doctrine-in-apigility-docs$_ptDocumentation$

Versão latest

out 03, 2017

Table of Contents

This will document how to create an API with Doctrine and Apigility.

CAPÍTULO 1

Introduction

The landscape of API strategies grows every day. In the field of PHP there are strategies from simple REST-only resources to fully Richardson Maturity Model Level 3 API engines. Apigility falls into the Level 3 category.

An API will serve data and Doctrine in Apigility tries, at it's core, to map Doctrine entities to API resources. So a Doctrine enabled resource in Apigility will provide GET, POST, PUT, PATCH, and DELETE. It also enables the ORM entity relationships to embed related data in a HAL (Hypertext Application Language) response. This allows complex joins between entities to be represented in the API response.

When should you use Doctrine in Apigility?

When you have a Doctrine project with properly mapped associations, metadata, between entities Doctrine in Apigility will give you a powerful head-start toward building an API. Correct metadata is absolutly core to building an API with this tool. To help design your ORM Skipper is strongly recommended.

You can use Doctrine in Apigility to serve pieces of your schema by filtering with hydrator strategies or you can serve your entire schema in an "Open-schema API" where relationships between entities are fully explored in the HAL _embedded data.

If you're familiar with the benefits of ORM and will use it in your project and you require a fully-fledged API engine than this API strategy may what you're looking for.

CAPÍTULO 2

About

Much of this document was written by Tom H Anderson and some material was compiled from other sources in the Apigility realm.

Tom was the original author of zfcampus/zf-apigility-doctrine and has created many supporting libraries for Doctrine in Apigility. He contributed much to the early development of Apigility and continues to give to open source in PHP.

As soon as Apigility was announced developers were asking for Doctrine support though none of us really knew what that ment. What was created serves as a strong platform for an ORM based API.

CAPÍTULO 3

Authentication

The official Apigility documentation describes the lightweight authentication infrastructure included with Apigility and how it is intended to be extended. Extending this infrastructure is exactly what is required for Doctrine in Apigility.

When building an API application the central authentication is usually done through an OAuth2 implicit and/or authorization_code grant type. To support authorization you must redirect users to a login page when a user tries to authenticate through these routes.

api-skeletons/zf-oauth2-doctrine-identity replaces the default AuthenticatedIdentity provided by Apigility. This class `->getIdentity()` returns the AccessToken granted through OAuth2. From the Access Token you can fetch the authenticated User entity and OAuth2 Client entity.

Authentication should occur every time a user tries to access a route they are not authorized for and the user is not already Authenticated. To trigger Authentication you must have authorization setup for your resources.

CAPÍTULO 4

Authorization

api-skeletons/zf-oauth2-doctrine-permissions-acl implements ACL resources for a User with a many-to-many relationship to a Role entity. It works in tandem with other zf-oauth2-doctrine libraries to create easy to implement authorization in a Doctrine in Apigility API.

CAPÍTULO 5

OAuth2 Adapter

api-skeletons/zf-oauth2-doctrine

This module provides a Doctrine adapter for zfcampus/zf-mvc-auth and zfcampus/zf-oauth2 and entity definitions for all aspects of OAuth2 including Authorization Code, Access Tokens, Refresh Tokens, JWT & JTI, and Scopes.

A Skipper module is provided to embed in your Entity Relationship Diagram.

Entity Relationship Diagram

Entity Relationship Diagram created with Skipper

The ERD module is located at media/OAuth2-orm.module.xml and is intended to be embedded in the ERD for your project.

Relations

The User entity is provided by your application and a dynamic join is made at runtime when the metadata is gathered for Doctrine. There is a dynamic join with Client, AuthorizationCode, AccessToken and RefreshToken.

The central OAuth2 entity is the Client. There is a dynamic join from the Client entity to the configured User entity. For every application owned by a User there will be a Client entry. The User referenced from a Client is the User who owns that Client.

The AuthorizationCode entity is used when a User connects from an application using OAuth2. The reference to the User entity from the AuthorizationCode entity is for the User that is trying to log into the Client referenced from the AuthorizationCode entity. The same is true for the AccessToken and RefreshToken entities.

The Scope entities are many to many relationships for Client, AuthorizationCode, AccessToken and RefreshToken. Scopes dictate what permissions a Client has into the API.

There is a one to one relationship from Client to PublicKey. This is because the encryption side of OAuth2 is less common and to encapsulate the encryption fields into the PublicKey entity. The JWT and JTI entities provide full support and their use in encryption falls outside the scope of this documentation.

Database Table Namespaces

All OAuth2 tables are suffixed with _OAuth2 such as Client_OAuth2. You can change these if you override. It is recommended your database table names match your entity names to provide canonical naming across your application. See also bushbaby/zf-oauth2-doctrine-mutatetablenames.

Application Configuration

Installation

Installation of this module uses composer. For composer documentation, please refer to getcomposer.org.

`composer require api-skeletons/zf-oauth2-doctrine "^1.0"`

Add this module to your application's configuration:

```
'modules' => [
   ...
   'ZF\OAuth2\Doctrine',
],
```

Global Configuration

Copy `config/oauth2.doctrine-orm.global.php.dist` to your autoload directory and rename to `oauth2.doctrine-orm.global.php` This config has multiple sections for multiple adapters. Out of the box this module provides a *default* adapter. You will need to edit this file with at least your User entity, which is not provided.

zfcampus/zf-mvc-auth Configuration

By default this module includes a *oauth2.doctrineadapter.default* adapter. The adapter is used to create storage from services. Add this configuration to your *config/autoload/zf-mvc-auth-oauth2-override.global.php*:

```
'zf-mvc-auth' => array(
    'authentication' => array(
        'adapters' => array(
            'oauth2_doctrine' => array(
                'adapter' => 'ZF\\MvcAuth\\Authentication\\OAuth2Adapter',
                'storage' => array(
                    'storage' => 'oauth2.doctrineadapter.default',
                    'route' => '/oauth',
                ),
            ),
        ),
    ),
),
```

zfcampus/zf-oauth2 Configuration

Add the default storage adapter to the zf-oauth default storage. `zfcampus/zf-oauth2` provides an `oauth2.local.php` file. This repository's recommendation is to create a new `config/autoload/oauth2.global.`

php file and set the following configuration as well as any OAuth2 server settings e.g. allow_implicit:

```
'zf-oauth2' => array(
    'storage' => 'oauth2.doctrineadapter.default',
```

Module Configuration

Using Default Entities

Details for creating your database with the included entities are outside the scope of this project. Generally this is done through doctrine/doctrine-orm-module with `php public/index.php orm:schema-tool:create`

By default this module uses the entities provided but you may use the adapter with your own entites (and map them in the mapping config section) by toggling this flag:

```
'zf-oauth2-doctrine' => [
    'default' => [
        'enable_default_entities' => true,
```

Customizing Many to One Mapping

If you need to customize the call to mapManyToOne, which creates the dynamic joins to the User entity from the default entites, you may add any parameters to the `['dynamic_mapping']['default_entity']['additional_mapping_data']` element. An example for a User entity with a primary key of user_id which does not conform to the metadata naming strategy is added to each entity:

```
'refresh_token_entity' => [
    'entity' => 'ZF\OAuth2\Doctrine\Entity\RefreshToken',
    'field' => 'refreshToken',
    'additional_mapping_data' => [
        'joinColumns' => [
            [
                'name' => 'user_id',
                'referencedColumnName' => 'user_id',
            ],
        ],
    ],
],
```

Identity field on User entity

By default this Doctrine adapter retrieves the user by the *username* field on the configured User entity. If you need to use a different or multiple fields you may do so via the `auth_identity_fields` key. For example, ZfcUser allows users to authenticate by username and/or email fields.

An example to match ZfcUser `auth_identity_fields` configuration:

```
'zf-oauth2-doctrine' => [
    'default' => [
        'auth_identity_fields' => ['username', 'email'],
```

User Entity Configuration

This repository supplies every entity you need to implement OAuth2 except the User entity. The reason is so the User entity can be decoupled from the OAuth2 Doctrine repository instead to be linked dynamically at run time. This allows, among other benefits, the ability to create an ERD without modifying the *OAuth2-orm.module.xml* module.

The User entity must implement `ZF\OAuth2\Doctrine\Entity\UserInterface`

The User entity for the unit test for this module is a good template to start from: https://github.com/api-skeletons/zf-oauth2-doctrine/blob/master/test/asset/module/Doctrine/src/Entity/User.php

Events

Events are fully supported. Return values are used if propagation is stopped allowing you to write your own handlers for any OAuth2 Adapter method. Each function which implements the OAuth2 interfaces may be attached to and optionally overridden.

Example Event Attachment

Module.php onBootstrap:

```
$doctrineOAuth2Adapter = $e->getParam('application')
    ->getServiceManager()
    ->get('oauth2.doctrineadapter.default')
     ;
$listenerAggregate = new \Application\EventSubscriber\OAuth2AggregateListener(
↪$objectManager);
$doctrineOAuth2Adapter->getEventManager()->attachAggregate($listenerAggregate);
```

ApplicationEventSubscriberOAuth2AggregateListener:

```
namespace Application\EventSubscriber;

use Zend\EventManager\Event;
use Zend\EventManager\AbstractListenerAggregate;
use Zend\EventManager\EventManagerInterface;

class OAuth2AggregateListener extends AbstractListenerAggregate
{
    protected $handlers = array();
    protected $logInAs;

    public function attach(EventManagerInterface $events)
    {
        $this->handlers[] = $events->attach('checkUserCredentials', array($this,
↪'checkUserCredentials'));
    }

    /**
     * Do work such as non-standard encrypted password checking
     */
    public function checkUserCredentials(Event $e)
    {
        if ($e->getParams()['username'] == 'specialUser') {
            $e->stopPropagation();

            return true;
        }
    }
}
```

zf-doctrine-apigility Integration

Validate zf-apigility-doctrine resources

To validate the OAuth2 session with Query Create Filters and Query Providers implement `ZF\OAuth2\Doctrine\OAuth2ServerInterface` and use `ZF\OAuth2\Doctrine\OAuth2ServerTrait`. Then call `$result = $this->validateOAuth2($scope);` in the filter function.

Override Default Entities

This example shows how to override default entities and modify them to use BIGINT. The same procedure is used for any other time default entities will be overridden.

The default foreign keys and primary keys in `zf-oauth2-doctrine` are standard integers. Many times databases will plan for the future and use BIGINT for these keys. It's not possible to create referential integrity between an INTEGER and a BIGINT. It is possible to quickly implement BIGINT integers in `zf-oauth2-doctrine`.

First you'll need a new module for the modified xml. Call it `OAuth2`. Next copy the `zf-oauth2-doctrine/config/orm/*.xml` into your new module `OAuth2/config/orm` (create this directory).

Edit the xml and change all *integers* to *bigint*. Next edit your *oauth2.doctrine-orm.global.php* file and set *'enable_default_entities'* => *false*

Now in OAuth2/config/module.config.php add the Doctrine config and alias the xml:

```
'doctrine' => array(
    'driver' => array(
        'oauth2_driver' => array(
            'class' => 'Doctrine\\ORM\\Mapping\\Driver\\XmlDriver',
            'paths' => array(
                0 => __DIR__ . '/orm',
            ),
        ),
        'orm_default' => array(
            'class' => 'Doctrine\\ORM\\Mapping\\Driver\\DriverChain',
            'drivers' => array(
                'ZF\\OAuth2\\Doctrine\\Entity' => 'oauth2_driver',
            ),
        ),
    ),
),
```

Multiple OAuth2 Instances

You may have multiple sets of distinct OAuth2 servers on the same application instance. This describes how to create a second OAuth2 server with it's own entities.

Create a new module called *SecondOAuth2* and copy the entities from *zf-oauth2-doctrine* into it's *src/SecondOAuth2/Entity* directory. Change all the namespaces to *SecondOAuth2* and this includes autogenerated fully qualified name spaces in the add and remove functions for collections.

Copy the ORM XML data from *zf-oauth2-doctine* to *SecondOAuth2/config/orm* and edit it for the new module. Be sure to change the table names. Add doctrine config to module.config.php

Copy this code into the *SecondOAuth2 Module.php* ‘‘‘php

public function onBootstrap(MvcEvent $e) {

/** @var ServiceLocatorInterface $serviceManager */ $serviceManager = $e->getParam(‘application’)->getServiceManager(); $serviceManager->get(‘oauth2.doctrineadapter.second’)->bootstrap($e);

}

public function getServiceConfig() {

return [

‘factories’ => [

‘oauth2.doctrineadapter.second’ => function ($serviceManager) { /** @var ServiceLocatorInterface | ContainerInterface $serviceManager / *$globalConfig = $serviceManager->get(‘Config’); $config = new Config($globalConfig[’zf-oauth2-doctrine’][’second’]);* /* @var DoctrineAdapterFactory $factory */ $factory = $serviceManager->get(DoctrineAdapterFactory::class); $factory->setConfig($config);

return $factory->createService($serviceManager);

}

],

];

}

‘‘‘ Create a copy of the default config in oauth2.doctrine-orm.global.php and name the block ‘second’. Create a new User entity and assign that entity to the default group. There should not be a reason to have two OAuth2 servers connected to the same User entity. Change namespaces as needed.

Finally follow the configuration section of this documentation to configure the second OAuth2 server.

Add-Ons

api-skeletons/zf-oauth2-doctrine-console

Console routes to manage OAuth2 entities. Add and list Clients, Scopes, and more from the command line.

api-skeletons/zf-oauth2-doctrine-identity

This should have been a part of zf-oauth2-doctrine from the beginning but because zf-oauth2-doctrine is so mature identity mapping has been implemented in this add-on repository. This will change the AuthenticatedIdentity of [zfcampus/zf-mvc-auth](https://github.com/zfcampus/zf-mvc-auth) to a Doctrine enabled identity with quick access to the User, Client, and AccessToken entities.

api-skeletons/zf-oauth2-doctrine-permissions-acl

ACL Permissions and Authenticated Identity management. This uses interfaces rather than defining entities to create guards on resources based on [api-skeletons/zf-oauth2-doctrine-identity](https://github.com/API-Skeletons/zf-oauth2-doctrine-identity)

bushbaby/zf-oauth2-doctrine-mutatetablenames

Allows configuration of the table names that this module uses without overriding default entities.

CAPÍTULO 6

OAuth2 Adapter Console Management

api-skeletons/zf-oauth2-doctrine-console

About

This repository provides console routes to manage a headless OAuth2 server.

Installation

Installation of this module uses composer. For composer documentation, please refer to getcomposer.org:

```
composer require api-skeletons/zf-oauth2-doctrine-console "*"
```

Add this module to your application's configuration:

```
'modules' => array(
   ...
   'ZF\OAuth2\Doctrine\Console',
),
```

Console Routes

- `oauth2:client:create` Create a new client with or without a user.
- `oauth2:client:update` Update a client.
- `oauth2:client:delete` Delete a client.
- `oauth2:client:list` List all clients.
- `oauth2:scope:create` Create a scope.

- `oauth2:scope:update` Update a scope.
- `oauth2:scope:delete` Delete a scope.
- `oauth2:scope:list` List all scopes.
- `oauth2:public-key:create` Create the public/private key record for the given client. This data is used to sign JWT access tokens. Each client may have only one key pair.
- `oauth2:public-key:delete` Remove the key pair public key from a client.
- `oauth2:jwt:create` Create a new JWT for a given client. This JWT will be used by an oauth2 connection requesting a grant_type of `urn:ietf:params:oauth:grant-type:jwt-bearer`. Creating the JWT puts the oauth2 connection request's public key in place in the OAuth2 tables.
- `oauth2:jwt:delete` Delete a JWT.
- `oauth2:jwt:list` List all JWT.

For the connecting side of JWT, zf-oauth2-client provides a command line tool to generate a JWT reqeust. See also http://bshaffer.github.io/oauth2-server-php-docs/grant-types/jwt-bearer/

CAPÍTULO 7

Hydrator Strategies

api-skeletons/zf-doctrine-hydrator

A collection of hydrator strategies for phpro/zf-doctrine-hydration-module written for use with zfcampus/zf-apigility-doctrine

Installation

Installation of this module uses composer. For composer documentation, please refer to getcomposer.org:

```
composer require api-skeletons/zf-doctrine-hydrator "*"
```

Add this module to your application's configuration:

```
'modules' => array(
   ...
   'ZF\Doctrine\Hydrator',
),
```

StrategyCollectionExtract

Extract a collection. Often when this is used the entities in the collection have a many to one relationship with the entity the collection belongs to. In this case it is recommended you use the EntityLink for the parent on the child.

StrategyCollectionLink

This strategy will replace a collection with just a self link to the collection. It uses zfcampus/zf-doctrine-querybuilder to create a query for just the collection data.

StrategyEntityLink

Will replace an entity with just a self link to the entity. Use when a child is part of a collection referenced by the parent and the parent is a property of the child.

How to use these hydration strategies

In your configuration for your Doctrine in Apigility API the `doctrine-hydrator` section add a strategy to an entity. In this example an Artist is a member of a Band which, by example, is a many to one relationship (an artist can only have one band):

```
'doctrine-hydrator' => array(
    'DatabaseApi\\V1\\Rest\\Album\\SongHydrator' => array(
        'entity_class' => 'Database\\Entity\\Song',
        'object_manager' => 'doctrine.entitymanager.orm_default',
        'by_value' => true,
        'use_generated_hydrator' => true,
    ),
    'DatabaseApi\\V1\\Rest\\Album\\AlbumHydrator' => array(
        'entity_class' => 'Database\\Entity\\Album',
        'object_manager' => 'doctrine.entitymanager.orm_default',
        'by_value' => true,
        'strategies' => array(
            'artist' => 'ZF\Doctrine\Hydrator\Strategy\EntityLink',
            'song' => 'ZF\Doctrine\Hydrator\Strategy\CollectionLink',
        ),
        'use_generated_hydrator' => true,
    ),
    'DatabaseApi\\V1\\Rest\\Artist\\ArtistHydrator' => array(
        'entity_class' => 'Database\\Entity\\Artist',
        'object_manager' => 'doctrine.entitymanager.orm_default',
        'by_value' => true,
        'strategies' => array(
            'album' => 'ZF\Doctrine\Hydrator\Strategy\CollectionExtract',
        ),
        'use_generated_hydrator' => true,
    ),
```

When an Artist is queried all Albums for the Artist will be returned. For each Album the Artist will be returned only as a self link. The result of a call to *https://api/artist/1* will look like:

```
{
    "id": 1,
    "name": "Soft Cell",
    "_embedded": {
        "album": [
            {
                "id": 1,
                "name": "Non-Stop Erotic Cabaret",
                "_embedded": {
                    "artist": {
                        "_links": {
                            "self": "https://api/artist/1"
                        }
                    },
                    "song": {
                        "_links": {
                            "self": "https://api/song?filter%5B0%5D%5Bfield%5D=album&
↪filter%5B0%5D%5Btype%5D=eq&filter%5B0%5D%5Bvalue%5D=1"
                        }
                    }
                },
                "_links": {
                    "self": "https://api/album/1"
                }
            }
        ],
    },
    "_links": {
        "self": "https://api/artist/1"
    }
}
```

CAPÍTULO 8

Filtering and Sorting

zfcampus/zf-doctrine-querybuilder

This library provides query builder directives from array parameters. This library was designed to apply filters from an HTTP request to give an API fluent filter and order-by dialects.

This library implements rich filter and sorting access to developers implementing your API.