Searcher Documentation

Release

Krzysztof Gzocha

September 02, 2016

CHAPTER 1

Searcher

1.1 What?

Searcher is a framework-agnostic search query builder. Search queries are written using Criterias and can be run againstMySQL, MongoDB or even files. Supported PHP versions: >=5.4, 7 and HHVM. You can find searcher in two most important places:

- GitHub repository: https://github.com/krzysztof-gzocha/searcher
- Packagist: https://packagist.org/packages/krzysztof-gzocha/searcher

1.2 Why?

Did you ever seen code responsible for searching some entities basing on many different criteria? It can be quite a mess! Imagine that you have a form with 20 fields and all of them have their impact on searching conditions. It's maybe not a great idea to pass whole form to some service at let it parse everything in one place.

1.3 How?

Thanks to this library you can split the responsibility of building query criteria to several smaller classes. One class per filter. One **CriteriaBuilder** per **Criteria**. In this way inside **CriteriaBuilder** you care only for one **Criteria**, which makes it a lot more readable. You can later use exactly the same **Criteria** for different search, with different **CriteriaBuilder** and even different **SearchingContext** which can use even different database.

CHAPTER 2

Installation

The best and simplest way to install this library is via Composer. To download and install Composer on your system please follow these instructions.

If you have already installed composer on your machine, then you can install Searcher library by typing this into your terminal:

```
$ composer require krzysztof-gzocha/searcher
```

or if you have just downloaded composer.phar to the same folder as your application:

```
$ php composer.phar require krzysztof-gzocha/searcher
```

After proper installation you should be able to find below text in your's composer.json file:

```
"require":{
    /** some libraries **/

    "krzysztof-gzocha/searcher": "^3.0.0"
}
```

2.1 Installation on production

Searcher library has configured `.gitattributes` file, so whenever you will install it via command:

```
$ composer install --prefer-dist
```

it will exclude files and folders that are not required in production environment (like docs/, tests/, etc). You can read more about this command in here and here.

2.2 Troubleshooting

Searcher has just one requirement (PHP language version >=5.4), but it has several development requirements, which can require some PHP extensions, like `ext-mongo`. If you do not have this extension installed on your system, but you still want to test this library without installing it you can use flag `--ignore-platform-reqs` to tell composer that it should not check for PHP extensions on your system. Whole installation command in this case will look like this:

```
$ composer require krzysztof-gzocha/searcher --ingore-platform-reqs
```

You can read more about `composer require` on composer pages.

CHAPTER 3

Criteria

3.1 Idea

In simplest words `Criteria` classes will hold all the parameters and their values that can be used with `CriteriaBuilder` class(s) in searching process. You have to be aware that `Criteria` can be used with multiple `CriteriaBuilder`, which means that it can be used for searches in MySQL, MongoDB, SQLite or whatever context you will use (I will describe contexts later) and that's why we we are not talking about any specific context. Instead we will use *abstract database* Of course you can use multiple `Criteria` within single searching process. Any class that implements `\KGzocha\Searcher\Criteria\CriteriaInterface` can be used as `Criteria`. There is only one method inside this interface that is required to be implemented and it is `shouldBeApplied()`. The name of the method should speak for it self - if it will return true then the criteria will be used in searching process. Of course if there will be at least one `CriteriaBuilder` that will handle it, but I will describe builders later on.

3.2 Example

Single query criteria can have zero or more fields that can be included in searching process. Let's say for example that we want to search in our *abstract database* for a particular person by his specific age. In order to to do you can create very simple class:

```
use \KGzocha\Searcher\Criteria\CriteriaInterface;

class SpecificAgeCriteria implements CriteriaInterface
{
    private $age;

    public function getAge()
    {
        return $this->age;
    }

    public function setAge($age)
    {
        $this->age = $age;
    }

    /**
    * Only required method.
    * If will return true, then it will be passed to some of the CriteriaBuilder(s)
    */
    public function shouldBeApplied()
```

```
    {
        return null !== $this->age;
    }
}
```

As you can see this is very small and simple class holding just one field, its getter and setter and `shouldBeApplied()` method. In this example we want *age* criteria to be used only if age field inside is specified, so we need to check if `null !== $this->age` inside `shouldBeApplied()`.

3.3 Multiple fields example

Ok, we have criteria for one fields, but what if having more makes more sense? Well, nothing is gonna stop you to do it! Lets assume that you still want to query your *abstract database* of people by theirs age, but you do not want specific age, but rather age range. Nothing is simpler! We just need to create criteria with minimum and maximum age, so let's do that! We are still *filtering* people by 1 *filter*, so keeping two fields in single `Criteria` makes sense, but you should keep your `Criteria` as small as possible. It should be readable and naming used inside should be obvious.

```
use \KGzocha\Searcher\Criteria\CriteriaInterface;

class AgeRangeCriteria implements CriteriaInterface
{
    private $minimumAge;
    private $maximumAge;

    public function getMinimumAge()
    {
        return $this->minimumAge;
    }

    public function setMinimumAge($age)
    {
        $this->minimumAge = $age;
    }

    public function getMaximumAge()
    {
        return $this->maximumAge;
    }

    public function setMaximumAge($age)
    {
        $this->maximumAge = $age;
    }

    /**
    * Please notice that there is OR condition inside
    */
    public function shouldBeApplied()
    {
        return null !== $this->minimumAge || null !== $this->maximumAge;
    }
}
```

Now you can specify both minimum and maximum age of people that you want to search for. Please notice that in this example in `shouldBeApplied()` method I've used **or** condition, so this criteria will be applied even if you will

specify at least one of the fields. If there would be **and** condition then this criteria would be applied only if both of the fields would be fulfilled.

3.4 Implemented criteria

You can find and use already implemented Criteria in here. You will find there query criteria for:

- Coordinates
- DateTime
- DateTimeRange
- Integer
- IntegerRange
- Number
- OrderBy (with `MappedOrderByAdapter`)
- Pagination (with `ImmutablePaginationAdapter`)
- Text
- AlwaysAppliedCriteria

3.5 Always applied criteria

In some cases you might find `AlwaysAppliedCriteria` useful, as you might use it to trigger some `CriteriaBuilder`, which will add some very important constraints to the `QueryBuilder`. For example you might want to use it to force searcher to return entities/rows/files/documents only with specified status. In such scenario you can add `AlwaysAppliedCriteria` directly to the `CriteriaCollection` and add `CriteriaBuilder` for it - builder will always be triggered, which will make impossible for end-user to change this behaviour.

3.6 Order adapter

Imagine situation in which you have constructed query using Doctrine's ORM as query builder. Now you want to allow user to pick how he would like to get the results ordered, but in the way that will tell him nothing about the query it self. For example you would like to order your query by parameter `p.id`, but you want user to see `peopleId` instead. To do so you can use `\KGzocha\Searcher\Criteria\Adapter\MappedOrderByAdapter` and following code snippet:

```
use \KGzocha\Searcher\Criteria\Adapter\MappedOrderByAdapter;

$mappedFields = [
    'peopleId' => 'p.id',
    '<order by field>' => '<mapped field>',
    /** rest of the mapping **/
];

$criteria = new MappedOrderByAdapter(
    new OrderByCriteria('peopleId'),    // hydrated OrderBy criteria
    $mappedFields
);
```

```
$criteria->getMappedOrderBy() == 'p.id'
$criteria->getOrderBy() == 'peopleId'
```

In this way we are also ensuring that only values specified in `$mappedFields` will hit criteria builders.

> **Warning:** If OrderByCriteria will be hydrated with value that is not in the mapped fields, then `getMappedOrderBy()` will return null and `shouldBeApplied()` will return false

3.7 Pagination adapter

Often you want empower user to paginate your result and to do so you can use already implemented `PaginationCriteria`, but sometimes you would like to forbid changing of number of items per page. This feature is also already implemented and it's very easy to use.

```
use \KGzocha\Searcher\Criteria\Adapter\ImmutablePaginationAdapter'

$criteria = new ImmutablePaginationAdapter(
    new PaginationCriteria($page = 1, $itemsPerPage = 50)
);
```

With criteria constructed as above user can change only the page. There is no possibility to change number of items per page.

3.8 Too long, didn't read

What do you need to know about Criteria:

1. It can be **any** class implementing `CriteriaInterface`
2. Holds parameters and values that will be used in searching process
3. Implementation of `shouldBeApplied` can change searching behaviour
4. Can be used with multiple `CriteriaBuilder`

CHAPTER 4

Criteria builder

4.1 Idea

In searcher library `CriteriaBuilder` class is used to actually build a *part* of the searching query for some *abstract database*. To do so it requires a specific `Criteria` and some *abstract query builder*. Query criteria builders can work with multiple other `SearchingContext`, which can work with multiple libraries and databases like:

- Doctrine ORM for MySQL, MariaDB, Postgres
- Doctrine ODM for MongoDB
- ruflin/elastica for ElasticSearch
- and anything else that will came to your mind - even **files**

Only classes that are implementing `\KGzocha\Searcher\CriteriaBuilder\CriteriaBuilderInterface` can be used as a criteria builders. This means that builders need to implement 3 methods:

- `buildCriteria()` which will setup the conditions on `SearchingContext` with values taken from `Criteria`,
- `allowsCriteria()` which determines if this builders can handle specific `Criteria`,
- `supportsSearchingContext()` which obviously determines if this builder can be used with this specific `SearchingContext`.

4.2 Doctrine example

Describing `Criteria` we've used example of searching for people, lets create a builder for this example using Doctrine's ORM. In order to do it we will need to create a builder which allows our *SpecificAgeCriteria* and will support Doctrine's searching context which is `\KGzocha\Searcher\Context\Doctrine\QueryBuilderSearchingContext`.

```
use \KGzocha\Searcher\CriteriaBuilder\Doctrine\AbstractORMCriteriaBuilder;
use \KGzocha\Searcher\Criteria\CriteriaInterface;
use \KGzocha\Searcher\Context\Doctrine\QueryBuilderSearchingContext;
use \KGzocha\Searcher\Context\SearchingContextInterface;

class SpecificAgeCriteriaBuilder extends AbstractORMCriteriaBuilder
{
    public function allowsCriteria(CriteriaInterface $criteria)
    {
        return $criteria instanceof SpecificAgeCriteria;
```

```
    }

    /**
     * @param SpecificAgeCriteria $criteria
     * @param QueryBuilderSearchingContext $searchingContext
     */
    public function buildCriteria(
        CriteriaInterface $criteria,
        SearchingContextInterface $searchingContext
    ) {
        $searchingContext
            ->getQueryBuilder()
            ->andWhere('p.age = :age')  // use andWhere, not where
            ->setParameter(
                'age',
                $criteria->getAge()
            );
    }
}
```

That's it! You can see new classes are used in here like `AbstractORMCriteriaBuilder` or `QueryBuilderSearchingContext`, but don't worry those are very simple classes which are already implemented and should only help you to start using this library.

- `AbstractORMCriteriaBuilder` - abstract CriteriaBuilder class which will allow only `QueryBuilderSearchingContext` to be used (You can see there is no `supportSearchingContext` method).
- `QueryBuilderSearchingContext` - searching context which will work only with Doctrine's ORM QueryBuilder

Whats the most important for us are the methods `allowsCriteria()` and of course `buildCriteria()`. In `allowsCriteria()` we have to just specify that we want only care about *SpecificAgeCriteria*. The actual building of the query is taking place on `buildCriteria()`. What is going on there?

- We are fetching Doctrine's QueryBuilder by `$searchingContext->getQueryBuilder()`,
- we are adding *and* condition and setting parameter,
- we are specifying value of this parameter with value taken from `Criteria`.

The most **important part** in here is to use **andWhere** instead of **where**, why? Because there might be another `CriteriaBuilder` before and using **where** would might have overwrite it's logic. It's really important for you to always think about single `CriteriaBuilder` as a part of complete query. You should always work only on your part - you don't want to mess up logic from different `CriteriaBuilder`.

4.3 Troubleshooting

You might experience problems when trying to declare the same *join* in two separate criteria builders. In such scenarios you have to try to not perform second join if it's already there.

4.4 Too long, didn't read

What do you need to know about CriteriaBuilder:

1. It can be **any** class implementing `CriteriaBuilderInterface`

2. Build query using SearchingContext's QueryBuilder and values from Criteria
3. You should be careful with constructing queries and not overwrite logic from different builder
4. Always think about it like a single part from a massive and complex query

CHAPTER 5

Collections

The idea of having collections is very simple. We just want to keep our criteria and its builders in single place and since array assignment in PHP always involves value copying it is better to keep it in an object. Of course the object it self holds an array inside, but now we can pass our collection by a reference, not by copying its value.

5.1 CriteriaCollection

In searcher library we have two collections responsible for holding a collection of `Criteria`. They might have different ways of adding criteria, but you can fetch all of the criteria it holds by simply calling `getCriteria()` method, which will return an array of all the criteria. If you want to extend capabilities of those classes, then you just need to implement your own class and make sure that it will implement at least one of the interfaces:

- `\KGzocha\Searcher\Criteria\Collection\CriteriaCollectionInterface` for *Regular collection*
- or `\KGzocha\Searcher\Criteria\Collection\NamedCriteriaCollectionInterface` for *Named collection*.

Both of the collections are implementing `getApplicableCriteria()` method, which you might find useful, when you will need only the criteria, which are applicable. This method will return an array of criteria that are returning `true` in `shouldBeApplied()` method.

5.1.1 Regular collection

One `\KGzocha\Searcher\Criteria\Collection\CriteriaCollection` which just holds the criteria as values of the array and it does not care about the keys. You can pass your criteria to it in two ways:

By constructor (I encourage you to use this method), which will accept array or any object implementing `\Traversable`, like:

```
$myArrayOfCriteria = [];
$collection = new CriteriaCollection($myArrayOfCriteria);
```

or by adding criteria one-by-one using `addCriteria()` method, like:

```
$myCriteria = /** Whatever implementing CriteriaInterface **/
$collection = new CriteriaCollection();

$collection->addCriteria($myCriteria);
```

5.1.2 Named collection

The other type of of collection is just extending the first one with possibility of adding a name to every criteria. I've created this class to help to hydrate your criteria. It is easier to fetch some parameter from the URI or POST data and map it's value to some model when those models have names. You can find this class in `\KGzocha\Searcher\Criteria\Collection\NamedCriteriaCollection`. It has exactly the behaviour for unnamed criteria as the first collection, so you can use methods described in *Regular collection* to add criteria, but it also allows you to do more. You can add new criteria with their names also in two ways:

By using `addNamedCriteria()` (which I encourage you to use), like:

```
$myCriteria = /** Whatever implementing CriteriaInterface **/
$collection = new NamedCriteriaCollection();

$collection->addNamedCriteria('name-of-the-criteria', $myCriteria);
```

or by using magic fields (which I personally do not like), like this:

```
$myCriteria = /** Whatever implementing CriteriaInterface **/
$collection = new NamedCriteriaCollection();

$collection->nameOfTherCriteria = $myCriteria;
```

Warning: In second approach you need to be aware that you can not use character that PHP will see as a logic (like "-", "+", "=",..).

Regardless how you add your criteria you can fetch them via `getNamedCriteria()` method, like:

```
$myCriteria = /** Whatever implementing CriteriaInterface **/
$collection = new NamedCriteriaCollection();

$result = $collection->getNamedCriteria('name-of-the-criteria');
```

If there will be a criteria assigned to name `name-of-the-criteria` then it will be returned. If not this method will return just null.

5.2 CriteriaBuilderCollection

Collection for `CriteriaBuilder` is easier than for `Criteria`, because there is only one in the library. There is no *Named collection* for `CriteriaBuilder`, but of course if you need it you can simply implement it. You just need to use `\KGzocha\Searcher\CriteriaBuilder\Collection\CriteriaBuilderCollectionInterface` as interface. You are able to add new builders in two ways:

By constructor (I encourage you to use this method) by passing an array or any `\Traversable` object with builders:

```
$builders = /** \Traversable|array of builders */
$collection = new CriteriaBuilderCollection($builders);
```

or by adding builders one by one with `addCriteriaBuilder()` method, like:

```
$builder = /** Whatever implement CriteriaBuilderInterface **/
$collection = new CriteriaBuilderCollection();

$collection->addCriteriaBuilder($builder);
```

Regardless the method you will use for adding a builders you can fetch them with `getCriteriaBuilders()`, which will return an array of all the builders.

There is also one method that might be useful when you want to retrieve all the builders that are supporting specific `SearchingContext`. Let's look on example code:

```
$searchingContext = new QueryBuilderSearchingContext();      // Some Doctrine's SearchingContext
$builder = /** Builder which support only QueryBuilderSearchingContext **/
$collection = new CriteriaBuilderCollection();
$collection->addCriteriaBuilder($builder);

$builders = $collection->getCriteriaBuildersForContext($searchingContext);
```

Now in `$builders` array we will have `$builder` object, because it is supporting specified SearchingContext.

CHAPTER 6

Searching Context

This service is used to provide user-defined *query builder* to criteria builders within searching process. At the beginning it might not be obvious what *query builder* really is, but it's really simple: it everything you want it to be and what can be used to actually build some interesting query! You can use `\Doctrine\ORM\QueryBuilder` to build query for SQL databases or whatever that suits you. The important part that you need to know as a developer is that you will have access to this from criteria builders and criteria builders will use it to actually build the query.

You can even use searching context that will work with Symfony's Finder component **to search for FILES, not records/documents in database**!

There are a few already implemented contexts:

- **Doctrine**
  - `\KGzocha\Searcher\Context\Doctrine\ODMBuilderSearchingContext` for **ODM**
  - `\KGzocha\Searcher\Context\Doctrine\QueryBuilderSearchingContext` for **ORM**
- `\KGzocha\Searcher\Context\Elastica\QuerySearchingContext` for **ElasticSearch**
- `\KGzocha\Searcher\Context\FinderSearchingContext` for **files or directories**

If you can not find useful context, then you can always implement one - it's very easy. Your's new searching context service just need to implement `\KGzocha\Searcher\Context\SearchingContextInterface`. Only two public methods are required: `getQueryBuilder()` for allowing criteria builders fetch yours *QueryBuilder* and `getResults()` for allowing searcher to fetch the results after the query is constructed.

Note: No one will tell you what `getResults()` should return, so it can be a number, array, collection or whatever. This is giving you great power, but with great power comes great responsibility. Please take care of your results and make sure you will always return what you really expect.

6.1 Symfony/Finder example

As an example this is source code of searching context with symfony/finder as query builder. It will allow all criteria builders (that are supporting this context) to use Finder and construct very complex query!

```
use \KGzocha\Searcher\Context\SearchingContextInterface;
use Symfony\Component\Finder\Finder;

class FinderSearchingContext implements SearchingContextInterface
```

```
{
    /**
    * @var Finder
    */
    private $finder;

    /**
    * @param $finder Finder
    */
    public function __construct(Finder $finder)
    {
        $this->finder = $finder;
    }

    /**
    * @return Finder
    */
    public function getQueryBuilder()
    {
        return $this->finder;
    }

    /**
    * @return Iterator
    */
    public function getResults()
    {
        // I assumed that you want Iterator as a result
        return $this->finder->getIterator();
    }
}
```

Now you can simply instantiate it as follows:

```
$context = new FinderSearchingContext(new Finder());
```

That's it! Now we can use it with `Searcher`.

CHAPTER 7

Searcher

Last step for searching process is to instantiate searcher service, perform searching and fetch the results. To properly instantiate `\KGzocha\Searcher\Searcher` class you need searching context and collection of criteria builders

```
use \KGzocha\Searcher\Searcher;
use \KGzocha\Searcher\Context\FinderSearchingContext;
use \KGzocha\Searcher\CriteriaBuilder\Collection\CriteriaBuilderCollection;
use \Symfony\Component\Finder\Finder;

// Just example context and builder collection
$context = new FinderSearchingContext(new Finder());
$criteriaBuilders = new CriteriaBuilderCollection();

$searcher = new Searcher($criteriaBuilders, $context);
```

Of course above code is just an example with usage of Symfony's Finder component, but you can use different context and collection. Only requirement is that both of them needs to implement proper interfaces.

7.1 Fetch results

When `Searcher` service is ready you can ask for results by calling `search()` method with collection of criteria as parameter. This method will return results provided by `SearchingContextInterface::getResults()`, so if your context will return an integer, searcher will also return integer.

```
use \KGzocha\Searcher\Criteria\Collection\CriteriaCollection;

$results = $searcher->search(new CriteriaCollection());    // just dummy, empty collection
```

Warning: Please, pay attention what is returned from searching context.

If you are afraid that your SearchingContext and QueryBuilder might return `null` when you are expecting `array` or `\Traversable` object, then you can use a wrapper that will handle this kind of situations for you. Using `\KGzocha\Searcher\WrappedResultsSearcher` will always return `\KGzocha\Searcher\Result\ResultCollection` on each `search()` and `ResultCollection` will accept the results only if they are traversable, so if your context will return `null` the collection will be just empty. Rest of the Searcher behaviour will remain unchanged. Code snippet below is showing how to use it:

```
use \KGzocha\Searcher\Searcher;
use \KGzocha\Searcher\WrappedResultsSearcher;
use \KGzocha\Searcher\Result\ResultCollection;
```

```php
$searcher = new WrappedResultsSearcher(new Searcher($builders, $context));

/** @var $results ResultCollection **/
$results = $searcher->search($criteriaCollection);

// Even if $context->getResults() will return null it will not break
foreach ($results as $result) {
    var_dump($result);
}

// This will also work
foreach ($results->getResults() as $result) {
    var_dump($result);
}
```

That's all about `Searcher` service. Check out examples and framework integrations for more.

CHAPTER 8

Chain searching

8.1 What is it?

Chain searching is a searching process which is divided into two or more sub-searches, resulting in the first sub-search being passed to next as a criteria, and so on. The end result of this process is an aggregated collection of the results from all the sub-searches in the chain.

Each sub-search is represented by separate `\KGzocha\Searcher\Chain\CellInterface` instance, which encapsulates the searcher and transformer instances, which internally are used to perform sub-searches. Therefore, cells are independent from each other and can hold searchers with different searching contexts. This allows one to perform a first query on one database, followed by another one on a different database and few more even on files, without problems.

8.2 Transformer

Transformer is a service that performs the transformation from the results of some sub-search into a `CriteriaCollectionInterface`, that will then be used in the next sub-search. Optionally, a transformer can also implement the `skip()` method, which returns a boolean that, if `true`, will tell the chain to skip this sub-search and move to the next one.

There is also one specific transformer `\KGzocha\Searcher\Chain\EndTransformer`, which is basically null object that should be injected into the last `Cell` to force the end of the transformations.

8.3 Example

First of all we need at least two cells. Any lower number will trigger an exception - there is no point of chaining only 1 (or zero) searchers, so let's assume we have two configured searchers, with different searching contexts. Let's also assume that we need to fetch users by some criteria with the first query (and context), and then search for some statistics for them in the second query (and second context):

```
$userSearcher = $this->getFirstSearcher();          // Will search for users
$statisticSearcher = $this->getSecondSearcher();    // Will search for statistics
```

and we have our first `CriteriaCollection` taken from the end-user:

```
$entryCriteria = new CriteriaCollection([/** criteria $userSearcher **/]);
```

We know that the second searcher will expect, for example, a criteria with an array of user ids, so we need to create a transformer that will transform the results from `$userSearcher` into a `CriteriaCollection`, that will then be injected into the `$statisticSearcher`:

```
class Transformer implements \KGzocha\Searcher\Chain\TransformerInterface
{
    /**
     * @param mixed $results
     */
    public function transform($results, CriteriaCollectionInterface $criteria)
    {
        // Assuming that UserIdsCriteria will holds an array of user IDs
        $userIdsCriteria = new UserIdsCriteria(array_map(
            function ($user) {
                return $user->getId();
            },
            $results
        ));

        // We can use some criteria from previous search, but in this scenario we do not need them.

        return new CriteriaCollection($userIdsCriteria);
    }

    /**
     * We do not want to skip this results
     *
     * @param mixed $results
     * @return bool
     */
    public function skip($results)
    {
        return false;
    }
}
```

In this transformer you can see that we are getting all user ids from the `$results`, populating `UserIdsCriteria` with those and returning a `CriteriaCollection`. Pretty simple stuff.

Now we are ready and we can create an instance of `ChainSearch` and populate it with our two cells, like this:

```
$cells = [
    new Cell(
        $userSearcher,
        new Transformer(),
        'users'            // Just an optional name
    ),
    new Cell(
        $statisticSearcher,
        new EndTransformer(),   // We don't want to go further
        'statistics'
    ),
];

$chainSearch = new ChainSearch($cells);
$results = $chainSearch->search($entryCriteria);
```

Now, the variable `$results` will hold a `ResultCollection` with two elements:

```
$results->getResults() => [
    'users' => [/** results from $userSearcher **/],
    'statistics' => [/** results from $statisticSearcher **/],
]
```

CHAPTER 9

Complete example

Below code snippet is theoretical example of searching **things** using Doctrine's ORM QueryBuilder. We are gonna to search for a things that have parameter `height` exactly the same as the one specified in the criteria. We do not care about the pagination, nor order of the results.

```php
use \KGzocha\Searcher\Criteria\CriteriaInterface;
use \KGzocha\Searcher\CriteriaBuilder\CriteriaBuilderInterface;
use \KGzocha\Searcher\Context\Doctrine\QueryBuilderSearchingContext;
use \KGzocha\Searcher\Context\SearchingContextInterface;
use \KGzocha\Searcher\CriteriaBuilder\Collection\CriteriaBuilderCollection;
use \KGzocha\Searcher\Criteria\Collection\CriteriaCollection;
use \KGzocha\Searcher\Searcher;

class HeightCriteria implements CriteriaInterface
{
    /**
     * @var float height in meters
     */
    private $height;

    /**
     * @param null|float $height
     */
    public function __construct($height = null)
    {
        $this->height = $height;
    }

    /**
     * @return float
     */
    public function getHeight()
    {
        return $this->height;
    }

    /**
     * @param float $height
     */
    public function setHeight($height)
    {
        $this->height = (float) $height;
    }
```

```
    /**
     * @inheritDoc
     */
    public function shouldBeApplied()
    {
        return $this->height != null;
    }
}

class HeightCriteriaBuilder implements CriteriaBuilderInterface
{
    /**
     * @param HeightCriteria               $criteria
     * @param QueryBuilderSearchingContext $searchingContext
     */
    public function buildCriteria(
        CriteriaInterface $criteria,
        SearchingContextInterface $searchingContext
    ) {
        $searchingContext
            ->getQueryBuilder()
            ->andWhere('t.height = :number')
            ->setParameter('number', $criteria->getHeight());
    }

    /**
     * @inheritDoc
     */
    public function allowsCriteria(CriteriaInterface $criteria)
    {
        return $criteria instanceof HeightCriteria;
    }

    /**
     * @inheritDoc
     */
    public function supportsSearchingContext(
        SearchingContextInterface $searchingContext
    ) {
        return $searchingContext instanceof QueryBuilderSearchingContext;
    }
}

/** @var \Doctrine\ORM\QueryBuilder $queryBuilder */
$queryBuilder = /** lets assume its already created QueryBuilder */null;

$criteria = new HeightCriteria(200); // hydrated criteria
$criteriaBuilder = new HeightCriteriaBuilder();
$context = new QueryBuilderSearchingContext($queryBuilder);

$searcher = new Searcher(
    new CriteriaBuilderCollection([$criteriaBuilder]),
    $context
);

$results = $searcher->search(new CriteriaCollection([$criteria]));

foreach ($results as $result) {
```

```
    var_dump($result);
}
```

CHAPTER 10

Integration with Symfony

Currently searcher library can be easily connected to any application written in Symfony framework version 2 or higher. This documentation will guide you through installation and usage of the library inside Symfony application. Knowledge of the library itself is **strongly recommended** before starting to integrate with any framework.

Integration is done through the separate *bundle* - **SearcherBundle** - that can be easily added to existing app. You can find it in:

- GitHub: https://github.com/krzysztof-gzocha/searcher-bundle
- Packagist: https://packagist.org/packages/krzysztof-gzocha/searcher-bundle

10.1 Installation

The easiest way to install this bundle is through usage of composer - the same way as the library. In order to do it you need just to type in your favorite terminal:

```
$ composer require krzysztof-gzocha/searcher-bundle
```

Note: SearcherBundle already has specified searcher library as requirement, so you do not need to require library itself. Bundle will be enough.

Next step is to let Symfony know about it by adding it to registered bundles in `AppKernel.php` file like this:

```
public function registerBundles()
{
  $bundles = array(
    /** Your bundles **/
    new \KGzocha\Bundle\SearcherBundle\KGzochaSearcherBundle(),
  );
  /** rest of the code  **/
}
```

Now Symfony will be aware of searcher, but if we want to fully use it we would need to configure it, which is also very simple if you know how the library works.

10.2 Basic configuration

I recommend to start with creation of new file, called `searcher.yml` in which we will put all our searcher specific configuration. Then we should import this file in global `config.yml` file by adding this code at the top of it:

```
imports:
    - { resource: searcher.yml }
```

Note: Of course you do not need to create seperate file for searcher configuration. I just find this way prettier, but you can use whatever way that is better for you.

Now in `searcher.yml` we can start to develop our basic configuration. Configuration below is basic working example that we will describe in details.

```
k_gzocha_searcher:
  contexts:
    people:
      context:
        service: your_searching_context_service_id

      criteria:
        - { class: \AgeRangeCriteria, name: age_range}
        - { service: some_service_id, name: other_criteria }

      builders:
        - { class: \AgeRangeCriteriaBuilder, name: age_range }
        - { service: other_service_id, name: my_criteria_builder }
```

First line `k_gzocha_searcher` is just a statement used to specify that everything below is the actual config of searcher. Fun starts after second line `contexts`, which describes a fact that below this statement SearcherContexts are specified. Now, line `people` doesn't sound like connected to the library or bundle and it shouldn't, because it is a name of the searching context, that we will use. In next point we have to specify name of the service that implements SearchingContextInterface. Unfortunately you have to create this service on your own and bundle will not help you with that... yet. Going further.. In next step we need to specify our criteria, that might be used during searching process. In this example we have two of them: `age_range` which is a simple class `\AgeRangeCriteria` and `other_criteria` which is already existing service with a name `some_service_id`.

Warning: Please remember that we are describing criteria with their class name or service name - never both. If both parameter will be provided, then class parameter will be omitted. Service parameter have higher priority. This rule is the same for every configuration point in this bundle.

In the last step we are configuring CriteriaBuilders that might be used by searcher and again we have two of them: `age_range` described as a class `\AgeRangeCriteriaBuilder` and `my_criteria_builder` described as already existing service with a name `other_service_id`.

Configuration created in this way will create services for every searcher class. Those services will be accessible for you. Here is a list of them:

- Searcher: `k_gzocha_searcher.people.searcher`
- Context: `k_gzocha_searcher.people.context`
- Criteria "age_range": `k_gzocha_searcher.people.criteria.age_range`
- Criteria "other_criteria": `k_gzocha_searcher.people.criteria.other_criteria`
- Builder "age_range": `k_gzocha_searcher.people.builder.age_range`

- Builder "my_criteria_builder": `k_gzocha_searcher.people.builder.my_criteria_builder`
- Criteria collection: `k_gzocha_searcher.people.criteria_collection` (Named collection is used by default)
- Builder collection: `k_gzocha_searcher.people.builder_collection`

You can found complete configuration reference in here.

10.3 Hydration

In pure searcher library there is nothing mentioned about how you can fetch values from user of your application and pass them to corresponding criteria, but in Symfony we have very powerful tool to do it properly - forms! Let's assume we have our `\AgeRangeCriteria` configured with name `age_range` and let's assume that it has two fields `minimalAge` and `maximalAge`. Now we can build a form, that will help us hydrate this criteria:

```
use KGzocha\Bundle\SearcherBundle\Form\SearchForm;

class MySearchForm extends SearchForm
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('minimalAge', 'integer', [
                'property_path' => $this->getPath('ageRange', 'minimalAge'),
            ])
            ->add('maximalAge', 'integer', [
                'property_path' => $this->getPath('ageRange', 'maximalAge'),
            ])
            /** and any other fields.. **/
            ->add('<parameter name in request>', '<form type>', [
                'property_path' => $this->getPath(
                    '<criteria name from config>',
                    '<criteria field name inside the class>'
                ),
            ]);
    }
}
```

10.4 Example action

Assuming that we have installed bundle, configured searcher and created form we can perform our first search by creating simple action inside a controller:

```
public function searchAction(Request $request)
{
    $form = $this->createForm(
        new MySearchForm(),
        $this->get('k_gzocha_searcher.people.criteria_collection')
    );

    $form->handleRequest($request);

    if ($form->isValid()) {
        $searcher = $this->get('k_gzocha_searcher.people.searcher');
```

```
        $results = $searcher->search($form->getData());
        // Yay, we have our results!
    }

    /** rest of the code **/
}
```

Warning: By default Searcher is wrapped with WrappedResultsSearcher, so results will be actually an instance of ResultCollection. If you would like to have pure Searcher then you have to specify searcher.wrapper_class in the config as null or create searcher service yourself and specify searcher.service.