  /**
5  * @Entity
6  * @Table(name="posts_video")
```

```
 7   */
 8   class VideoPost extends Post
 9   {
10           /** @Column(type="string") */
11           protected $videoUrl;
12
13           /**
14           * @param mixed $videoUrl
15           */
16           public function setVideoUrl($videoUrl)
17           {
18                   $this->videoUrl = $videoUrl;
19           }
20
21           /**
22           * @return mixed
23           */
24           public function getVideoUrl()
25           {
26                   return $this->videoUrl;
27           }
28   }
```

When letting Doctrine 2 create the database schema, we will end up with three tables for this hierarchy, while the `posts` table holds the main information:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	auto_ increment
user_id	INT	11	YES	MUL	
category_id	INT	11	YES	MUL	
title	VARCHAR	255			
content	LONGTEXT				
discr	VARCHAR	255			

The `posts_video` table holds only the specifics of this type:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	
videoUrl	VARCHAR	255	YES		

The same is true for `posts_image`:

Col	Type	Length	NULL?	KEY	Extras
id	INT	11		PRI	
imageUrl	VARCHAR	255	YES		

By evaluating the foreign key reference, entities of type `ImagePost` and `VideoPost` can be reconstructed by table joins.

Mapped Superclass

Lastly, a *mapped superclass* strategy can be used. The main difference from the other two strategies already discussed is that the superclass itself cannot be an entity. Taking the example above, this means that class `Post` cannot itself be an entity. It simply provides member variables and associations to be inherited by its subclasses.

To configure the mapped superclass strategy means adding the `@MappedSuperclass` annotation to the `Post` class:

```
1   <?php
2   namespace Entity;
3
4   /** @MappedSuperclass */
5   class Post
6   {
7                   // member variables and associations shared between all types
8   }
```

The two subclasses, `ImagePost` and `VideoPost`, do not need to be changed. However, with the mapped superclass strategy applied, we cannot persist objects of type `Post` anymore, which mainly consisted of text. We could fix this by adding another new type called `TextPost`:

```
 1   <?php
 2   namespace Entity;
 3
 4   /**
 5   * @Entity
 6   * @Table(name="posts_text")
 7   */
 8   class TextPost extends Post
 9   {
10           /**
11           * @Id @Column(type="integer")
12           * @GeneratedValue
13           */
```

```
14          protected $id;
15
16  }
```

Now we have three concrete entity types and one superclass that is not an entity. This results in three tables in the database: `posts_text`, `posts_images` and `posts_video`.

As shown above, it's important that each entity specifies its `ID` column, while the mapped superclass does not. It only imparts member variables and associations, but not the ID column definition.

When inheriting attributes or associations as shown above, one may want to overwrite certain definitions.

Let's assume we don't want to talk about `content` in a `VideoPost`, but call it `closed_caption`. This requires the following configuration:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
6   * @Table(name="posts_video")
7   * @AttributeOverrides({
8   *     @AttributeOverride(name="content",
9   *         column=@Column(
10  *             name = "closed_caption",
11  *             type = "text"
12  *         )
13  *     )
14  * })
15  */
16  class VideoPost extends Post
17  {
18          // [..]
19  }
```

While Doctrine 2 requires member variables to be either `private` or `protected`, it's important to remember that member variables need to be `protected` here; otherwise they won't be available in the subclass. When overwriting the property `content` with `closed_caption` as shown above, this leads to a `closed_caption` column in the database for this type of entity.

Also, entities of type `VideoPost` are organized in categories called `channels`. We can overwrite the association as shown below:

```
1   <?php
2   namespace Entity;
3
4   /**
5   * @Entity
```

```
 * @Table(name="posts_video")
 * @AssociationOverrides({
 *     @AssociationOverride(name="category",
 *          joinColumns=@JoinColumn(
 *              name="channel_id",  referencedColumnName="id"
 *        )
 *    )
 * })
 */
class VideoPost extends Post
{
          // [..]
}
```

Summary

In this chapter, we learned all about entities, the most important aspect of ORM and Doctrine. Designing entities properly can be fiddly at times, but Doctrine makes it as simple as possible to lay out the foundation for persisting PHP objects. Since entities alone already are a great thing to have, it get's even better, once we start interlinking them by using associations, which we will cover next.