Doctrine 2 & PHP

Damir Mitrović
damir@edev.rs

Damir Mitrović

- Software developer since 2000
- Worked in companies:
  - Active Media/Autview
  - DMV/Yipee inc
  - Loopia/Troxo/Atomia
  - youngculture/enjoy.ing
  - edev

edev

- Web & mobile application development
- References:
  - CRS AG - http://www.crsag.com/
  - Gritness/Under Armour - https://www.crunchbase.com/organization/gritness
  - Itrust - http://www.itrust.ch/
  - Swissbilling - http://swissbilling.ch/
  - Medici - https://medici.md/
  - Proseller AG - https://www.proseller.ch/
  - Homecaredelivered – HCD http://www.hcd.com/

Doctrine

- PHP libraries primarily focused on database storage and object mapping.
- The core projects are the Object Relational Mapper (ORM) and the Database Abstraction Layer (DBAL) it is built upon.

- Libraries
  - Annotations
  - Cache
  - Collections
  - Common
  - **Database Abstraction Layer**
  - Inflector
  - Lexer
  - Migrations
  - MongoDB Abstraction Layer
  - MongoDB Object Document Mapper
  - **Object Relational Mapper**
  - PHPCR ODM

https://www.doctrine-project.org/

Who uses doctrine

- Symfony - tight integration with Doctrine.
- Drupal 8 – built on top of Symfony
- Laravel – Eloquent ORM as default, replaces Eloquent via laravel-doctrine/orm package
- PHP Unit
- Zend/Yii/Codeigniter/CakePHP via third party packages

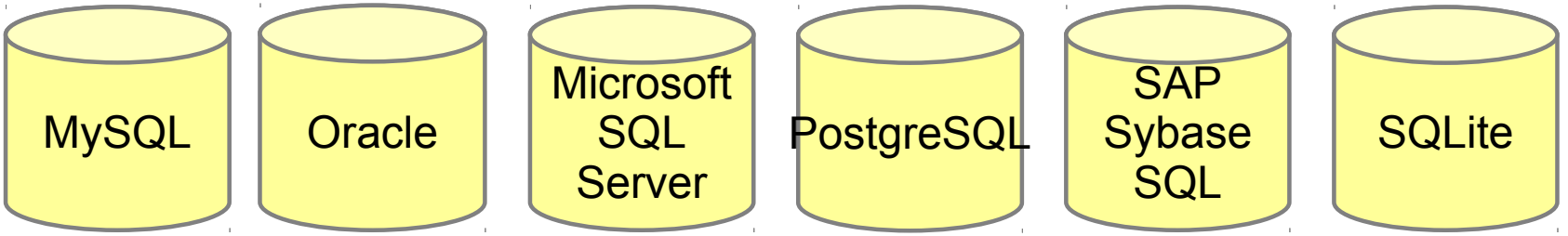
The Layers

ORM - object relational mapper with SQL dialect called Doctrine Query Language (DQL)

DBAL - database schema introspection and manipulation through an OO API.

PDO - data-access abstraction layer - use the same functions to issue queries and fetch data

PDO_CUBRID (Cubrid) PDO_DBLIB (FreeTDS / Microsoft SQL Server / Sybase) PDO_FIREBIRD (Firebird)
PDO_IBM (IBM DB2) PDO_INFORMIX (IBM Informix Dynamic Server) PDO_MYSQL (MySQL 3.x/4.x/5.x)
PDO_OCI (Oracle Call Interface) PDO_ODBC (ODBC v3 IBM DB2, unixODBC and win32 ODBC)
PDO_PGSQL (PostgreSQL) PDO_SQLITE (SQLite 3 and SQLite 2)
PDO_SQLSRV (Microsoft SQL Server / SQL Azure) PDO_4D (4D)



Installation and configuration

- Requirements:
  - php
  - composer
- Installation is done by composer
- composer install

```
{
   "require": {
      "doctrine/orm": "2.*",
      "symfony/yaml": "2.*",
      "fzaninotto/faker": "^1.7",
      "symfony/console": "^3.4"
   },
   "autoload": {
      "psr-0": {
         "": "src/"
      }
   }
}
```

Composer

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.

It manages libraries on a per-project basis, installing them in a directory (e.g. vendor) inside your project. By default it does not install anything globally.

Composer

- Composer is strongly inspired by node's npm and ruby's bundler.
- Suppose:
  - You have a project that depends on a number of libraries.
  - Some of those libraries depend on other libraries.
- Composer:
  - Enables you to declare the libraries you depend on.
  - Finds out which versions of which packages can and need to be installed, and installs them (meaning it downloads them into your project).

Class loading

- Autoloading is taken care of by Composer.
- You just have to include the composer autoload file in your project.

```
<?php
// bootstrap.php
// Include Composer Autoload (relative to project root).
require_once "vendor/autoload.php";
```

DBAL layer

- The Doctrine database abstraction & access layer (DBAL) offers a lightweight and thin runtime layer around a PDO-like API and a lot of additional, horizontal features like database schema introspection and manipulation through an OO API.
- The following database vendors are currently supported:
  - MySQL
  - Oracle
  - Microsoft SQL Server
  - PostgreSQL
  - SAP Sybase SQL Anywhere
  - SQLite

```
<?php
use Doctrine\DBAL\Configuration;
use Doctrine\DBAL\DriverManager;

require_once "vendor/autoload.php";

$config = new Configuration();

$connectionParams = array(
    'dbname' => 'doctrine',
    'user' => 'doctrine',
    'password' => 'doctrine12345',
    'host' => 'localhost',
    'driver' => 'pdo_mysql',
);
$connection =
DriverManager::getConnection($connectionParams, $config);

$schemaManager = $connection->getSchemaManager();

if (!$schemaManager->tablesExist('books')) {

    $bookTable = new \Doctrine\DBAL\Schema\Table('books');
    $bookTable->addColumn("id", "integer", array("unsigned"
=> true, "autoincrement" => true));
    $bookTable->addColumn("isbn", "string");
    $bookTable->addColumn("title", "text");
    $bookTable->addColumn("author", "text");
    $bookTable->setPrimaryKey(array("id"));

    $schemaManager->createTable($bookTable); // save to DB
}
```

DBAL layer

- listDatabases()
- listSequences()
- listTableColumns()
- listTableDetails()
- listTableForeignKeys()
- listTableIndexes()
- listTables()
- listViews()
- createSchema()

```
<?php
use Doctrine\DBAL\Configuration;
use Doctrine\DBAL\DriverManager;
use Symfony\Component\Console\Output\ConsoleOutput;
use Symfony\Component\Console\Helper\Table;

require_once "vendor/autoload.php";

$config = new Configuration();

$output = new ConsoleOutput();
$tableOutput = new Table($output);

// commented from previous example
$connectionParams  = array(...);

$connection = DriverManager::getConnection($connectionParams, $config);

$schemaManager = $connection->getSchemaManager();

$tables = $schemaManager->listTables();

foreach ($tables as $table) {
   renderTable($tableOutput, $table);
}

function renderTable($tableOutput, $tableName)
{
   $tableOutput->setHeaders(array($tableName->getName()));
   foreach ($tableName->getColumns() as $column) {
      $tableOutput->addRow(array($column->getName()));
   }
   $tableOutput->render();
}
```

PDO layer – insert records to db

- PDO is an acronym for PHP Data Objects.
- PDO is a lean, consistent way to access databases. This means developers can write portable code much easier.
- PDO is data access layer which uses a unified API (Application Programming Interface).

```
<?php
//pdo_populate_table.php
require_once "vendor/autoload.php";

$user = 'doctrine';
$pass = 'doctrine12345';

$dsn = "mysql:host=localhost;dbname=doctrine";
$opt = [
    PDO::ATTR_ERRMODE            =>
PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC
];
$pdo = new PDO($dsn, $user, $pass, $opt);

$stmt = $pdo->prepare("INSERT INTO `books` (`id`, `isbn`,
`title`, `author`)
VALUES (NULL, :isbn, :title, :author)");

$faker = Faker\Factory::create();

for ($i = 0; $i < 10; $i++) {
    $isbn = $faker->ean13();
    $sentence = $faker->sentence(5);
    $title = substr($sentence, 0, strlen($sentence) - 1);
    $author = $faker->name;

    $stmt->bindValue(":isbn", $isbn);
    $stmt->bindValue(":title", $title);
    $stmt->bindValue(":author", $author);
    $stmt->execute();
}
```

What is Doctrine ORM?

- Doctrine ORM is JSR-317 / Hibernate inspired Object Relational Mapper (ORM) for PHP 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.
- https://en.wikipedia.org/wiki/Data_mapper_pattern
- http://hibernate.org/
- https://jcp.org/aboutJava/communityprocess/final/jsr317/index.html
- http://en.wikipedia.org/wiki/Object-relational_mapping

What is Doctrine ORM:

- Good for:
  - OLTP - Online transaction processing
  - DDD – domain driven dessign
  - FAST PROTOTYPING
  - OO-FIRST
- Not good for:
  - DYNAMIC DATA STRUCTURES → EAV
  - REPORTING

What are Entities?

- Entities should work without the ORM
- Entities should work without the DB
- Entities (mostly) represent your domain
- 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.
- The database is just saving things

What are Entities?

- Design entities first
- Define the database after modeling your domain
- Define mappings after designing the entities

Example Model – Bug Tracker

- 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 his reported or assigned Bugs.
- Bugs can be paginated through a list-view.

Obtaining an EntityManager

- Once you have prepared the class loading, you acquire an EntityManager instance.
- The EntityManager class is the primary access point to ORM functionality provided by Doctrine.

```
<?php
//bootstrap.php
require_once "vendor/autoload.php";

use Doctrine\ORM\Tools\Setup;
use Doctrine\ORM\EntityManager;

$isDevMode = false; // is true caching is done in memory with the ArrayCache. Proxy objects are recreated on every request.

$config = Setup::createAnnotationMetadataConfiguration(array(__DIR__."/src"), $isDevMode);

// the connection configuration
$dbParams = array(
    'driver' => 'pdo_mysql',
    'dbname' => 'doctrine',
    'user' => 'doctrine',
    'password' => 'doctrine123456',
    'host' => 'localhost',
);

// obtaining the entity manager
$entityManager = \Doctrine\ORM\EntityManager::create($dbParams, $config);
```

Obtaining an EntityManager

- Or if you prefer XML – usually used in public packages:

```
<?php
$paths = array(__DIR__."/config/xml");
$config = Setup::createXMLMetadataConfiguration($paths, $isDevMode);
$entityManager = EntityManager::create($dbParams, $config);
```

- Or if you prefer YAML:

```
<?php
$paths = array(__DIR__."/config/yaml");
$config = Setup::createXMLMetadataConfiguration($paths, $isDevMode);
$entityManager = EntityManager::create($dbParams, $config);
```

Start with the Product Entity

- 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;

```
<?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;
    }
}
```

Describe the structure of the Product entity

- 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.

```
<?php
// src/Product.php

/**
 * @Entity @Table(name="products")
 **/
class Product
{
    /** @Id @Column(type="integer") @GeneratedValue **/
    protected $id;

    /** @Column(type="string") **/
    protected $name;

    // .. (other code)
}
```

Describe the structure of the Product entity

- In external xml

```
<doctrine-mapping xmlns="http://doctrine-
project.org/schemas/orm/doctrine-mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
        xsi:schemaLocation="http://doctrine-
project.org/schemas/orm/doctrine-mapping
                http://doctrine-
project.org/schemas/orm/doctrine-mapping.xsd">

        <entity name="Product" table="products">
            <id name="id" type="integer">
                <generator strategy="AUTO" />
            </id>

            <field name="name" type="string" />
        </entity>
    </doctrine-mapping>
```

Describe the structure of the Product entity

- In YAML file
- YAML, YAML Ain't Markup Language, is a human friendly data serialization standard for all programming languages.
- YAML is a great format for your configuration files. YAML files are as expressive as XML files and as readable as INI files.

```
Product:
  type: entity
  table: products
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    name:
      type: string
```

Update database schema

- Doctrine ships with a number of command line tools that are very helpful during development. You can call this command from the Composer binary directory:

```
php vendor/bin/doctrine

php vendor/bin/doctrine orm:schema-tool:update –dump-sql

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

Create script for insert product to database

- 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.

```
<?php
// create_product.php
require_once "bootstrap.php";


$faker = Faker\Factory::create();

$productName = $faker->sentence(3);
$productName = substr($productName, 0,
strlen($productName) - 1);

$product = new Product();
$product->setName($productName);

$entityManager->persist($product);
$entityManager->flush();

echo "Created Product with ID " . $product->getId() . "\n";
```

List of all the Products in the database

- 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().
- find(id)
- findBy(array(criteria))
- findOneBy(array(criteria())

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

Update products name on given id

- 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();
```

Adding Bug and User Entities

- 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.

```
<?php
// src/Bug.php
/**
 * @Entity(repositoryClass="BugRepository")
@Table(name="bugs")
 */
class Bug
{
    /**
     * @Id @Column(type="integer") @GeneratedValue
     * @var int
     */
    protected $id;
    /**
     * @Column(type="string")
     * @var string
     */
    protected $description;
    /**
     * @Column(type="datetime")
     * @var DateTime
     */
    protected $created;
    /**
     * @Column(type="string")
     * @var string
     */
    protected $status;

    public function getId()
    {
        return $this->id;
```

Adding Bug and User Entities

- Never have to work with the foreign keys directly.
- 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.

```
<?php
// src/User.php
/**
 * @Entity @Table(name="users")
 */
class User
{
    /**
     * @Id @GeneratedValue @Column(type="integer")
     * @var int
     */
    protected $id;
    /**
     * @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;
    }
}
```

Adding Bug and User Entities – extend model

- A Bug can occur on different Products (platforms)

```
<?php
// src/Bug.php
use Doctrine\Common\Collections\ArrayCollection;

class Bug
{
    // ... (previous code)

    protected $products;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
}
```

Adding Bug and User Entities – extend model

- 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 his reported or assigned Bugs.

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

Object-Relational Mapping practices

- 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.

  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.

Object-Relational Mapping practices

- 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.

Bug and User Entities – bi-directional reference

- 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.**

```
<?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;
    }
}
```

Bug and User Entities – bi-directional reference

- 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.

```
<?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;
    }
}
```

Bug mappings

- Created - datetime type translates the YYYY-mm-dd HH:mm:ss database format into PHP DateTime instance
- Two 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.

```
<?php
// src/Bug.php
/**
 * @Entity @Table(name="bugs")
 **/
class Bug
{
    /**
     * @Id @Column(type="integer") @GeneratedValue
     **/
    protected $id;
    /**
     * @Column(type="string")
     **/
    protected $description;
    /**
     * @Column(type="datetime")
     **/
    protected $created;
    /**
     * @Column(type="string")
     **/
    protected $status;

    /**
     * @ManyToOne(targetEntity="User", inversedBy="assignedBugs")
     **/
    protected $engineer;

    /**
     * @ManyToOne(targetEntity="User", inversedBy="reportedBugs")
     **/
    protected $reporter;

    /**
     * @ManyToMany(targetEntity="Product")
     **/
    protected $products;

    // ... (other code)
}
```

User mappings

- Both reportedBugs and assignedBugs are inverse relations, which means the join details have already been defined on the owning side.
- We only have to specify the property on the Bug class that holds the owning sides.

```
<?php
// src/User.php
/**
 * @Entity @Table(name="users")
 **/
class User
{
    /**
     * @Id @GeneratedValue @Column(type="integer")
     * @var int
     **/
    protected $id;

    /**
     * @Column(type="string")
     * @var string
     **/
    protected $name;

    /**
     * @OneToMany(targetEntity="Bug", mappedBy="reporter")
     * @var Bug[] An ArrayCollection of Bug objects.
     **/
    protected $reportedBugs = null;

    /**
     * @OneToMany(targetEntity="Bug", mappedBy="engineer")
     * @var Bug[] An ArrayCollection of Bug objects.
     **/
    protected $assignedBugs = null;

    // .. (other code)
}
```

Usage – create user

- php create_user.php marko

```
<?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";
```

Usage – create bug

- php create_bug.php 1 1 1

```
<?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";
```

Use case – list of bugs

- 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.

```
<?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";
}
```

Array Hydration of the Bug List

- Hydration can be an expensive process so only retrieving what you need can yield considerable performance benefits for read-only requests.
- It does not know the concept of columns and tables, but only those of Entity-Class and property.
- 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.

```
<?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";
}
```

Entity Repositories

- Repository pattern - allows separating query logic from the model
- 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.
  - findOneBy()
  - findBy()

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

Entity Repositories

- To be able to use this query logic through: $this->getEntityManager()->getRepository('Bug') we have to adjust the metadata slightly.
- Using EntityRepositories you can avoid coupling your model with specific query logic. You can also re-use query logic easily throughout your application.

```php
<?php
/**
 * @Entity(repositoryClass="BugRepository")
 * @Table(name="bugs")
 **/
class Bug
{
    //...
}
```

```php
<?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";
}
```