Simplifying database interactions with Doctrine DBAL

August 15th, 2014

I previously wrote about switching from the mysql extension to PDO. PDO introduces a number of convenient features beyond the mysql extension such as transactions, prepared statements, and more fetching options. However there are still a few things that are a bit painful.

This article will introduce Doctrine DBAL to help alleviate some of these pain points.

is a wrapper around PDO. It adds a few conveniences beyond straight PDO as well as a query builder.

There are a number of projects under the Doctrine umbrella including a full . This article will only cover the DBAL project.

Installing Doctrine DBAL

The recommended way to install Doctrine DBAL is via composer.

Add to the composer.json require section

```
"doctrine/dbal": "2.3.4"
```

Alternatively you can download a zip archive from the project page.

If you aren't using composer or don't have a PSR-0 compatible autoloader you will need to add the class loader in the Doctrine/Common folder to load the Doctrine DBAL classes.

Setup Class Loader

```
use Doctrine\Common\ClassLoader;

require '/path/to/doctrine/lib/Doctrine/Common/ClassLoader.php';

$classLoader = new ClassLoader('Doctrine', '/path/to/doctrine');
$classLoader->register();
```

Connecting to The Database

Doctrine DBAL can connect to any type of database that PDO can connect to. For this article I am going to assume you are connecting to MySQL but the api is the same regardless of the database you are connecting to.

Connection Documentation

```
$config = new \Doctrine\DBAL\Configuration();

$connectionParams = array(
    'dbname' => 'database',
    'user' => 'user',
    'password' => 'password',
```

```
    'host' => 'localhost',
    'port' => 3306,
    'charset' => 'utf8',
    'driver' => 'pdo_mysql',
);
$dbh = \Doctrine\DBAL\DriverManager::getConnection($connectionParams, $config);
```

One advantage that Doctrine DBAL has over plain PDO is that the Doctrine connection doesn't actually connect to the database until the first query is run. This means you can create the connection in the bootstrap of your application and if no queries are run for a particular request it wont need to actually connect to the database server. PDO connects to the database server as soon as you create a PDO instance.

Running Queries

Being a wrapper around PDO you can use the full PDO api including prepared statements and transactions just like you would with straight PDO.

Fetching Data works exactly the same as with straight PDO.

Fetching Documentation

```
// Fetch one row
$sth = $dbh->query("SELECT * FROM users WHERE id = 1");
$user = $sth->fetch();

// Fetch all rows
$sth = $dbh->query("SELECT * FROM users");
$users = $sth->fetchAll();

// Fetch column as scalar value
$sth = $dbh->query("SELECT email FROM users WHERE id = 1");
$email = $sth->fetchColumn();

// Fetch column as array of scalar values
$sth = $dbh->query("SELECT email FROM users");
$emails = $sth->fetchAll(PDO::FETCH_COLUMN);

// Fetch column as key value pairs
$sth = $dbh->query("SELECT id, email FROM users");
$users = $sth->fetchAll(PDO::FETCH_KEY_PAIR);
```

Prepared statements are also the same as straight PDO.

Prepared Statements Documentation

```
$sth = $dbh->prepare("SELECT * FROM users WHERE id = ?");
$sth->bindValue(1, $id, PDO::PARAM_INT);
$sth->execute();
$user = $sth->fetchAssoc();

$sth = $dbh->prepare("UPDATE users SET name = :name, email = :email WHERE id
= :id");
$sth->bindValue(":name", $name);
$sth->bindValue(":email", $email);
$sth->bindValue(":id", $id, PDO::PARAM_INT);
$sth->execute();
```

With Doctrine DBAL you can combine preparing and executing into one step.

Preparing and Executing in one command

```
$sth = $conn->executeQuery('SELECT * FROM users WHERE email = ?',
array('email@excample.com'));
$user = $sth->fetch();

// executeUpdate will return the number of affected rows
$count = $dbh->executeUpdate("UPDATE users SET name = ?, email = ? WHERE id
= ?", array($name, $email, $id));
// Same with named parameters
$count = $dbh->executeUpdate("UPDATE users SET name = :name, email = :email
WHERE id = :id", array('name' => $name, 'email' => $email, 'id' => $id));

// You can even prepare, execute, and fetch in one step
$users = $dbh->fetchAll("SELECT * FROM users WHERE name LIKE ?", array($name .
'%'));
```

Inserts, Updates, and Deletes

With Doctrine DBAL you don't even need to write sql for inserts, updates, and deletes.

Inserts Documentation

```
// Insert a new user into the users table
$dbh->insert('users', array('name' => 'Bob', 'email' => 'bob@example.com'));
// This is the same as running the following query
// INSERT INTO users (name, email) VALUES ('Bob', 'bob@example.com')
```

Updates Documentation

```
$dbh->update('users', array('name' => 'Bob'), array('id' => 1));
// This is the same as running the following query
// UPDATE users SET name = 'Bob' WHERE id = 1
```

Deletes Documentation

```
$dbh->delete('users', array('id' => 1));
// This is the same as running the following query
// DELETE FROM users WHERE id = 1
```

Query Builder

Doctrine DBAL also features a query builder to help build complicated SQL queries.

```
// This will run the following query
/*
SELECT u.id, u.name, u.email, p.bio
FROM users u
INNER JOIN user_profile p ON (u.id = p.user_id)
WHERE u.id = 1
ORDER BY u.name ASC
*/
$id = 1;
$query = $dbh->createQueryBuilder();
$query->select('u.id', 'u.name', 'u.email', 'p.bio')
      ->from('users', 'u')
      ->innerJoin('u', 'user_profile', 'p', 'u.id = p.user_id')
      ->orderBy('u.name', 'ASC')
      ->where('u.id = :id')
      ->setParameter(':id', $id)
```

```
;
$sth = $query->execute();
$user = $sth->fetchAssoc();
```

Where I find the query builder to be the most helpful is when you have parts of the query that are dependent on user provided data.

For example imagine a page with a search form with multiple fields that filter results by different columns. You might have a text field to search the names of users and another text field to search by email address. There is also a select box to search by account type with the values Any, Editors, and Members. If Any is selected you do not want to filter by account type but if Editors or Members is selected you do. With the other fields you only want to filter them if the user entered anything in the fields. You always only want to return active accounts.

```
// User provided search data
$search = array(
  'name' => 'Bob',
  'email' => null,
  'type' => 'any'
);

// Query That should be run
// SELECT * FROM users WHERE name LIKE '%Bob%'

// User provided search data
$search = array(
  'name' => 'Bob',
  'email' => 'email@example.com',
  'type' => 'member'
);

// Query That should be run
// SELECT * FROM users WHERE active = 1 AND name LIKE '%Bob%' AND email LIKE
'%email@example.com%' AND type = 'member' ORDER BY name ASC
```

With straight SQL the only way to build this query is with string concatenation.

Building query with string concatenation and PDO

```
$where = array("active = 1");
if ($search['name']) {
    $where[] = "name LIKE :name";
}
if ($search['email']) {
    $where[] = "email LIKE :email";
}
if ($search['type'] != 'any') {
    $where[] = "type = :type";
}
if ($where) {
    $where = implode(" AND ", $where);
} else {
    $where = '';
}

$sql = "SELECT * FROM users WHERE " . $where . " ORDER BY name ASC";
$sth = $dbh->prepare($sql);
if ($search['name']) {
    $sth->bindValue(':name', "%" . $search['name'] . "%");
}
if ($search['email']) {
```

```
    $sth->bindValue(':email', "%" . $search['email'] . "%");
}
if ($search['type'] != 'any') {
    $sth->bindValue(':type', $search['type']);
}
$sth->execute();
$users = $sth->fetchAll(PDO::FETCH_ASSOC);
```

This example is actually pretty simple and the code is already difficult to understand.

Lets look at the same example using the query builder

Using the query builder

```
$query = $dbh->createQueryBuilder();
$query->select('*');
$query->from('users');
$query->where("active = 1");
if ($search['name']) {
    $query->andWhere('name LIKE :name');
    $query->setParameter(':name', "%" . $search['name'] . "%");
}
if ($search['email']) {
    $query->andWhere('email LIKE :email');
    $query->setParameter(':email', "%" . $search['email'] . "%");
}
if ($search['type'] != 'any') {
    $query->andWhere('type = :type');
    $query->setParameter(':type', $search['type']);
}
$query->orderBy('name', 'ASC');
$sth = $query->execute();
$users = $sth->fetchAll(PDO::FETCH_ASSOC);
```

Not only is that code shorter but if you read it out loud it even sounds more like English. This greatly helps at understanding what the code does at a glance 6 months later.

The query builder does result in some extra overhead as it needs to compile the query so raw SQL is generally a better idea if the query is simple.

Conclusion

Doctrine DBAL adds a lot of functionality for a very lightweight overhead. Even if you don't use the query builder it is worth it for the shorter syntax on inserts, updates, and deletes as well as the ability to prepare, execute, and fetch data in one statement.

There is a lot more functionality I didn't cover so I recommend taking a look at the documentation for other pieces of functionality such as events, a schema manager, caching, sharding, and more.

https://www.thedevfiles.com/2014/08/simplifying-database-interactions-with-doctrine-dbal

Create a Doctrine DBAL Connection for Another Database

There are 3 ways to create a DBAL connection to access another database:

\Doctrine\Bundle\DoctrineBundle\ConnectionFactory

```
Copy
/** @var \Doctrine\Bundle\DoctrineBundle\ConnectionFactory $connectionFactory */
$connectionFactory = $this->getContainer()-
>get('doctrine.dbal.connection_factory');
$connection = $connectionFactory->createConnection([
        'pdo' => new \PDO(
                "mysql:host=$hostname;dbname=$dbname",
                $username,
                $password)
]);

$connection->executeQuery('SELECT * FROM your_table');
```

\Doctrine\DBAL\DriverManager

```
Copy
/**
 * Creates connection based on application configuration.
 */
function createConnection(): \Doctrine\DBAL\Connection
{
    $config = new \Doctrine\DBAL\Configuration();
    $parser = new \Symfony\Component\Yaml\Parser();
    $configParams = $parser->parse(file_get_contents(getcwd() .
'/config/parameters.yml'));
    $configParams = $configParams['parameters'];

    return \Doctrine\DBAL\DriverManager::getConnection([
        'dbname' => $configParams['database_name'],
        'user' => $configParams['database_user'],
        'password' => $configParams['database_password'],
        'host' => $configParams['database_host'],
        'driver' => $configParams['database_driver']
    ], $config);
}
```

```
Copy
public function exSQL()
{


    $config = new \Doctrine\DBAL\Configuration();

    $connectionParams = array(
        'dbname' => 'dbname',
        'user' => 'user',
        'password' => 'password',
        'host' => 'prod_host',
        'driver' => 'pdo_mysql',
    );
```

```
    $conn = \Doctrine\DBAL\DriverManager::getConnection($connectionParams,
$config);


    $sql = "SELECT ...";
    $stmt = $conn->query($sql);

    while ($row = $stmt->fetch()) {
        var_dump($row);
    }

    // This works !!!


}
```

\Doctrine\ORM\EntityManager

src\Stars\Bundle\LoggerEventBundle\Resources\config\oro\app.yml

```
Copy
doctrine:
    dbal:
        connections:
            emgsihe:
                driver:   pdo_mysql
                host:     '%database_host%'
                port:     '%database_port%'
                dbname:   emgsihe
                user:     '%database_user%'
                password: '%database_password%'
                charset:  UTF8
    orm:
        entity_managers:
            emgsihe:
                connection: emgsihe

Copy
// ...

use \Doctrine\ORM\EntityManager;
//...

public function __construct(EntityManager $entityManager)
{
    $this->entityManager = $entityManager;
}

public function exSQL($conn_name)
{

    $conn = $this->entityManager->getConnection($conn_name);

    $sql = "SELECT ...";
    $stmt = $conn->query($sql);

    while ($row = $stmt->fetch()) {
        var_dump($row);
    }

}
```

https://makandracards.com/oro/67274-create-a-doctrine-dbal-connection-for-another-database

Doctrine (DBAL) Cheatsheet

Snipps about how to use Doctrine DBAL core functions.

Simple Connection

```
<?php
$config = new \Doctrine\DBAL\Configuration();
//..
$connectionParams = array(
    'dbname' => 'mydb',
    'user' => 'user',
    'password' => 'secret',
    'host' => 'localhost',
    'driver' => 'pdo_mysql',
);
$conn = \Doctrine\DBAL\DriverManager::getConnection($connectionParams, $config);
```

Simple Queries and Dynamic Parameters

```
<?php
// $conn instanceof Doctrine\DBAL\Connection
$sql = "SELECT * FROM articles WHERE id = ?";
$stmt = $conn->prepare($sql);
$stmt->bindValue(1, $id);
$stmt->execute();

// Named parameters
$sql = "SELECT * FROM users WHERE name = :name OR username = :name";
$stmt = $conn->prepare($sql);
$stmt->bindValue("name", $name);
$stmt->execute();
```

Binding Types

```
<?php
/*
Doctrine DBAL extends PDOs handling of binding types in prepared statement
considerably. Besides the well known \PDO::PARAM_* constants you can make use of
two very powerful additional features.
*/
$date = new \DateTime("2011-03-05 14:00:21");
$stmt = $conn->prepare("SELECT * FROM articles WHERE publish_date > ?");
$stmt->bindValue(1, $date, "datetime");
$stmt->execute();
```

Prepare

```
<?php
$statement = $conn->prepare('SELECT * FROM user');
$statement->execute();
$users = $statement->fetchAll();
```

```
/*
array(
  0 => array(
    'username' => 'jwage',
    'password' => 'changeme'
  )
)
*/
```

Execute Update

```
<?php
// $sql, $params, $types
$count = $conn->executeUpdate('UPDATE user SET username = ? WHERE id = ?',
array('jwage', 1));
echo $count; // 1
```

Execute Query

```
<?php
// $sql, $params, $types
$statement = $conn->executeQuery('SELECT * FROM user WHERE username = ?',
array('jwage'));
$user = $statement->fetch();

/*
array(
  0 => 'jwage',
  1 => 'changeme'
)
*/
```

Fetch All

```
<?php
$users = $conn->fetchAll('SELECT * FROM user');

/*
array(
  0 => array(
    'username' => 'jwage',
    'password' => 'changeme'
  )
)
*/
```

Fetch Array

```
<?php
$user = $conn->fetchArray('SELECT * FROM user WHERE username = ?',
array('jwage'));

/*
array(
  0 => 'jwage',
  1 => 'changeme'
)
*/
```

Fetch Column

```
<?php
$username = $conn->fetchColumn('SELECT username FROM user WHERE id = ?',
array(1), 0);
echo $username; // jwage
```

Fetch Assoc

```
<?php
$user = $conn->fetchAssoc('SELECT * FROM user WHERE username = ?',
array('jwage'));
/*
array(
  'username' => 'jwage',
  'password' => 'changeme'
)
*/
```

Delete

```
<?php
$conn->delete('user', array('id' => 1));
// DELETE FROM user WHERE id = ? (1)
```

Insert

```
<?php
$conn->insert('user', array('username' => 'jwage'));
// INSERT INTO user (username) VALUES (?) (jwage)
```

Update

```
<?php
$conn->update('user', array('username' => 'jwage'), array('id' => 1));
// UPDATE user (username) VALUES (?) WHERE id = ? (jwage, 1)
```

Quote

```
<?php
$quoted = $conn->quote('value');
$quoted = $conn->quote('1234', \PDO::PARAM_INT);
```

Quote Identifier

```
<?php
$quoted = $conn->quoteIdentifier('id');
```

Query Builder

```
<?php
// $conn instanceof Doctrine\DBAL\Connection

// Query
$query = $conn->createQueryBuilder()
              ->select('*')
              ->from('users')
              ->orderBy('id', 'DESC')
```

```
                ->where('id = :id')
                ->setParameter('id', 1))
                ->setMaxResults(10)
                ->setFirstResult(1);
// Results
$rows = $query->execute()->fetchAll();
```

Query Builder w/count

```
<?php
// $conn instanceof Doctrine\DBAL\Connection

$query = $conn->createQueryBuilder()
                ->select('COUNT(id) as count')
                ->from($table)
                ->where('id = :id')
                ->setParameter('id', 1);
$rowcount = $query->execute()->fetch();
```

Query Builder w/multiple joins

```
<?php
// $conn instanceof Doctrine\DBAL\Connection

$query = $conn->createQueryBuilder()
                ->select('U.*')
                ->from('users', 'U')
                ->leftJoin('U', 'genders', 'G', 'U.gender_id = G.id')
                ->leftJoin('U', 'houses', 'H', 'U.house_id = H.id')
                ->innerJoin('H', 'kindoms', 'K', 'H.kindom_id = K.id')
                ->where('U.id = ?')
                ->setParameter(0, 1);
```

Doctrine DBAL and the LIKE operator

Development • Nov 18, 2019

Doctrine is a great abstraction layer, and I've resolved to use it more and more through my projects to keep things simple and maintainable. The problem is, the documentation isn't always great for it. Take the below image as an example of the documentation for the building a "LIKE" clause for a query:

```php
<?php
// $qb instanceof QueryBuilder

$qb->select(array('u')) // string 'u' is converted to array internally
    ->from('User', 'u')
    ->where($qb->expr()->orX(
        $qb->expr()->eq('u.id', '?1'),
        $qb->expr()->like('u.nickname', '?2')
    ))
    ->orderBy('u.surname', 'ASC');
```

The above makes it look like binding parameter 2 would automatically set the relevant wildcards round the parameter, making things simple. If you try to do it this way, the results don't come up as expected. Looking at the ExpressionBuilder code outlines why this doesn't work.

The 'like' comparison builder code does the following:

```
/**
 * Creates a LIKE() comparison expression with the given arguments.
 *
 * @param string $x Field in string format to be inspected by LIKE() comparison.
 * @param mixed  $y Argument to be used in LIKE() comparison.
 */
public function like(string $x, $y/*, ?string $escapeChar = null */) : string
{
        return $this->comparison($x, 'LIKE', $y) .
                (func_num_args() >= 3 ? sprintf(' ESCAPE %s', func_get_arg(2)) :
'');
}
```

The code is there from either a past escape character, or for future use, but this simply hands off the comparison building to the comparison() function of the class.

```
/**
 * Creates a comparison expression.
 *
 * @param mixed  $x        The left expression.
 * @param string $operator One of the ExpressionBuilder::* constants.
 * @param mixed  $y        The right expression.
 */
public function comparison($x, string $operator, $y) : string
{
        return $x . ' ' . $operator . ' ' . $y;
}
```

As shown above, this is a simple string concatenation, meaning nothing clever actually takes place as part of the LIKE comparison builder. Therefore, to get the LIKE to actually work, you need to bind the wildcards as part of the parameter substitution:

```
$result = $queryBuilder->select('*')
        ->from($this->getTable())
        ->where(
                $queryBuilder->expr()->like('name', ':name')
        )
        ->orderBy('name', 'asc')
        ->setParameter(':name', '%' . $name . '%')
        ->execute();
```

Arguably this provides a more flexible solution rather than needing to create methods for `startsWith()` which would do the following for the code above:

```
$result = $queryBuilder->select('*')
        ->from($this->getTable())
        ->where(
                $queryBuilder->expr()->like('name', ':name')
        )
        ->orderBy('name', 'asc')
        ->setParameter(':name', $name . '%')
        ->execute();
```

And then also needing an `endsWith()` function for creating the following:

```
$result = $queryBuilder->select('*')
        ->from($this->getTable())
        ->where(
                $queryBuilder->expr()->like('name', ':name')
        )
        ->orderBy('name', 'asc')
        ->setParameter(':name', '%' . $name)
        ->execute();
```

If there were functions for `startsWith()` and `endsWith()`, then there would logically need to be the inverse functions of `notStartsWith()` and `notEndsWith()`. These are essentially duplicates of the `like()` and `notLike()` functions, but with very specific customisation. They may increase the readability of the code, but don't offer much more than the existing functions.

The take away lesson from this is that you need to add the wildcard characters around the variable you are substituting to get the result you desire.

https://www.garybell.co.uk/doctrine-dbal-and-the-like-operator/

Using Doctrine with Slim

This cookbook entry describes how to integrate from scratch the widely used Doctrine ORM into a Slim application.

Adding Doctrine to your application

The first step is importing the library into the `vendor` directory of your project using composer.

```
composer require doctrine/orm
```

Provide database credentials

Next, add the Doctrine settings alongside your Slim configuration.

```
<?php

// settings.php

define('APP_ROOT', __DIR__);

return [
    'settings' => [
        'displayErrorDetails' => true,
        'determineRouteBeforeAppMiddleware' => false,

        'doctrine' => [
            // if true, metadata caching is forcefully disabled
            'dev_mode' => true,

            // path where the compiled metadata info will be cached
            // make sure the path exists and it is writable
            'cache_dir' => APP_ROOT . '/var/doctrine',

            // you should add any other path containing annotated entity classes
            'metadata_dirs' => [APP_ROOT . '/src/Domain'],

            'connection' => [
                'driver' => 'pdo_mysql',
                'host' => 'localhost',
                'port' => 3306,
                'dbname' => 'mydb',
                'user' => 'user',
                'password' => 'secret',
                'charset' => 'utf-8'
            ]
        ]
    ]
];
```

Define the EntityManager service

Now we define the `EntityManager` service, which is the primary way to interact with Doctrine. Here we show how to configure the metadata reader to work with PHP annotations, which is at the

same time the most used mode and the most tricky to set up. Alternatively, XML or YAML can also be used to describe the database schema.

```
<?php

// bootstrap.php

use Doctrine\Common\Annotations\AnnotationReader;
use Doctrine\Common\Cache\FilesystemCache;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Mapping\Driver\AnnotationDriver;
use Doctrine\ORM\Tools\Setup;
use Slim\Container;

require_once __DIR__ . '/vendor/autoload.php';

$container = new Container(require __DIR__ . '/settings.php');

$container[EntityManager::class] = function (Container $container):
EntityManager {
    $config = Setup::createAnnotationMetadataConfiguration(
        $container['settings']['doctrine']['metadata_dirs'],
        $container['settings']['doctrine']['dev_mode']
    );

    $config->setMetadataDriverImpl(
        new AnnotationDriver(
            new AnnotationReader,
            $container['settings']['doctrine']['metadata_dirs']
        )
    );

    $config->setMetadataCacheImpl(
        new FilesystemCache(
            $container['settings']['doctrine']['cache_dir']
        )
    );

    return EntityManager::create(
        $container['settings']['doctrine']['connection'],
        $config
    );
};

return $container;
```

Create the Doctrine console

To run database migrations, validate class annotations and so on you will use the `doctrine` CLI application that is already present at `vendor/bin`. But in order to work, this script needs a `cli-config.php` file at the root of the project telling it how to find the `EntityManager` we just set up:

```
<?php

// cli-config.php

use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Tools\Console\ConsoleRunner;
use Slim\Container;
```

```
/** @var Container $container */
$container = require_once __DIR__ . '/bootstrap.php';

return ConsoleRunner::createHelperSet($container[EntityManager::class]);
```

Take a moment to verify that the console app works. When properly configured, its output will look more or less like this:

```
$ php vendor/bin/doctrine
Doctrine Command Line Interface 2.5.12

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output,
2 for more verbose output and 3 for debug

Available commands:
  help                              Displays help for a command
  list                              Lists commands
 dbal
  dbal:import                       Import SQL file(s) directly to Database.
  dbal:run-sql                      Executes arbitrary SQL directly from the
command line.
 orm
  orm:clear-cache:metadata          Clear all metadata cache of the various cache
drivers.
  orm:clear-cache:query             Clear all query cache of the various cache
drivers.
  orm:clear-cache:result            Clear all result cache of the various cache
drivers.
  orm:convert-d1-schema             [orm:convert:d1-schema] Converts Doctrine 1.X
schema into a Doctrine 2.X schema.
  orm:convert-mapping               [orm:convert:mapping] Convert mapping
information between supported formats.
  orm:ensure-production-settings    Verify that Doctrine is properly configured
for a production environment.
  orm:generate-entities             [orm:generate:entities] Generate entity
classes and method stubs from your mapping information.
  orm:generate-proxies              [orm:generate:proxies] Generates proxy classes
for entity classes.
  orm:generate-repositories         [orm:generate:repositories] Generate
repository classes from your mapping information.
  orm:info                          Show basic information about all mapped
entities
  orm:mapping:describe              Display information about mapped objects
  orm:run-dql                       Executes arbitrary DQL directly from the
command line.
  orm:schema-tool:create            Processes the schema and either create it
directly on EntityManager Storage Connection or generate the SQL output.
  orm:schema-tool:drop              Drop the complete database schema of
EntityManager Storage Connection or generate the corresponding SQL output.
  orm:schema-tool:update            Executes (or dumps) the SQL needed to update
the database schema to match the current mapping metadata.
  orm:validate-schema               Validate the mapping files.
```

If it works, you can now create your database and load the schema by running `php vendor/bin/doctrine orm:schema-tool:update`

Using the EntityManager in our own code

Congratulations! Now you can already manage your database from the command line, and use the `EntityManager` wherever you need it.

```
$container[UserRepository::class] = function ($container) {
    return new UserRepository($container[EntityManager::class]);
};

// src/UserRepository.php

class UserRepository
{
    /**
     * @var EntityManager
     */
    private $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }

    public function signUp(string $email, string $password): User
    {
        $user = new User($email, $password);

        $this->em->persist($user);
        $this->em->flush();

        return $user;
    }
}
```

Other resources

- The official Doctrine ORM documentation.
- A full example of the above configuration in a small, functioning project.

https://www.slimframework.com/docs/v3/cookbook/database-doctrine.html

Substituindo o PDO por Doctrine DBAL

Por Luiz Carlos em

O *PDO* é uma biblioteca PHP de abstração de dados para trabalhar com banco de dados, desde que ela foi lançada no PHP 5, mudou-se a forma de como manipular a comunicação com banco de dados, trazendo inúmeras melhorias. A primeira claramente, é deixando a manipulação procedural de lado para focar na orientação a objetos e também as melhorias quanto a facilidade de manipulação de dados. Mas ainda assim, trabalhar com SQL puro pode se tornar maçante e/ou chato, porque, às vezes os comandos são grandes e/ou complexos, dando margem para erros que atrasam nosso desenvolvimento e manutenção das aplicações. O *PDO* é fantástico para manipular banco de dados, porém, para um banco de dados mais complexo, ele pode não ser adequado, por isto, mostro uma alternativa para trabalhar no topo dele.

Qual é a ideia?

Usaremos o *Doctrine DBAL*, que é um componente do framework *Doctrine*, este componente é uma abstração de dados acima do PDO, ou seja, abstrai mais ainda a manipulação do banco de dados. No final das contas, por trás, ele usa o *PDO* mesmo, mas, traz uma abstração absurda quanto a manipulação das *queries*.

O que faremos?

1. Instalar o *Composer*
2. Criar uma aplicação com o *DBAL*
3. Mostrar o uso do *DBAL*.
4. Considerações finais.

Passos

1. **Instalar o *Composer*.**

Se você não conhece, o Composer é um gerenciador de dependências de bibliotecas PHP, semelhante ao *RubyGems* do Ruby. Ele faz a instalação de bibliotecas pré-registradas e suas dependências, além, de fazer um carregamento automático de tudo que precisamos para executar nossa aplicação.

Existem muitas maneiras de instala-lo, confira aqui. Para ganharmos tempo irei instalar usando PHP puro. Crie uma pasta para nosso novo projeto e execute este comando na raiz:

```
php -r "readfile('https://getcomposer.org/installer');" | php
```

Isto baixará o arquivo *composer.phar* que nós permitirá usar o *Composer*.

2. **Criar uma aplicação com o *DBAL*.**

Agora vamos criar nossa aplicação com o *Composer* e instalar o *DBAL*, faça:

```
php composer.phar require doctrine/dbal:~2.3
```

Isto criará nossa aplicação e instalará como dependência o *DBAL* versão >= 2.3 e <3.0 e suas dependências.

Vamos criar um arquivo *index.php* na raiz do projeto, iniciar o *DBAL* e passar as configurações de banco de dados:

```
require_once __DIR__ . '/vendor/autoload.php';

$config = new \Doctrine\DBAL\Configuration();

$connectionParams = array(
    'dbname' => 'code_dbal',
    'user' => 'root',
    'password' => 'root',
    'host' => 'localhost',
    'port' => 3306,
    'charset' => 'utf8',
    'driver' => 'pdo_mysql',
);
$dbh = \Doctrine\DBAL\DriverManager::getConnection($connectionParams, $config);
```

Na primeira linha iniciamos o carregamento do que precisamos na aplicação, então, posteriormente não precisaremos ficar importando nada com *includes* e *requires*. Mais abaixo configuramos a conexão do *DBAL* com banco de dados.

3. **Mostrar uso do *DBAL***

Para ganharmos tempo, criei um pequeno script no mysql para podermos fazer transações no banco de dados. No final do post você poderá ver todo código criado e o script do banco. O banco tem duas tabelas: categorias e produtos (todo produto terá uma categoria).

A priori criar as *queries* com *DBAL* é bem semelhante ao *PDO*:

```
//Pegar uma categoria
$statement = $dbh->query("SELECT * FROM categorias WHERE id = 1");
$categoria = $statement->fetch();
var_dump($categoria);

//Pegar todas categorias
$statement = $dbh->query("SELECT * FROM categorias");
$categorias = $statement->fetchAll();
var_dump($categorias);

//Pegar um campo de um produto
$statement = $dbh->query("SELECT nome FROM produtos WHERE id = 1");
$nome = $statement->fetchColumn();
echo $nome;

//Pegar coleção de nomes em array
$statement = $dbh->query("SELECT nome FROM produtos");
$nomes = $statement->fetchAll(\PDO::FETCH_COLUMN);
var_dump($nomes);

//Pegar coleção de categorias em array key => value
$statement = $dbh->query("SELECT id, nome FROM categorias");
$categorias = $statement->fetchAll(\PDO::FETCH_KEY_PAIR);
var_dump($categorias);
```

Mas, a primeira vantagem do *DBAL* sobre o *PDO*, é que o *DBAL* quando instanciado não realiza conexão com banco de dados até que uma primeira *query* seja executada, já o *PDO* não.

Outra vantagem é o ganho para se criar *INSERTS*, *UPDATES* E *DELETES*, veja como a sintaxe é mais simples:

```
//Para incluir uma categoria
$dbh->insert('categorias', array('nome' => 'Minha categoria'));

//Para incluir uma categoria
$dbh->update('categorias', array('nome' => 'Minha categoria atualizada'),
array('id' => 1));

//Para excluir um produto
$dbh->delete('produtos', array('id' => 1));
```

Agora vou mostrar a minha funcionalidade preferida, que é o *QueryBuilder*. Ele alivia e muito o trabalho de criar *querys* complexas, com muitas regras e *joins*. Vamos criar algumas *querys*:

```
//Pegar produtos que tem a categoria 2
$query = $dbh->createQueryBuilder();
$query->select('p.*')
    ->from('produtos', 'p')
    ->where('p.categoria_id = :categoria')
    ->setParameter(':categoria', 2);
;
$statement = $query->execute();
$produtos = $statement->fetch();
var_dump($produtos);

//Pegar todos produtos e suas respectivas categorias
$query = $dbh->createQueryBuilder();
$query->select('p.nome as produto,c.nome as categoria')
    ->from('produtos', 'p')
    ->leftJoin('p','categorias','c','c.id = p.categoria_id');
;
$statement = $query->execute();
$produtos = $statement->fetch();
print_r($produtos);
```

O *QueryBuilder* facilita nosso trabalho, porque abstrai a escrita de *SQL* puro, não temos que escrever uma *query* inteira em string, podemos trabalhar de forma orientada a objetos. Ele já tem todas funções que precisamos: *where*, *order by*, *joins*, *select*, *group by*, *froms*, etc.

4. **Considerações finais**

É isso aí pessoal, espero que tenham gostado e que comecem a usar o DBAL para facilitar a vida com o trabalho com banco de dados. Baixem o código deste tutorial, criem novas tabelas, novas *queries* e confirmem esta ferramenta fantástica que é o *Doctrine DBAL*.

Referências:

http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html

http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/data-retrieval-and-manipulation.html

http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/query-builder.html

https://blog.schoolofnet.com/substituindo-o-pdo-por-doctrine-dbal/

Yet Another Database Abstraction Layer with PHP and DBAL


by
Gonzalo Ayuso
.
Apr. 11, 14 · Java Zone · Interview
Like (0)
Save
Tweet
4.01K Views

I'm not a big fan of ORMs. I feel very confortable working with raw SQLs and because of that I normally use DBAL (or PDO in old projects). I've got one small library to handle my dayly operations with databases and today I've written this library

First of all imagine one DBAL connection. I'm using a sqlite in-memomy database in this example but we can use any database supported by DBAL (aka "almost all"):

```
use Doctrine\DBAL\DriverManager;

$conn = DriverManager::getConnection([
    'driver' => 'pdo_sqlite',
    'memory' => true
]);
```

We can also create one DBAL connection from a PDO connection. It's usefull to use DBAL within legacy applications instead of creating a new connection (remember that DBAL works over PDO)

```
use Doctrine\DBAL\DriverManager;

$conn = DriverManager::getConnection(['pdo' => new PDO('sqlite::memory:')]);
```

Now we set up the database for the example

```
$conn->exec("CREATE TABLE users (
            userid VARCHAR PRIMARY KEY  NOT NULL ,
            password VARCHAR NOT NULL ,
```

```
            name VARCHAR,
            surname VARCHAR
            );");
$conn->exec("INSERT INTO users VALUES('user','pass','Name','Surname');");
$conn->exec("INSERT INTO users VALUES('user2','pass2','Name2','Surname2');");
```

Our table “users” has two records. Now we can start to use our library.

First we create a new instance of our library:

```
use G\Db;

$db = new Db($conn);
```

Now a simple query from a string:

```
$data = $db->select("select * from users");
```

Sometimes I’m lazy and I don’t want to write the whole SQL string and I want to perform a select * from table:

```
use G\Sql;
$data = $db->select(SQL::createFromTable("users"));
```

Probably we need to filter our Select statement with a WHERE clause:

```
$data = $db->select(SQL::createFromTable("users", ['userid' => 'user2']));
```

And now something very intersting (at least for me). I want to iterate over the recordset and maybe change it. Of course I can use “foreach” over $data and do whatever I need, but I preffer to use the following sintax:

```
$data = $db->select(SQL::createFromTable("users"), function (&$row) {
    $row['name'] = strtoupper($row['name']);
});
```

For me it’s more readable. I iterate over the recordset and change the row ‘name’ to uppercase. Here you can see what is doing my “select” function:

```
/**
* @param Sql|string $sql
* @param \Closure $callback
* @return array
*/
public function select($sql, \Closure $callback = null)
{
    if ($sql instanceof Sql) {
        $sqlString = $sql->getString();
        $parameters = $sql->getParameters();
        $types = $sql->getTypes();
    } else {
        $sqlString = $sql;
        $parameters = [];
        $types = [];
    }

    $statement = $this->conn->executeQuery($sqlString, $parameters, $types);
    $data = $statement->fetchAll();
    if (!is_null($callback) && count($data) > 0) {
        $out = [];
        foreach ($data as $row) {
```

```
            if (call_user_func_array($callback, [&$row]) !== false) {
                $out[] = $row;
            }
        }
        $data = $out;
    }

    return $data;
}
```

And finally transactions (I normally never use autocommit and I like to handle transactions by my own)

```
$db->transactional(function (Db $db) {
    $userId = 'temporal';

    $db->insert('users', [
        'USERID'   => $userId,
        'PASSWORD' => uniqid(),
        'NAME'     => 'name3',
        'SURNAME'  => 'name3'
    ]);

    $db->update('users', ['NAME' => 'updatedName'], ['USERID' => $userId]);
    $db->delete('users', ['USERID' => $userId]);
});
```

The “transactional” function it’s very simmilar than DBAL’s transactional function

```
public function transactional(\Closure $callback)
{
    $out = null;
    $this->conn->beginTransaction();
    try {
        $out = $callback($this);
        $this->conn->commit();
    } catch (\Exception $e) {
        $this->conn->rollback();
        throw $e;
    }

    return $out;
}
```

I change a little bit because I like to return a value within the closure and allow to do things like that:

```
$status = $db->transactional(function (Db $db) {
    $userId = 'temporal';

    $db->insert('users', [
        'USERID'   => $userId,
        'PASSWORD' => uniqid(),
        'NAME'     => 'name3',
        'SURNAME'  => 'name3'
    ]);

    $db->update('users', ['NAME' => 'updatedName'], ['USERID' => $userId]);
    $db->delete('users', ['USERID' => $userId]);

    return "OK"
});
```

The other functions (insert, update, delete) only bypass the calls to DBAL's funcitons:

```
private $conn;

public function __construct(Doctrine\DBAL\Connection $conn)
{
    $this->conn = $conn;
}

public function insert($tableName, array $values = [], array $types = [])
{
    $this->conn->insert($tableName, $values, $types);
}

public function delete($tableName, array $where = [], array $types = [])
{
    $this->conn->delete($tableName, $where, $types);
}

public function update($tableName, array $data, array $where = [], array $types
= [])
{
    $this->conn->update($tableName, $data, $where, $types);
}
```

And that's all. You can use the library with composer and download at github.

BTW I've test the new Sensiolabs product (SensioLabs Insight) to analyze the code and verify good practices and I've got the Platinum medal #yeah!

https://dzone.com/articles/yet-another-database