DoctrineORMModule Documentation

Release latest

Jan 16, 2019

Table of Contents

This module works with Zend Framework 2 and 3.

CHAPTER 1

Zend Developer Tools in DoctrineORMModule

If you ever tried Zend Developer Tools you will surely understand the importance of being able to track performance pitfalls or excessive amount of queries in your applications when developing.

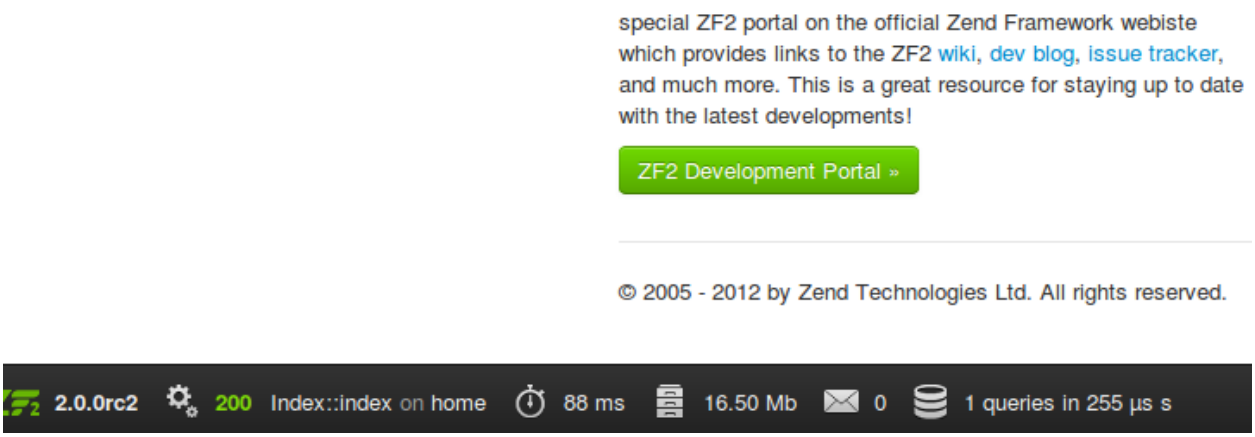
1.1 Setup

To setup Zend Developer Tools, run

```
composer require zendframework/zend-developer-tools
```

Then enable `ZendDeveloperTools` in your modules and enable profiling and the toolbar (see docs of Zend Developer Tools for that).

Once `ZendDeveloperTools` is enabled, having `doctrine.entity_manager.orm_default` as your default `EntityManager`, you will notice that the queries performed by the ORM get logged and displayed in the toolbar.



1.2 Customization

If you want to customize this behavior (or track multiple `EntityManager` instances) you can do it in different ways. Please note that if you have set an `SQLLogger` in your configuration, this functionality won't override it, so you can use these features in total safety.

1.2.1 Multiple EntityManager/Connection instances and logging

WARNING! These are advanced features! Even if the code is fully tested, this is usually not required for most users!

To setup logging for an additional DBAL Connection or EntityManager, put something like following in your module:

```
<?php

namespace MyNamespace;

class Module
{
    public function getConfig()
    {
        return [
            'doctrine' => [
                'sql_logger_collector' => [
                    'other_orm' => [
                        // name of the sql logger collector (used by␣
↪ZendDeveloperTools)
                        'name' => 'other_orm',

                        // name of the configuration service at which to attach the␣
↪logger
                        'configuration' => 'doctrine.configuration.other_orm',

                        // uncomment following if you want to use a particular SQL␣
↪logger instead of relying on
                        // the attached one
                        //'sql_logger' => 'service_name_of_my_dbal_sql_logger',
                    ],
                ],
            ],

            'zenddevelopertools' => [

                // registering the profiler with ZendDeveloperTools
                'profiler' => [
                    'collectors' => [
                        // reference to the service we have defined
                        'other_orm' => 'doctrine.sql_logger_collector.other_orm',
                    ],
                ],

                // registering a new toolbar item with ZendDeveloperTools (name must␣
↪be the same of the collector name)
                'toolbar' => [
                    'entries' => [
                        // this is actually a name of a view script to use - you can␣
↪use your custom one
```

(continues on next page)

(continued from previous page)

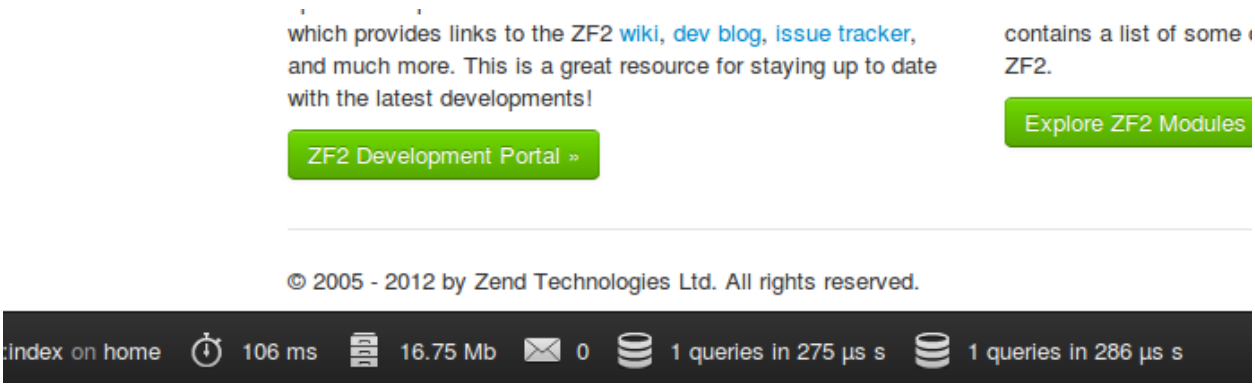
```
                        'other_orm' => 'zend-developer-tools/toolbar/doctrine-orm',
                    ],
                ],
            ],
        ];
    }

    public function getServiceConfiguration()
    {
        return [
            'factories' => [
                // defining a service (any name is valid as long as you use it␣
↪consistently across this example)
                'doctrine.sql_logger_collector.other_orm' => new␣
↪\DoctrineORMModule\Service\SQLLoggerCollectorFactory('other_orm'),
            ],
        ];
    }

    public function onBootstrap(\Zend\EventManager\EventInterface $e)
    {
        $config = $e->getTarget()->getServiceManager()->get('Config');

        if (isset($config['zenddevelopertools']['profiler']['enabled'])
            && $config['zenddevelopertools']['profiler']['enabled']
        ) {
            // when ZendDeveloperTools is enabled, initialize the sql collector
            $app->getServiceManager()->get('doctrine.sql_logger_collector.other_orm');
        }
    }
}
```

This example will simply generate a new icon in the toolbar, with the log results of your `other_orm` connection:



CHAPTER 2

Configuration

2.1 Register a Custom DQL Function

```
return [
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'numeric_functions' => [
                    'ROUND' => 'Db\DoctrineExtensions\Query\Mysql\Round',
                ],
            ],
        ],
    ],
];
```

2.2 Register a Type mapping

```
return [
    'doctrine' => [
        'connection' => [
            'orm_default' => [
                'doctrine_type_mappings' => [
                    'enum' => 'string',
                ],
            ],
        ],
    ],
];
```

2.3 How to add new type

```
return [
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'types' => [
                    'newtype' => 'Db\DBAL\Types\NewType',
                ],
            ],
        ],
    ],
];
```

```
return [
    'doctrine' => [
        'connection' => [
            'orm_default' => [
                'doctrine_type_mappings' => [
                    'mytype' => 'mytype',
                ],
            ],
        ],
    ],
];
```

2.4 Doctrine Type Comment

Option to set the doctrine type comment (DC2Type:myType) for custom types

```
return [
    'doctrine' => [
        'connection' => [
            'orm_default' => [
                'doctrineCommentedTypes' => [
                    'mytype',
                ],
            ],
        ],
    ],
];
```

2.5 Built-in Resolver

How to Define Relationships with Abstract Classes and Interfaces (ResolveTargetEntityListener)

```
return [
    'doctrine' => [
        'entity_resolver' => [
            'orm_default' => [
                'resolvers' => [
```

(continues on next page)

(continued from previous page)

```
                    'Acme\\InvoiceModule\\Model\\InvoiceSubjectInterface',
                    'Acme\\CustomerModule\\Entity\\Customer',
                ],
            ],
        ],
    ],
];
```

2.6 Set a Custom Default Repository

```
return [
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'default_repository_class_name' => 'MyCustomRepository',
            ],
        ],
    ],
];
```

2.7 How to Use Two Connections

See also this blog article.

```
return [
    'doctrine' => [
        'connection' => [
            'orm_crawler' => [
                'driverClass'   => 'Doctrine\DBAL\Driver\PDOMySql\Driver',
                'eventmanager'  => 'orm_crawler',
                'configuration' => 'orm_crawler',
                'params'        => [
                    'host'     => 'localhost',
                    'port'     => '3306',
                    'user'     => 'root',
                    'password' => 'root',
                    'dbname'   => 'crawler',
                    'driverOptions' => [
                        1002 => 'SET NAMES utf8',
                    ],
                ],
            ],
        ],

        'configuration' => [
            'orm_crawler' => [
                'metadata_cache'    => 'array',
                'query_cache'       => 'array',
                'result_cache'      => 'array',
                'hydration_cache'   => 'array',
                'driver'            => 'orm_crawler_chain',
```

(continues on next page)

(continued from previous page)

```
                'generate_proxies'  => true,
                'proxy_dir'         => 'data/DoctrineORMModule/Proxy',
                'proxy_namespace'   => 'DoctrineORMModule\Proxy',
                'filters'           => [],
            ],
        ],

        'driver' => [
            'orm_crawler_annotation' => [
                'class' => 'Doctrine\ORM\Mapping\Driver\AnnotationDriver',
                'cache' => 'array',
                'paths' => [
                    __DIR__ . '/../src/Crawler/Entity',
                ],
            ],
            'orm_crawler_chain' => [
                'class'   => 'Doctrine\ORM\Mapping\Driver\DriverChain',
                'drivers' => [
                    'Crawler\Entity' =>  'orm_crawler_annotation',
                ],
            ],
        ],

        'entitymanager' => [
            'orm_crawler' => [
                'connection'    => 'orm_crawler',
                'configuration' => 'orm_crawler',
            ],
        ],

        'eventmanager' => [
            'orm_crawler' => [],
        ],

        'sql_logger_collector' => [
            'orm_crawler' => [],
        ],

        'entity_resolver' => [
            'orm_crawler' => [],
        ],
    ],
];
```

The `DoctrineModule\ServiceFactory\AbstractDoctrineServiceFactory` will create the following objects as needed: * 'doctrine.connection.orm_crawler' * 'doctrine.configuration.orm_crawler' * 'doctrine.entitymanager.orm_crawler' * 'doctrine.driver.orm_crawler' * 'doctrine.eventmanager.orm_crawler' * 'doctrine.entity_resolver.orm_crawler' * 'doctrine.sql_logger_collector.orm_crawler'

You can retrieve them from the service manager via their keys.

2.8 How to Use Naming Strategy

Official documentation

Zend Configuration

```
return [
    'service_manager' => [
        'invokables' => [
            'Doctrine\ORM\Mapping\UnderscoreNamingStrategy' =>
↪'Doctrine\ORM\Mapping\UnderscoreNamingStrategy',
        ],
    ],
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'naming_strategy' => 'Doctrine\ORM\Mapping\UnderscoreNamingStrategy',
            ],
        ],
    ],
];
```

2.9 How to Use Quote Strategy

Official documentation

Zend Configuration

```
return [
    'service_manager' => [
        'invokables' => [
            'Doctrine\ORM\Mapping\AnsiQuoteStrategy' =>
↪'Doctrine\ORM\Mapping\AnsiQuoteStrategy',
        ],
    ],
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'quote_strategy' => 'Doctrine\ORM\Mapping\AnsiQuoteStrategy',
            ],
        ],
    ],
];
```

CHAPTER 3

Caching

Caching is very important in Doctrine.

In this example for Metadata, Queries, and Results we set an array cache for the result_cache. Please note the array cache is for development only and shown here along side other cache types.

If you want to set a cache for query, result and metadata, you can specify this inside your `config/autoload/local.php`

```
return [
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'query_cache'       => 'filesystem',
                'result_cache'      => 'array',
                'metadata_cache'    => 'apc',
                'hydration_cache'   => 'memcached',
            ],
        ],
    ],
];
```

The previous configuration takes into consideration different cache adapters. You can specify any other adapter that implements the `Doctrine\Common\Cache\Cache` interface. Find more here.

3.1 Example with Redis

```
return [
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'query_cache'       => 'redis',
                'result_cache'      => 'redis',
```

(continues on next page)

(continued from previous page)

```
                'metadata_cache'    => 'redis',
                'hydration_cache'   => 'redis',
            ],
        ],
    ],
];
```

In this case you have to specify a custom factory in your `service_manager` configuration to create a `Redis` object:

```
// module.config.php
return [
    'service_manager' => [
        'factories' => [
            __NAMESPACE__ . '\Cache\Redis' => __NAMESPACE__ . '\Cache\RedisFactory',
        ],
    ],
    'doctrine' => [
        'cache' => [
            'redis' => [
                'namespace' => __NAMESPACE__ . '_Doctrine',
                'instance'  => __NAMESPACE__ . '\Cache\Redis',
            ],
        ],
    ],
];
```

```
// RedisFactory.php
namespace YourModule\Cache;

use Zend\ServiceManager\FactoryInterface;
use Zend\ServiceManager\ServiceLocatorInterface;

class RedisFactory implements FactoryInterface
{
    public function createService(ServiceLocatorInterface $serviceLocator)
    {
        $redis = new Redis();
        $redis->connect('127.0.0.1', 6379);

        return $redis;
    }
}
```

Read more about Caching.

3.2 How to enable and configure Second Level Cache

```
return [
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'result_cache' => 'redis', // Second level cache reuse the cache␣
↪defined in result cache
```

(continues on next page)

(continued from previous page)

```
                'second_level_cache' => [
                    'enabled'               => true,
                    'default_lifetime'      => 200,
                    'default_lock_lifetime' => 500,
                    'file_lock_region_directory' => __DIR__ . '/../my_dir',
                    'regions' => [
                        'My\FirstRegion\Name' => [
                            'lifetime'      => 800,
                            'lock_lifetime' => 1000,
                        ],
                        'My\SecondRegion\Name' => [
                            'lifetime'      => 10,
                            'lock_lifetime' => 20,
                        ],
                    ],
                ],
            ],
        ],
    ],
];
```

You also need to add the `Cache` annotation to your model (read more). Read more about Second Level Cache.

CHAPTER 4

Doctrine Migrations

Support for the migrations library is included. Only one migration configuration is possible.

4.1 Configure

```
return [
    'doctrine' => [
        'migrations_configuration' => [
            'orm_default' => [
                'directory' => 'path/to/migrations/dir',
                'name' => 'Migrations Name',
                'namespace' => 'Migrations  Namespace',
                'table' => 'migrations_table',
                'column' => 'version',
                'custom_template' => null,
            ],
        ],
    ],
];
```

4.2 Multiple Migration Configurations

At this time if you want to have migrations for multiple entity manager database configurations you must use the .phar archive and external configuration files.

CHAPTER 5

Miscellaneous

The items listed below are optional and intended to enhance integration between Zend Framework and Doctrine 2.

5.1 ObjectExists Validator and NoObjectExists Validator

ObjectExists and NoObjectExists are validators similar to Zend Validators. You can pass a variety of options to determine validity. The most basic use case requires an entity manager, an entity, and a field. You also have the option of specifying a query_builder Closure to use if you want to fine tune the results.

```
<?php
$validator = new \DoctrineModule\Validator\NoObjectExists([
    // object repository to lookup
    'object_repository' => $serviceLocator->get('doctrine.entitymanager.orm_default')
        ->getRepository('Db\Entity\User'),

    // fields to match
    'fields' => ['username'],
]);

// following works also with simple values if the number of fields to be matched is 1
echo $validator->isValid(['username' => 'test']) ? 'Valid' : 'Invalid. A duplicate␣
↪was found.';
```

5.2 Authentication Adapter

The authentication adapter is intended to provide an adapter for `Zend\Authentication`. It works much like the `DbTable` adapter in the core framework. You must provide the entity manager instance, entity name, identity field, and credential field. You can optionally provide a callable method to perform hashing on the password prior to checking for validation.

```
<?php

use DoctrineModule\Authentication\Adapter\DoctrineObject as DoctrineObjectAdapter;

$adapter = DoctrineObjectAdapter(
    $entityManager,
    'Application\Test\Entity',
    'username', // optional, default shown
    'password',  // optional, default shown,
    function($identity, $credential) { // optional callable
        return \Application\Service\User::hashCredential(
            $credential,
            $identity->getSalt(),
            $identity->getAlgorithm()
        );
    }
);

$adapter->setIdentityValue('admin');
$adapter->setCredentialValue('password');
$result = $adapter->authenticate();

echo $result->isValid() ? 'Authenticated' : 'Could not authenticate';
```

5.3 Custom DBAL Types

To register custom Doctrine DBAL types add them to the `doctrine.configuration.orm_default.types` key in you configuration file:

```
<?php
return [
    'doctrine' => [
        'configuration' => [
            'orm_default' => [
                'types' => [
                    // You can override a default type
                    'date' => 'My\DBAL\Types\DateType',

                    // And set new ones
                    'tinyint' => 'My\DBAL\Types\TinyIntType',
                ],
            ],
        ],
    ],
];
```

With this configuration you may use them in your ORM entities to define field datatypes:

```
<?php

class User
{
    /**
     * @ORM\Column(type="date")
     */
```

(continues on next page)

(continued from previous page)

```
    protected $birthdate;

    /**
     * @ORM\Column(type="tinyint")
     */
    protected $houses;
}
```

To have Schema-Tool convert the underlying database type of your new "tinyint" directly into an instance of TinyIntType you have to additionally register this mapping with your database platform.

```
<?php
return [
    'doctrine' => [
        'connection' => [
            'orm_default' => [
                'doctrine_type_mappings' => [
                    'tinyint' => 'tinyint',
                ],
            ],
        ],
    ],
];
```

Now using Schema-Tool, whenever it finds a column of type "tinyint" it will convert it into a "tinyint" Doctrine Type instance for Schema representation. Keep in mind that you can easily produce clashes this way because each database type can only map to exactly one Doctrine mapping type.