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 prerequisites for the tutorial that have to be installed:

- PHP (latest stable version)
- Composer Package Manager (`Install Composer <https://getcomposer.org/doc/00-intro.md>`_)

The code of this tutorial is available on Github.

What is Doctrine?

Doctrine ORM is an object-relational mapper (ORM) for PHP 7.1+ 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**, unless it does 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 their 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` inside that directory with the following contents:

```
{
    "require": {
        "doctrine/orm": "^2.6.2",
        "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, into the `vendor` directory.

Add the following directories:

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

The YAML driver is deprecated and will be removed in version 3.0. It is strongly recommended to switch to one of the other mappings.

Obtaining the EntityManager

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

<?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; $proxyDir = null; $cache = null; $useSimpleAnnotationReader = false; $config = Setup::createAnnotationMetadataConfiguration(array(__DIR__."/src"), $isDevMode, $proxyDir, $cache, $useSimpleAnnotationReader); // 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 YAML driver is deprecated and will be removed in version 3.0. It is strongly recommended to switch to one of the other mappings.

It is recommended not to use the SimpleAnnotationReader because its usage will be removed for version 3.0.

The `require_once` statement sets up the class autoloading for Doctrine and its dependencies using Composer's autoloader.

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 this case, we'll use 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

Doctrine has a command-line interface that allows you to access the SchemaTool, a component that can generate a relational database schema based entirely on the defined entity classes and their metadata. For this tool to work, a `cli-config.php` file must exist in the project root directory:

<?php // cli-config.php require_once "bootstrap.php"; return \Doctrine\ORM\Tools\Console\ConsoleRunner::createHelperSet($entityManager);

Now call the Doctrine command-line tool:

```
vendor/bin/doctrine orm:schema-tool:create
```

Since we haven't added any entity metadata in `src` yet, you'll see a message stating No Metadata Classes to process. In the next section, we'll create a Product entity along with the corresponding metadata, and run this command again.

Note that as you modify your entities' metadata during the development process, you'll need to update your database schema to stay in sync with the metadata. You can easily recreate the database using the following commands:

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

```
vendor/bin/doctrine orm:schema-tool:create
```

Or you can 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. This is a cornerstone of the `Doctrine\DBAL` package, which can even be used without the Doctrine ORM package.

Starting with the Product Entity

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

```
<?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; } }
```

When creating entity classes, all of the fields should be `protected` or `private` (not `public`), with getter and setter methods for each one (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 an Entity can be configured using DocBlock annotations directly in the Entity class itself, or in an external XML or YAML file. This Getting Started guide will demonstrate metadata mappings using all three methods, but you only need to choose one.

- *PHP*

```
<?php // src/Product.php use Doctrine\ORM\Mapping as ORM; /** * @ORM\Entity * @ORM\
Table(name="products") */ class Product { /** * @ORM\Id * @ORM\Column(type="integer") *
@ORM\GeneratedValue */ protected $id; /** * @ORM\Column(type="string") */ protected
$name; // .. (other code) }
```

- *XML*

The YAML driver is deprecated and will be removed in version 3.0. It is strongly recommended to switch to one of the other mappings.

```
# 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 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. It has a `generator` tag nested inside, which specifies that the primary key generation mechanism should automatically use the database platform's 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 and its metadata, let's update the database schema:

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

Specifying both flags `--force` and `--dump-sql` will cause the DDL statements to be executed and then printed to the screen.

Now, we'll create a new script to insert products into the database:

```
<?php // create_product.php <name> 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` class 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 initiate a transaction to actually *perform* the insertion, you have to explicitly call `flush()` on the `EntityManager`.

This distinction between persist and flush is what allows the aggregation of all database 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 in which writes are performed on each entity in isolation.

Next, we'll fetch a list of all the Products in the database. 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 type of entity. It is provided by Doctrine and contains some finder methods like `findAll()`.

Let's continue by creating a script to display the name of a product based on its ID:

```
<?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());
```

Next we'll update a product's name, given its id. This simple example will help demonstrate Doctrine's implementation of the UnitOfWork pattern. Doctrine keeps track of all the entities that were retrieved from the Entity Manager, and can detect when any of those entities' properties have been modified. As a result, rather than needing to call `persist($entity)` for each individual entity whose properties were changed, a single call to `flush()` at the end of a request is sufficient to update the database for all of the modified entities.

```
<?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 example by creating the `Bug` and `User` classes. We'll store them in `src/Bug.php` and `src/User.php`, respectively.

```
<?php // src/Bug.php use Doctrine\ORM\Mapping as ORM; /** * @ORM\Entity * @ORM\
Table(name="bugs") */ class Bug { /** * @ORM\Id * @ORM\Column(type="integer") * @ORM\
GeneratedValue * @var int */ protected $id; /** * @ORM\Column(type="string") * @var string */
protected $description; /** * @ORM\Column(type="datetime") * @var DateTime */ protected
$created; /** * @ORM\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 use Doctrine\ORM\Mapping as ORM; /** * @ORM\Entity * @ORM\
Table(name="users") */ class User { /** * @ORM\Id * @ORM\GeneratedValue * @ORM\
Column(type="integer") * @var int */ protected $id; /** * @ORM\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 we've seen so far are of simple types (integer, string, and datetime). But now, we'll add properties that will store objects of specific *entity types* in order to model the relationships between different entities.

At the database level, relationships between entities are represented by foreign keys. But with Doctrine, you'll never have to (and never should) work with the foreign keys directly. You should only work with objects that represent foreign keys through their own identities.

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; class User { // ...
(previous code) protected $reportedBugs; protected $assignedBugs; public function __construct() {
$this->reportedBugs = new ArrayCollection(); $this->assignedBugs = new ArrayCollection(); } }
```

Whenever an entity is created from the database, a `Collection` implementation of the type `PersistentCollection` will be injected into your entity instead of an `ArrayCollection`. This helps Doctrine ORM understand the changes that have happened to the collection that are noteworthy for persistence.

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 ORM. These assumptions are not unique to Doctrine ORM but are best practices in handling database relations and Object-Relational Mapping.

- 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-one relation, the Many-side is the owning side by default because it holds the foreign key. Accordingly, the One-side is the inverse side by default.
- In a many-to-one relation, the One-side can only be the owning side if the relation is implemented as a ManyToMany with a join table, and the One-side is restricted to allow only UNIQUE values per database constraint.
- 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 side is allowed to be the owning side.
- 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.

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 // src/Bug.php class Bug { // ... (previous code) protected $engineer; protected $reporter;
public function setEngineer(User $engineer) { $engineer->assignedToBug($this); $this->engineer
= $engineer; } public function setReporter(User $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) protected $reportedBugs; protected
$assignedBugs; public function addReportedBug(Bug $bug) { $this->reportedBugs[] = $bug; }
public function assignedToBug(Bug $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 a uni-directional ManyToMany relation in the database that points from Bugs to Products.

```
<?php // src/Bug.php class Bug { // ... (previous code) protected $products; public function
assignToProduct(Product $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 `Bug` entity, as we did for the `Product` before:

- *PHP*

```
<?php // src/Bug.php use Doctrine\ORM\Mapping as ORM; /** * @ORM\Entity * @ORM\
Table(name="bugs") */ class Bug { /** * @ORM\Id * @ORM\Column(type="integer") * @ORM\
GeneratedValue */ protected $id; /** * @ORM\Column(type="string") */ protected $description; /
** * @ORM\Column(type="datetime") */ protected $created; /** * @ORM\
Column(type="string") */ protected $status; /** * @ORM\ManyToOne(targetEntity="User",
inversedBy="assignedBugs") */ protected $engineer; /** * @ORM\
ManyToOne(targetEntity="User", inversedBy="reportedBugs") */ protected $reporter; /** *
@ORM\ManyToMany(targetEntity="Product") */ protected $products; // ... (other code) }
```

- *XML*

The YAML driver is deprecated and will be removed in version 3.0. It is strongly recommended to switch to one of the other mappings.

```
# 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.

Finally, we'll add metadata mappings for the `User` entity.

- *PHP*

```
<?php // src/User.php use Doctrine\ORM\Mapping as ORM; /** * @ORM\Entity * @ORM\
Table(name="users") */ class User { /** * @ORM\Id * @ORM\GeneratedValue * @ORM\
Column(type="integer") * @var int */ protected $id; /** * @ORM\Column(type="string") * @var
string */ protected $name; /** * @ORM\OneToMany(targetEntity="Bug", mappedBy="reporter")
* @var Bug[] An ArrayCollection of Bug objects. */ protected $reportedBugs; /** * @ORM\
OneToMany(targetEntity="Bug", mappedBy="engineer") * @var Bug[] An ArrayCollection of
Bug objects. */ protected $assignedBugs; // .. (other code) }
```

- *XML*

The YAML driver is deprecated and will be removed in version 3.0. It is strongly recommended to switch to one of the other mappings.

```
# config/yaml/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.

Update your database schema by running:

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

Implementing more Requirements

So far, we've seen the most basic features of the metadata definition language. To explore additional functionality, let's first create new `User` entities:

```
<?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 necessary data to create a new Bug entity:

```
<?php // create_bug.php <reporter-id> <engineer-id> <product-ids> require_once "bootstrap.php";
$reporterId = $argv[1]; $engineerId = $argv[2]; $productIds = explode(",", $argv[3]); $reporter =
$entityManager->find("User", $reporterId); $engineer = $entityManager->find("User",
$engineerId); if (!$reporter || !$engineer) { echo "No reporter and/or engineer found for the given
id(s).\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 as follows:

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

See how simple it is to relate a Bug, Reporter, Engineer and Products? Also recall that thanks to the UnitOfWork pattern, Doctrine will detect these relations and update all of the modified entities in the database automatically when `flush()` is called.

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 // 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
```

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.

If you need to build your query dynamically, 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 // 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 // show_bug.php <id> 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
```

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 <user-id> 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 <bug-id> 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 $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 $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 // 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 DESC"; $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 use Doctrine\ORM\Mapping as ORM; /** * @ORM\Entity(repositoryClass="BugRepository") * @ORM\Table(name="bugs") **/ class Bug { //... }

- *XML*

The YAML driver is deprecated and will be removed in version 3.0. It is strongly recommended to switch to one of the other mappings.

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.

The method `count()` takes an array of fields or association keys and the values to match against. This provides you with a convenient and lightweight way to count a resultset when you don't need to deal with it:

<?php $productCount = $entityManager->getRepository(Product::class) ->count(['name' => $productName]);

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.