zf-doctrine-graphql Documentation

Release latest

Jan 11, 2019

Contents

Welcome to the documentation for api-skeletons/zf-doctrine-graphql

This repository uses Doctrine metadata to introspect your entities and builds relationships dynamically covering your entire schema, if you wish, allowing deep GraphQL queries on your data with a single entry point.

CHAPTER 1

About

Authored by Tom H Anderson <tom.h.anderson@gmail.com> of API Skeletons and a member of the Doctrine Core team specializing in Zend modules, this module is the fourth offering in the space of Doctrine and GraphQL according to Packagist. Other implementations have used strategies such as annotations or GraphQL types which are only one entity deep and only support a single object manager.

This repository was created because using Hydrators to extract data from entities is the correct way to configure the output from the entities. Then, allowing mulitiple hydrator configurations allows you to create GraphQL endpoints which are specific to a part of the data. For instance, you may want to *rot13()* all email addresses for normal user GraphQL queries but return them unencrypted for an admin user. With this library such data manipulation is possible.

The hydrator factory for this repository was taken from phpro/zf-doctrine-hydration-module and allows for customization of each field for each entity using Hydrator Stratigies. It allows for Hydrator Filters to completely remove data from the result. It allows for Hydrator Naming Strategies. With all of these hydrator features this repository delivers superior data mutability for any case you may have to serve it over GraphQL.

This repository allows for multiple object managers. Each hydrator configuration section specifies a specific object manager.

This repository allows for multiple GraphQL Schemas. Served via different RPC endpoints on your application or through a more complicated selection of Schema based on input parameters, this repository is flexible enough for any GraphQL needs.

Doctrine provides full navigation of a database schema when properly configured and this repository leverages that flexibility to mirror the full database schema navigation to provide GraphQL queries as complex as your data.

CHAPTER 2

Installation

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

```
$ composer require api-skeletons/zf-doctrine-graphql
```

Once installed, add **ZF\Doctrine\GraphQL** to your list of modules inside *config/application.config.php* or *config/modules.config.php*.

2.1 zf-component-installer

If you use zf-component-installer, that plugin will install zf-doctrine-graphql as a module for you.

2.2 zf-doctrine-criteria configuration

You must copy the config for zf-doctrine-criteria to your autoload directory:

```
$ cp vendor/api-skeletons/zf-doctrine-criteria/config/zf-doctrine-criteria.global.php.
↪dist config/autoload/zf-doctrine-criteria.global.php
```

CHAPTER 3

Use

This example merges work from a factory into the example. Moving the *$container* calls to a factory and injecting them into an RPC object will yield a working example.:

```
use Exception;
use GraphQL\GraphQL;
use GraphQL\Type\Schema;
use GraphQL\Type\Definition\Type;
use GraphQL\Type\Definition\ObjectType;
use ZF\Doctrine\GraphQL\Type\Loader as TypeLoader;
use ZF\Doctrine\GraphQL\Filter\Loader as FilterLoader;
use ZF\Doctrine\GraphQL\Resolve\Loader as ResolveLoader;
use ZF\Doctrine\GraphQL\Context;

$typeLoader = $container->get(TypeLoader::class);
$filterLoader = $container->get(FilterLoader::class);
$resolveLoader = $container->get(ResolveLoader::class);

$input = $_POST;

// Context is used for configuration level variables and is optional
$context = (new Context())
    ->setLimit(1000)
    ->setHydratorSection('default')
    ->setUseHydratorCache(true)
    ;

$schema = new Schema([
    'query' => new ObjectType([
        'name' => 'query',
        'fields' => [
            'artist' => [
                'type' => Type::listOf($typeLoader(Entity\Artist::class, $context)),
                'args' => [
                    'filter' => $filterLoader(Entity\Artist::class, $context),
```

(continues on next page)

(continued from previous page)

```
                ],
                'resolve' => $resolveLoader(Entity\Artist::class, $context),
            ],
            'performance' => [
                'type' => Type::listOf($typeLoader(Entity\Performance::class,
↪$context)),
                'args' => [
                    'filter' => $filterLoader(Entity\Performance::class, $context),
                ],
                'resolve' => $resolveLoader(Entity\Performance::class, $context),
            ],
        ],
    ]),
]);

$query = $input['query'];
$variableValues = $input['variables'] ?? null;

try {
    // Context in the `executeQuery` is required.  If you do not assign a specific␣
↪context as shown
    // you still need to send a `new Context()` to `executeQuery`.
    $result = GraphQL::executeQuery($schema, $query, $rootValue = null, $context,
↪$variableValues);
    $output = $result->toArray();
} catch (Exception $e) {
    $output = [
        'errors' => [[
            'exception' => $e->getMessage(),
        ]]
    ];
}

echo json_encode($output);
```

CHAPTER 4

Schema Configuration

4.1 Context

The *Context* object provided enables configuration of GraphQL through the following options:

- limit - Set a maximum limit of each data section in a query
- hydratorSection - Which section within the hydrator configuration should be used
- useHydratorCache - By default all hydrator operations are not cached. Enabling this value will cache all the hydrator operation in anticipation that the result may be reused.

Context is the configuration for each GraphQL entry point. This allows unlimited configuration through multiple hydrator sections.

You must use the same context object for a Query as you assign to the Loader. This may be done via different Schemas or different RPC endpoints.

4.2 useHydratorCache Context Option

The hydrator cache by defaults stores only the most recent hydrator extract data in anticipation that the next call to the FieldResolver will be the same object and the cache can be used. If the same object is not requesed for extraction then the cache is flushed and the new result is cached.

For a query

```
{ artist ( filter: { id: 2 } ) { performance { performanceDate } } }
```

All performance dates for the artist 2 will be returned. Internally each performance is extracted according to the hydrator filters and strategies assigned to the hydrator section and entity. This may be many more fields than just performanceDate. And since we are only interested in one value setting useHydratorCache to false will flush the cache with each new object so once a performanceDate is read and the next performance is sent to the FieldResolver the previous hydrator extract data is purged.

For a query

```
{ performance ( filter: { id:1 } ) { performanceDate set1 set2 artist { name } set3 }␣
↪}
```

useHydratorCache set to true will cause set3 to be pulled from the cache. If it were set to false set3 would generate a new hydrator extract operation on an entity which had already been extracted once before.

useHydratorCache set to false will fetch set1 and set2 from the single-entity cache created by the performanceDate.

4.3 Supported Data Types

This module would like to support all datatypes representable in a GraphQL response. At this time these data types are supported:

```
array    - Arrays are handled as arrays of strings because Doctrine does not type the␣
↪values of the array.
tinyint
smallint
integer
int
bigint
boolean
decimal
float
string
text
datetime
```

Dates are handled as ISO 8601 e.g. *2004-02-12T15:19:21+00:00*

If you have need to support a datatype not listed here please create an issue on the github project.

4.4 Provided Tools

There are three tools this library provides to help you build your GraphQL Schema.

- **TypeLoader** - This tool creates a GraphQL type for a top-level entity and all related entities beneath it. It also creates resolvers for related collections using the api-skeletons/zf-doctrine-criteria library.
- **FilterLoader** - This tool creates filters for all non-related fields (collections) such as strings, integers, etc. These filters are built from the zfcampus/zf-doctrine-querybuilder library.
- **ResolveLoader** - This tool builds the querybuilder object and queries the database based on the FilterLoader filters.

Each of these tools takes a fully qualified entity name as a paramter allowing you to create a top level GraphQL query field for any entity.

There is not a tool for mutations. Those are left to the developer to build.

CHAPTER 5

Hydrator Configuration

Even deveopers who have used Doctrine or an ORM a lot may not have experience with hydrators. This section is to educate and help the developer understand hydrators and how to use them in relation to Doctrine ORM and GraphQL.

A hydrator moves data into and out of an object as an array. The array may contain scalar values, arrays, and closures. For instance, if you have an entity with the fields:

```
id
name
description
```

a hydrator can create an array from your entity resulting in:

```
$array['id']
$array['name']
$array['description']
```

The Zend Framework documentaion on Hydrators. is a good read for background about coding hydrators from scratch.

The Doctrine Hydrator documentation is more complete and more pertinent to this repository. A notable section is By Value and By Reference,

5.1 Generating a Skeleton Configuration

This module uses hydrators to extract data from the Doctrine entities. You can configure multiple sections of hydrators so one permissioned user may receive different data than a different permission or one query to an entity may return different fields than another query to the same entity.

Because creating hydrator configurations for every section for every entity in your object manager(s) is tedious this module provides an auto-generating configuration tool.

To generate configuration:

```
php public/index.php graphql:config-skeleton [--hydrator-sections=] [--object-
↪manager=]
```

(continues on next page)

(continued from previous page)

The hydrator-sections parameter is a comma delimited list of sections to generate such as *default,admin*.

The object-manager parameter is optional and defaults to *doctrine.entitymanager.orm_default*. For each object manager you want to serve data with in your application create a configuration using this tool. The tool outputs a configuration file. Write the file to your project root location then move it to your *config/autoload* directory.

```
php public/index.php graphql:config-skeleton > zf-doctrine-graphql-orm_default.global.
↪php
mv zf-doctrine-graphql-orm_default.global.php config/autoload
```

(Writing directly into the *config/autoload* directory is not recommended at run time.)

Default hydrator strategies and filters are set for every association and field in your ORM. Modify each hydrator configuration section with your hydrator strategies and hydrator filters as needed.

5.2 Generated Configuration

Here is an example of a generated configuration:

```
'ZF\\Doctrine\\GraphQL\\Hydrator\\ZF_Doctrine_Audit_Entity_Revision' => [
    'default' => [
        'entity_class' => \ZF\Doctrine\Audit\Entity\Revision::class,
        'object_manager' => 'doctrine.entitymanager.orm_zf_doctrine_audit',
        'by_value' => true,
        'use_generated_hydrator' => true,
        'naming_strategy' => null,
        'hydrator' => null,
        'strategies' => [
            'id' => \ZF\Doctrine\GraphQL\Hydrator\Strategy\ToInteger::class,
            'comment' => \ZF\Doctrine\GraphQL\Hydrator\Strategy\FieldDefault::class,
            'connectionId' => \ZF\Doctrine\GraphQL\Hydrator\Strategy\ToInteger::class,
            'createdAt' => \ZF\Doctrine\GraphQL\Hydrator\Strategy\FieldDefault::class,
            'userEmail' => \ZF\Doctrine\GraphQL\Hydrator\Strategy\FieldDefault::class,
            'userId' => \ZF\Doctrine\GraphQL\Hydrator\Strategy\ToInteger::class,
            'userName' => \ZF\Doctrine\GraphQL\Hydrator\Strategy\FieldDefault::class,
            'revisionEntity' =>␣
↪\ZF\Doctrine\GraphQL\Hydrator\Strategy\AssociationDefault::class,
        ],
        'filters' => [
            'default' => [
                'condition' => 'and',
                'filter' => \ZF\Doctrine\GraphQL\Hydrator\Filter\FilterDefault::class,
            ],
        ],
        'documentation' => [
            '_entity' => '',
            'id' => '',
            'comment' => '',
            'connectionId' => '',
            'createdAt' => '',
            'userEmail' => '',
            'userId' => '',
            'userName' => '',
```

(continues on next page)

(continued from previous page)

```
        ],
    ],
],
```

The *entity_class* is the fully qualified entity class name this configuration section is for.

The *object_manager* is the service manager alias for the object manager which manages the *entity_class*.

by_value is an important switch. When set to *true* the values for the entity will be fetched using their getter methods such as *getName()* for a *name* field. When set to *false* the entity will be Reflected and the property value of the entity class will be extracte *by reference*. If your entities are not extracting properly try toggling this value.

by_value set to *false* is useful when your entity does not have getter methods such as a dynamically created entity. API-Skeletons/zf-doctrine-audit is a good example for this. The dynamically generated auditing entities do not have getter methods but do have properties to contain the field values. These can be extracted *by reference*.

use_generated_hydrator is usually set to true. Because GraphQL uses hydrators for extraction only this value is not used. But if you want to use the same configured hydrators to hydrate an entity please see the code for its use.

hydrator allows complete overriding of the extract service. If set the extract and hydrate services will be assigned to the specified hydrator.

naming_strategy is an instance of **ZendHydratorNamingStrategyNamingStrategyInterface** and is a service manager alias. You may only have one *naming_strategy* per hydrator configuration. A naming strategy lets you rename fields.

strategies are quite important for extracting entities. These can change the extracted value in whatever way you wish such as *rot13()* email addresses. They can return an empty value but for that case it's better to filter out the field completely.

filters are toggle switches for fields. If you return false for a field name it will remove the field from the extract result.

documentation section is for fields only. Relations are not documented because that is not supported by GraphiQL. Use this section to document each field and the entity. The reserved name *_entity* contains the documentation for the entity.

5.3 Strategies

There are some hydrator stragegies included with this module. In GraphQL types are very important and this module introspects your ORM metadata to correctly type against GraphQL types. By default integer, float, and boolean fields are automatically assigned to the correct hydrator strategy.

5.4 Many to Many Owning Side Relationships

```
{ artist { user { role { user { name } } } } }
```

This query would return all user names who share the same role permissions as the user who created the artist. To prevent this the *graphql:config-skeleton* command nullifies the owning side of many to many relations by default causing an error when the query tries to go from role > user but not when it goes from user > role becuase role is the owning side of the many to many relationship. See NullifyOwningAssociation for more information.

5.5 Documenting Entities

Introspection of entities is a core component to GraphQL. The introspection allows you to document your types. Because entities are types there is a section inside each hydrator configuration for documenting your entity and fields through introspection.

```
'documentation' => [
    '_entity' => 'The Artist entity contains bands, groups, and individual performers.
↪',
    'id' => 'Primary Key for the Artist',
    'abbreviation' => 'An abbreviation for the Artist',
    'createdAt' => 'DateTime the Artist record was created',
    'description' => 'A description of the Artist',
    'icon' => 'The Artist icon',
    'name' => 'The name of the performer.',
],
```

There is one special field, *_entity* which is the description for the entity itself. The rest of the fields describe documentation for each field.

Documentation is specific to each hydrator section allowing you to describe the same entity in different ways. The Documentation will be returned in tools like GraphiQL

GraphiQL is the standard for browsing introspected GraphQL types. zf-doctrine-graphql fully supports GraphiQL.

CHAPTER 6

Running Queries

This section is intended for the developer who needs to write queries against an implementation of this repository.

Queries are not special to this repository. The format of queries are exactly what GraphQL is spec'd out to be. For each implementation of GraphQL the filtering of data is not defined. In order to build the filters for this an underscore approach is used. *fieldName_filter* is the format for all filters.

An example query:

Fetch at most 100 performances in CA for each artist with 'Dead' in their name.

```
$query = "{ artist ( filter: { name_contains: \"Dead\" } ) { name performance (␣
↪filter: { _limit: 100 state:\"CA\" } ) { performanceDate venue } } }";
```

6.1 Filters

For each field, which is not a reference to another entity, a colletion of filters exist. Given an entity which contains a *name* field you may directly filter the name using

```
filter: { name: "Grateful Dead" }
```

You may only use each field's filter once per filter(). Should a child record have the same name as a parent it will share the filter names but fitlers are specific to the entity they filter upon.

Provided Filters:

```
fieldName_eq          -  Equals; same as name: value.  DateTime not supported.
fieldName_neq         -  Not Equals
fieldName_gt          -  Greater Than
fieldName_lt          -  Less Than
fieldName_gte         -  Greater Than or Equal To
fieldName_lte         -  Less Than or Equal To
fieldName_in          -  Filter for values in an array
fieldName_notin       -  Filter for values not in an array
```

(continues on next page)

(continued from previous page)

```
fieldName_between    -  Filter between `from` and `to` values.  Good substitute for␣
↪DateTime Equals.
fieldName_contains   -  Strings only. Similar to a Like query as `like '%value%'`
fieldName_startswith -  Strings only. A like query from the beginning of the value␣
↪`like 'value%'`
fieldName_endswith   -  Strings only. A like query from the end of the value `like '
↪%value'`
fieldName_memberof   -  Matches a value in an array field.
fieldName_isnull     -  Takes a boolean.  If TRUE return results where the field is␣
↪null.
                        If FALSE returns results where the field is not null.
                        NOTE: acts as "isEmpty" for collection filters.  A value of␣
↪false will
                        be handled as though it were null.
fieldName_sort       -  Sort the result by this field.  Value is 'asc' or 'desc'
fieldName_distinct   -  Return a unique list of fieldName.  Only one distinct␣
↪fieldName allowed per filter.
```

The format for using these filters is:

```
filter: { name_endswith: "Dead" }
```

For isnull the parameter is a boolean

```
filter: { name_isnull: false  }
```

For in and notin an array of values is expected

```
filter: { name_in: ["Phish", "Legion of Mary"] }
```

For the between filter two parameters are necessary. This is very useful for date ranges and number queries.

```
filter: { year_between: { from: 1966 to: 1995 } }
```

To select a distinct list of years

```
{ artist ( filter: { id:2 } ) { performance( filter: { year_distinct: true year_sort:
↪"asc" } ) { year } } }
```

All filters are AND filters. For OR support use multiple aliases queries and aggregate them. TODO: Add *orx* and *andx* support

6.2 Pagination

The filter supports *_skip* and *_limit*. There is a configuration variable to set the max limit size and anything under this limit is valid. To select a page of data set the *_skip:10 _limit:10* and increment *_skip* by the *_limit* for each request. These pagination filters exist for filtering collections too.

CHAPTER 7

Events

All events are grouped under a common **ZF\Doctrine\GraphQL\Event** object. In this repository the same event can be called in different places based on context such as when building an EntityType and when building the filters for an EntityType; both places need the same type override. That is why all events are grouped.

7.1 Filtering Query Builders

Each top level entity to query uses a QueryBuilder object. This QueryBuilder object should be modified to filter the data for the logged in user. This is the security layer. QueryBuilders are built then triggered through an event. Listen to this event and modify the passed QueryBuilder to apply your security. The queryBuilder already has the entityClassName assigned to fetch with the alias 'row'.

```
use Zend\EventManager\Event as ZendEvent;
use ZF\Doctrine\GraphQL\Event;

$events = $container->get('SharedEventManager');

$events->attach(
    Event::class,
    Event::FILTER_QUERY_BUILDER,
    function(ZendEvent $event)
    {
        switch ($event->getParam('entityClassName')) {
            case 'Db\Entity\Performance':
                // Modify the queryBuilder for your needs
                $event->getParam('queryBuilder')
                    ->andWhere('row.id = 1')
                    ;
                break;
            default:
                break;
        }
    },
```

(continues on next page)

(continued from previous page)

```
    100
);
```

7.2 Resolve

The **Event::RESOLVE** event includes the **parameters** and allows you to override the whole ResolveLoader event. This allows you to have custom parameters and act on them through the ResolveLoader RESOLVE event.

7.3 Resolve Post

The **Event::RESOLVE_POST** event allows you to modify the values returned from the ResolveLoader via an ArrayObject or replace the values.

7.4 Override GraphQL Type

The **Event::MAP_FIELD_TYPE** event allows you to override the GraphQL type for any field. Imagine you have an **array** field on an entity and the array field is multi-dimentional. Because this module handles arrays as arrays of strings (because GraphQL needs to know exact subtypes of types) it cannot handle a multi-dimentional array. A good solution is to turn the value into JSON in a hydrator strategy and override the type to a String.

```
use Zend\EventManager\Event as ZendEvent;
use GraphQL\Type\Definition\Type;
use ZF\Doctrine\GraphQL\Event;

$events = $container->get('SharedEventManager');

$events->attach(
    Event::class,
    Event::MAP_FIELD_TYPE,
    function(ZendEvent $event)
    {
        $hydratorAlias = $event->getParam('hydratorAlias');
        $fieldName = $event->getParam('fieldName');

        if ($hydratorAlias == 'ZF\\Doctrine\\GraphQL\\Hydrator\\DbTest_Entity_Artist
↪') {
            if ($fieldName === 'arrayField') {
                $event->stopPropagation();

                return Type::string();
            }
        }
    },
    100
);
```

CHAPTER 8

Custom Mapping Types

Doctrine allows Custom Mapping Types

You must create a custom GraphQL type for the field for handling serialization, etc. See **ZF\Doctrine\GraphQL\Type\DateTimeType** for an example of a custom GraphQL type.

Add the new custom GraphQL type to your configuration:

```
'zf-doctrine-graphql-type' => [
    'invokables' => [
        'datetime_microsecond' => Types\GraphQL\DateTimeMicrosecondType::class,
    ],
],
```

CHAPTER 9

GraphiQL and Documentation

Support for GraphiQL was added in v1.0.5 along with support for introspection queries.

The documentation for each field is created with a DocumentationProvider. Included with zf-doctrine-graphql is an ApigilityDocumentationProvider. If you have need for another form of documentation provider please create an issue on github. The more included providers the merrier.

CHAPTER 10

Internals

10.1 Hydrator Extract Tool

All hydrator extract operations are handled through the Hydrator Extract Tool. This tool is engineered to be overridden thanks to a service manager alias. Should you find the need to add custom caching to hydrator results this is where to do it. To register a custom hydrator extract tool use

```
'aliases' => [
    'ZF\Doctrine\GraphQL\Hydrator\HydratorExtractTool' => CustomExtractTool::class,
],
```

10.2 Field Resolver

This standard part of GraphQL resolves individual fields and is where the built in caching resides. This resolver uses the Hydrator Extract Tool and returns one field value at a time. For high performance writing your own Field Resolver is an option. To register a custom field resolver use *GraphQL::setDefaultFieldResolver($fieldResolver);*

Note: Authored by API Skeletons. All rights reserved.