CHAPTER 11

Advanced Topics

Framework Integrations

If one of the popular PHP frameworks is used for an application, integration of Doctrine mostly can be easily done. As an example, we will walk through the process of integrating Doctrine into a Zend Framework 2 application. If you are using a different framework, the official framework documentation or a quick web search usually brings up a suited tutorial.

The easiest way to use to Doctrine 2 with Zend Framework 2 (ZF2) is a Composer based installation. The ZF2 module doctrine-orm-module[1] not only ships the glue code to make both libraries work together, but also ensures that the Doctrine 2 library itself is downloaded and installed in an existing ZF2 application. One simply needs to add the following line to the `require` block of the projects' `composer.json` file:

```
1   "doctrine/doctrine-orm-module": "dev-master"
```

You must also add the following line above the `require` block:

```
1   "minimum-stability": "alpha"
```

If this configuration is missing, Composer might refuse to install the module. In fact, with this configuration, we tell Composer to install even non-stable modules.

The following commands start the download and the installation process:

```
1   $ php composer.phar update
```

As you can see, a whole bunch of other ZF2 modules and additional libraries are downloaded. Last but not least, two ZF2 modules must be activated via the `application.config.php`:

```
1   <?php
2   return array(
3           'modules' => array(
4                   'Application',
```

[1] https://github.com/doctrine/DoctrineORMModule

M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_11

```
                'DoctrineModule',
                'DoctrineORMModule'
        ),
        'module_listener_options' => array(
                'config_glob_paths' => array(
                        'config/autoload/{,*.}{global,local}.php',
                ),
                'module_paths' => array(
                        './module',
                        './vendor',
                ),
        ),
);
```

The modules register several services in Main Service Manager, all starting with the label "doctrine," such as `doctrine.cache.apc` and `doctrine.sqlloggercollector.ormdefault`.

The entity manager can be obtained using a long-winded label:

```
<?php
// [..]
$this->getServiceLocator()->get('doctrine.entitymanager.orm_default');
```

But before this will work, some additional configuration is needed. Doctrine 2 needs to know where the entity classes are located and what the caching strategy looks like. The following example from the `module.php` tells Doctrine that the mappings are given as annotation, the entities are located in `__DIR__ . '/../src/Application /Entity'`, and caching takes place via PHP arrays:

```
<?php
// [..]
'doctrine' => array(
        'driver' => array(
                'my_annotation_driver' => array(
                        'class' => 'Doctrine\ORM\Mapping\Driver\
                        AnnotationDriver',
                'cache' => 'array',
                'paths' => array(__DIR__ . '/../src/Application/
                Entity')
                )
        )
)
// [..]
```

To allow other ZF2 modules to provide entities to the application, a so-called Driver Chain allows us to combine multiple entity sources even with different mapping formats and caching strategies:

```
<?php
// [..]
'doctrine' => array(
        'driver' => array(
                'my_annotation_driver' => array(
                        'class' => 'Doctrine\ORM\Mapping\Driver\
                        AnnotationDriver',
                        'cache' => 'array',
                        'paths' => array(__DIR__ . '/../src/
                        Application/Entity')
                ),
                'orm_default' => array(
                        'drivers' => array(
                'Application\Entity' => 'my_annotation_driver'
            )
        )
        )
)
// [..]
```

Now the database connection must be configured. Usually, a dedicated config file in the autoload folder is used, e.g. `db.local.php`:

```
<?php
return array(
         'doctrine' => array(
                 'connection' => array(
                         'orm_default' => array(
                                 'driverClass' => 'Doctrine\DBAL\Driver\
                                 PDOMySql\Driver',
                                 'params' => array(
                                         'host' => 'localhost',
                                         'port' => '3306',
                                         'user' => 'username',
                                         'password' => 'password',
                                         'dbname' => 'database',
                                 )
                         )
                 )
         ),
);
```

Once done, we can start dealing with entities (e.g. a Product entity) through Doctrine 2:

```
<?php
namespace Application\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Entity
* @ORM\Table(name="product")
*/
class Product
{
        /**
        * @ORM\Id @ORM\Column(type="integer")
        * @ORM\GeneratedValue
        */
        protected $productId;

        /** @ORM\Column(type="string", nullable=true) */
        protected $name;

        /** @ORM\Column(type="integer") */
        protected $stock;

        /** @ORM\Column(type="string", nullable=true) */
        protected $description;

        /** @ORM\Column(type="string", nullable=true) */
        protected $features;

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

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

        public function setFeatures($features)
        {
                $this->features = $features;
        }

        public function getFeatures()
        {
```

```
47                      return $this->features;
48              }
49
50              public function setProductId($productId)
51              {
52                      $this->productId = $productId;
53              }
54
55              public function getProductId()
56              {
57                      return $this->productId;
58              }
59
60              public function setName($name)
61              {
62                      $this->name = $name;
63              }
64
65              public function getName()
66              {
67                      return $this->name;
68              }
69
70              public function setStock($stock)
71              {
72                      $this->stock = $stock;
73              }
74
75              public function getStock()
76              {
77                      return $this->stock;
78              }
79      }
```

From a controller, a finder method provided by a repository can be accessed like this:

```
1   <?php
2   // [..]
3   $this->getServiceLocator()->get('doctrine.entitymanager.orm_default')
4           ->getRepository('Application\Entity\Product')
5           ->findOneByProductId($id);
6   // [..]
```

The Doctrine 2 command line tools are available as well. You simply bring up a command line and change to the project's root folder:

```
1   $ php vendor/bin/doctrine-module orm:validate-schema
```

Native SQL Statements

Doctrine 2 ships with a special class called `NativeQuery` that allows you to execute native SQL select statements and to map the results returned to entity objects. The same operations that work out-of-the-box with Doctrine 2 can be implemented by hand in cases where native queries are needed or where they are the better solution to a problem. `NativeQuery` allows you to retrieve "raw data" and then subsequently work with entity objects. The official documentation[2] holds further information about how to properly implement native SQL statements.

Lastly, in general, one needs to remember that it's always possible to execute arbitrary SQL statements via the underlying database connection:

```
1   <?php
2   // [..]
3   $em->getConnection()->exec('DELETE FROM posts');
```

While this is possible, it should always be the last resort. It bypasses the Entity Manager and might produce hard-to-debug issues and data inconsistencies.

Doctrine 2 Extensions

While Doctrine 2 already ships with tons and tons of features, there is even more. In the Doctrine 2 extension repository[3] on GitHub, you will find several extensions with solutions to typical problems which otherwise must be solved individually by each application developer again and again:

`Tree`: Automates the tree handling process and adds some tree-specific functions on repositories.

`Translatable`: Gives a very handy solution for translating records into different languages.

`Sluggable`: Takes a specified field from an entity and makes it compatible for URLs.

`Timestampable`: Updates date fields on creates, updates, and even property changes.

`Blameable`: Updates string or reference fields on creates, updates, and even property changes with a string or object (e.g. user).

[2] http://docs.doctrine-project.org/en/latest/reference/native-sql.html
[3] https://github.com/l3pp4rd/DoctrineExtensions

`Loggable`: Helps tracking changes and history of objects, also supports version management.

`Sortable`: Makes any entity sortable.

`Translator`: Supports handling translations.

`Softdeleteable`: Allows you to mark entities as deleted, without physically deleting them.

`Uploadable`: Provides file upload handling to entity fields.

`References`: Supports linking entities in Documents and vice versa.

Summary

Congratulations! You made to it to the end of this book. I thank you very much for buying and reading my book on Doctrine 2 ORM. I believe you now have all knowledge and tools at hand to use Doctrine 2 in your own applications. As in every technical book, we didn't cover all of the features of Doctrine and it might be worth to continue learning by reading the official documentation. The contents of this book will help you to grasp other features and implementation details of Doctrine 2 not covered in this book.

Again, thanks for reading my book and happy coding!