RollerworksSearch Documentation

Release 1.0.0-beta5

Sebastiaan Stok

January 26, 2017

Contents

CHAPTER 1

Introduction

RollerworksSearch provides you with a powerful search system for your PHP applications.

Search operations are performed using search conditions which allow for any type of condition, all with minimum effort, whether you need to search in one database table or multiple.

This system was designed to be useful for any storage back-end, user input and locale.

1.1 Organization of this Book

This book has been written so that those who need information quickly are able to find what they need, and those who wish to learn more advanced topics can read deeper into each chapter.

The book begins with an overview of the Search system, discussing what's included in the package and preparing you for the remainder of the book.

It is possible to read this user guide just like any other book (from beginning to end). Each chapter begins with a discussion of the contents it contains, followed by a short code sample designed to give you a head start. As you get further into a chapter you will learn more about component's capabilities, but often you will be able to head directly to the topic you wish to learn about.

Throughout this book you will be presented with code samples, which most people should find ample to implement the Search system appropriately in their own projects.

1.2 Code Samples

Code samples presented in this book will be displayed on a different colored background in a mono-spaced font. Samples are not to be taken as copy and paste code snippets.

Code examples are used throughout the book to clarify what is written in text. They will sometimes be usable as-is, but they should always be taken as outline/pseudo code only.

A code sample will look like this:

```
class AClass
{
    ...
}

// A Comment
$obj = new AClass($arg1, $arg2, ... );
```

```
/* A note about another way of doing something
$obj = AClass::newInstance($arg1, $arg2, ... );
*/
```

The presence of 3 dots `...` in a code sample indicates that code has been excluded, for brevity. They are not actually part of the code.

Multi-line comments are displayed as `/* ... */` and show alternative ways of achieving the same result.

You should read the code examples given and try to understand them. They are kept concise so that you are not overwhelmed with information.

CHAPTER 2

Components Overview

Most features for searching are provided by the library using object-oriented PHP code as the interface.

In this chapter we will take a short tour of the various components, which put together form RollerworksSearch as a whole. You will learn key terminology used throughout the rest of this book and will gain an understanding of the classes you will work with as you integrate RollerworksSearch into your application.

This chapter is intended to prepare you for the information contained in the subsequent chapters of this book.

2.1 Information flow

In most cases you accept and process the input, optimize it and then pass it to a condition processor in the search storage layer. But its also possible to construct the search-condition yourself, and pass it directly to the condition processor without any optimizing.

Note the following:

- Transforming view values to a normalized version is done when processing the input.
- The optimizing process tries to produce the smallest search-condition possible, but is only able to do this when the system is properly configured.

The system is mainly concerned with the SearchCondition and configuration of the search fields.

2.2 System Requirements

The basic requirements to use RollerworksSearch are:

- PHP 5.3.3 or higher, with the SPL extension (standard)
- Multibyte string extension, for multibyte text handling.
- International support for transforming date-time values.

And a list 3rd party libraries (which you can find the installation chapter).

Tip: When you use Composer to install and update dependencies the installation of these libraries will be handled for your.

2.3 Component Breakdown

RollerworksSearch is made up of many classes. Each of these classes can be grouped into a general "component group" which describes the task it is designed to perform.

We'll take a brief look at the components which form RollerworksSearch as a whole, in this section of the book.

2.3.1 SearchCondition

Each search operation starts with a SearchCondition (`SearchConditionInterface`). A SearchCondition defines a set of requirements (conditions) for one or more fields. And holds the configuration of these fields within in a `FieldSet`.

A field (search field) can be compared to a form field or database column.

```
use Rollerworks\Component\Search\FieldSet;
use Rollerworks\Component\Search\ValuesGroup;
use Rollerworks\Component\Search\ValuesBag;
use Rollerworks\Component\Search\Value\SingleValue;

$fieldSet = new FieldSet('my_field_set');
// FieldSet configuration...

$rootValuesGroup = new ValuesGroup();

$fieldId = new ValuesBag();
$fieldId->addSingleValue(new SingleValue(10));
$fieldId->addSingleValue(new SingleValue(20));
$rootValuesGroup->addField('id', $fieldId);

$fieldDate = new ValuesBag();
$fieldDate->addSingleValue(
    new SingleValue(
        new \DateTime('2015-02-04 00:00:00', new \DateTimezone('UTC')) // normalized value
        '2015/04/24' // View in US date notation
    )
);

$subValuesGroup = new ValuesGroup();

$fieldId = new ValuesBag();
$fieldId->addSingleValue(new SingleValue(10));
$fieldId->addSingleValue(new SingleValue(20));

$subValuesGroup->addField('id', $fieldId);
$rootValuesGroup->addGroup($subValuesGroup);

$searchCondition = new SearchCondition($fieldSet, $rootValuesGroup);
```

At the root of each SearchCondition is a `ValuesGroup`, containing the values (as `ValuesBag`) per field name and *optionally* subgroups (each one being a `ValuesGroup`).

A `ValuesGroup` is defined with a `ValuesGroup::GROUP_LOGICAL_AND` by default which requires from each field inside a (sub)group that at least one of the field's values evaluate to true (is matching).

A record is intended as a database record with one or fields. A single User with an id, registration-date and username is considered one record.

Taken you will only get a result when the id is e.g. 10 or 20 **and** the date is "2015-02-04". If date is anything else then "2015-02-04" the record is not matching.

But it's also possible to set a group with `ValuesGroup::GROUP_LOGICAL_OR`, which removes the requirement that *all** fields must match. If any of the fields matches the record is considered matching.

Note: Subgroups function as a list of fields with `ValuesGroup::GROUP_LOGICAL_AND` even when the parent group is set with `ValuesGroup::GROUP_LOGICAL_OR`.

Only when fields within the parent group gave a positive match the subgroup will be evaluated.

When there are multiple subgroups these are OR'ed to each other, Meaning at least one (or more) subgroup(s) in the group must match.

Learn more about creating conditions at SearchConditions in action.

A `ValuesBag` holds all the values of a field per type.

Supported value-types are:

- Single value (any type of value)
- Excluded single value (any type of value which should not provide a positive match)
- Ranges (from - to, e.g. 10 - 100)
- Excluded ranges (from - to, e.g. 10 - 100 which is should not provide a positive match)
- Comparison value (mathematical comparison: <, >, >=, <=)
- PatternMatch (text based pattern matching, starts with, contains, ends with, regex), and supports excluding (e.g. not starts with) and case optional insensitive.

Values are stored in a normalized and view format. The actual transformation is handled by the DataTransformers registered on the search field configuration.

Tip: Either side of a Range value can be marked as exclusive. Meaning anything between the values except the values them self.

In practice this is the same as using `>20 AND <30`. But much easier to optimize.

Normally a `SearchCondition` is created when processing input. But you can also build the `SearchCondition` manually using the :class:`Rollerworks\\Component\\Search\\SearchConditionBuilder` see *Performing a manual search* for more information.

2.3.2 FieldSet

A :class:`Rollerworks\\Component\\Search\\FieldSet` holds the configuration of one or multiple `FieldConfigInterface` instances, each field is called a search field.

Tip: A `FieldSet` can also be created by using the `FieldSetBuilder`, which provides a much simpler interface.

Each search field works independent from a FieldSet and may be reused in multiple FieldSets. But the field's name must be unique within the FieldSet.

Normally you would create a FieldSet based on a subject-relationship. For example invoice search, order search, news items search, etc.

Note: The `FieldConfigInterface` is a public interface for your own implementation. The default implementation is a `SearchField` object.

2.3.3 SearchField

...

Property	Description	Value-type
Name	Name of the search field.	`string`
Type	An object implementing the `ResolvedFieldTypeInterface`. Provides a field type class for building the fields configuration.	`ResolvedFieldTypeInterface`
SupportValueType	Indication which value-types are accepted by the field.	`boolean`
ModelRefClass	Model's fully qualified class-name reference. This is required for certain storage engines like Doctrine ORM.	`string`
ModelRefProperty	Model's property name reference. This is used in combination with ModelRefClass	`string`
ValueComparison	ValuesComparison object used for range validating and optimizing.	`ValueComparisonInterface`
ViewTransformers	A list of transformers for transforming from view to normalized, and reverse.	`DataTransformerInterface[]`
Options	Configured options of the field. The options handled using the Type configuration.	`array`

2.3.4 Input

The input component processes user-input to a `SearchCondition`.

Input can be provided as a PHP Array, JSON, XML document, or using the FilterQuery format.

2.3.5 Exporters

While the input component processes user-input to a SearchCondition. The exporters do the opposite, transforming a SearchCondition to an exported format. Ready to be reused for input processing.

Exporting a SearchCondition is very handy if you want to store the condition on the client-side in either a cookie, URI query-parameter or hidden form input field.

Or if you need to perform a search operation on an external system that uses RollerworksSearch. Build-up your SearchCondition using the SearchConditionBuilder and export it for usage!

2.3.6 FieldAliasResolver

Sometimes you want to use a localized field-name rather then the actual field-name.

For example: "factuur-nummer" (in Dutch) for "invoice-number" (original name).

For this you can use the FieldAliasResolver (`FieldAliasResolverInterface`) which tries to resolve a field-alias to a real field-name.

RollerworksSearch comes bundled with three alias-resolvers:

- Noop: This resolver does nothing and simple returns the original input.
- Chain: This allows to chain multiple alias-resolvers, the first resolver which returns something else than the original input is considered the matching resolver.
- Array: This resolver uses a simple PHP array for keeping track of aliases.

Note: If the resolving process fails the originally provided field-name is used.

2.3.7 Condition Optimizers

Condition optimizers optimize SearchConditions, by removing duplicated values, normalizing overlapping and redundant values/conditions.

The following optimizers come already pre-bundles with RollerworksSearch.

Note: For the best result optimizers should be performed in correct order, therefore each optimizer has a priority between -10 and 10.

The `ChainOptimizer` automatically performs the optimizers in there correct order.

Name	Description	Priority
`ChainOptimizer`	Runs the registered optimizers in sequence with correct the priority.	0
`DuplicateRemover`	Removes duplicated values inside a condition group.	5
`ValuesToRange`	Converts incremented values to inclusive ranges. Example values 1,2,3,4,5 are converted to range 1-5	4
`RangeOptimizer`	Removes overlapping ranges/values and merges connected ranges.	-5

2.3.8 Field Type

Field types are used for configuring a search field's value comparison, ViewTransformers and accepted value-types.

For more information on using field types see Field Type

Note: Build-in types are provided by the Core extension.

You are free create your own field types for more advanced use-cases. See How to Create a Custom Search Field Type for more information.

2.3.9 SearchFactory

The SearchFactory forms the heart of the search system, it provides easy access to builders and keeps track of field types.

But you would rather want to use the `Searches` class which takes care of all the boilerplate of setting up a SearchFactory. See Performing searches for information and usage.

2.3.10 SearchConditionSerializer

The `SearchConditionSerializer` class functions as a helper for serializing a `SearchCondition`.

A SearchCondition holds a ValuesGroup (with nested ValuesBags and optionally other nested ValuesGroup objects). But also FieldSet.

The ValuesGroup and values can be easily serialized, but the FieldSet is a bit harder. So instead of serializing the FieldSet it stores only the FieldSet's name, and when unserializing it loads the FieldSet using the `FieldSetRegistryInterface`.

2.3.11 FieldSetRegistry

A FieldSetRegistry (`FieldSetRegistryInterface`) keeps track of all the FieldSets that you have created and registered.

The FieldSetRegistry is used when unserializing a serialized SearchCondition, so that don't have to inject the FieldSet explicitly. But you are free to use it whenever you find it useful.

CHAPTER 3

Installing the Library

Installing RollerworksSearch is trivial. Usually it's just a case of uploading the extracted source files to your web server.

3.1 Installing with Composer

Composer is a dependency management library for PHP, which you can use to download the RollerworksSearch system.

Start by downloading Composer anywhere onto your local computer. If you have curl installed, it's as easy as:

```
curl -s https://getcomposer.org/installer | php
```

Installing RollerworksSearch with Composer is as easy as:

```
$ php composer.phar require rollerworks/search
```

From the directory where your `composer.json` file is located.

Now, Composer will automatically download all required files, and install them for you. After this you can start integrating RollerworksSearch with your application.

3.1.1 Optional packages

The `rollerworks/search` package itself does not provide any mechanise for searching a storage engine (like Doctrine or ElasticSearch).

To search in a storage engine you need to install additional packages or build your own SearchCondition processor.

To get you started we already provide a number of additional packages for searching;

Doctrine

- rollerworks/search-doctrine-orm allows searching a relational SQL database using Doctrine2 ORM.
- rollerworks/search-doctrine-dbal allows searching a relational SQL database using Doctrine2 DBAL.

Metadata extension

https://github.com/rollerworks/rollerworks-search-metadata

Provides a Metadata loader using the rollerworks/metadata package.

CHAPTER 4

Performing searches

This chapter will explain how you can integrate RollerworksSearch into your application.

You should have already installed the package and have a good understanding about all the components. If not please do this before continuing.

4.1 Using the FactoryBuilder

The FactoryBuilder helps with setting up the search system. You only need to set a SearchFactory up once, and then it can be reused multiple times.

Note: The `Searches` class and SearchFactoryBuilder are only meant to be used when you using RollerworksSearch as a standalone. When making a framework plugin, you want to properly create the SearchFactory and FieldsRegistry manually using a Dependency Injection system.

```
use Rollerworks\Component\Search\Searches;

$searchFactory = new Searches::createSearchFactoryBuilder()
    // Here you can optionally add new types or (type) extensions
    ->getSearchFactory();
```

4.2 Creating a FieldSet

Now, before you can start performing searches, the system first needs a `FieldSet` which will hold the configuration of your search fields.

You can create as many FieldSets as you want, but each FieldSet needs a name that should not clash with other FieldSets. So it's best to use descriptive names like: 'customer_invoices' and 'customers'.

```
$fieldset = $searchFactory->createFieldSetBuilder()
    ->add('id', 'integer')
    ->add('name', 'text')
    ->getFieldSet();
```

Tip: You can also use the FieldSetBuilder to import the fields from models using the Metadata component.

Loading a Fields configuration using metadata reduces the duplication between FieldSets. You only configure the field once and can reuse it multiple times throughout the application.

4.3 Performing a manual search (SearchConditionBuilder)

In most cases you would ask the system to process an input and pass it to a list of condition optimizers before applying it on the storage layer. But it's also possible to create a SearchCondition manually.

The `SearchConditionBuilder` is just for this, if you already know how an XML document is build then this should be pretty straightforward.

Each time you call `group()` it will create a new `SearchConditionBuilder` with a new depth. When you call `end()` it will return to the parent builder.

Calling `field()` will get you a new `ValuesBagBuilder` which allows adding new values, and then calling `end()` to get back to the ConditionBuilder.

Note: Each value-type (except pattern-match) has a normalized value and a view value. Unless you pass a view value, the normalized value is used (as string).

When a normalized value can not be casted to a string, this will throw an error.

```
use Rollerworks\Component\Search\SearchConditionBuilder;
use Rollerworks\Component\Search\Value\Compare;
use Rollerworks\Component\Search\Value\PatternMatch;
use Rollerworks\Component\Search\Value\Range;
use Rollerworks\Component\Search\Value\SingleValue;

$searchCondition = new SearchConditionBuilder::create($fieldset)
    ->field('id')
        ->addSingleValue(new SingleValue(12))
        ->addSingleValue(new SingleValue(30))
        ->addRange(new Range(50, 60))
    ->end()
    ->getSearchCondition();
```

This example will give you a SearchCondition with exactly one SearchGroup and the following condition: id is 1 or 30 or (inclusive between 50 and 60).

Or if you need a more complex condition.

```
use Rollerworks\Component\Search\SearchConditionBuilder;
use Rollerworks\Component\Search\ValuesGroup;
use Rollerworks\Component\Search\Value\Compare;
use Rollerworks\Component\Search\Value\PatternMatch;
use Rollerworks\Component\Search\Value\Range;
use Rollerworks\Component\Search\Value\SingleValue;

$searchCondition = new SearchConditionBuilder::create($fieldset)
    ->field('id')
        ->addSingleValue(new SingleValue(12))
        ->addSingleValue(new SingleValue(30))
        ->addRange(new Range(50, 60))
    ->end()
    ->group(ValuesGroup::GROUP_LOGICAL_OR)
```

```
        ->field('id')
            ->addSingleValue(new SingleValue(12))
            ->addSingleValue(new SingleValue(30))
            ->addRange(new Range(50, 60))
        ->end()
        ->field('name')
            ->addSingleValue(new PatternMatch('rory', PatternMatch::PATTERN_STARTS_WITH))
            ->addSingleValue(new PatternMatch('amy', PatternMatch::PATTERN_STARTS_WITH))
            ->addSingleValue(new PatternMatch('williams', PatternMatch::PATTERN_ENDS_WITH))
        ->end()
    ->end()
    ->getSearchCondition();
```

Tip: When you call `field()` with an existing field, the original field is returned.

Set the second parameter to true to force a new one, note this will remove the old field!

4.4 Processing input

The most common case is processing the input to a SearchCondition, the system can process a wide range of supported formats.

This example uses the FilterQuery with the FieldSet configuration shown above.

```
use Rollerworks\Component\Search\Exception\InvalidSearchConditionException;
use Rollerworks\Component\Search\Exception\InputProcessorException;
use Rollerworks\Component\Search\ConditionOptimizer\ChainOptimizer;
use Rollerworks\Component\Search\ConditionOptimizer\DuplicateRemover;
use Rollerworks\Component\Search\ConditionOptimizer\ValuesToRange;
use Rollerworks\Component\Search\ConditionOptimizer\RangeOptimizer;
use Rollerworks\Component\Search\Input\FilterQueryInput;
use Rollerworks\Component\Search\Input\FilterQuery\QueryException;
use Rollerworks\Component\Search\Input\ProcessorConfig;
use Rollerworks\Component\Search\Searches;

$searchFactory = new Searches::createSearchFactoryBuilder()
    ->getSearchFactory();

// Each input processor is reusable.
// So its possible to use the FilterQueryInput instance multiple times.
$inputProcessor = new FilterQueryInput();

// The provided query can come from anything, like $_GET or $_POST
$query = ... ;

// The ProcessorConfig allows configuring value limits
// group nesting and maximum group count.
$config = new ProcessorConfig($fieldSet);

// The input processor will transform all values to the normalized value
// and validates range bounds are valid.

try {
    $searchCondition = $inputProcessor->process($config, $query);
} catch (InvalidSearchConditionException $e) {
```

```
    // The SearchCondition contains errors.
    // This is good moment to tell the user the condition
    // has errors which should be resolved.

    // The errors are stored on the SearchCondition.
    // See the section about handling processing errors
    // for more information on handling these.
} catch (QueryException $e) {
    // This exception is specific for the FilterQueryInput
    // and is thrown when there is a syntax error in the input.
    // The message will point exactly what is wrong with the user input
    echo $e->getMessage();
} catch (InputProcessorException $e) {
    // Generic processing error
    echo $e->getMessage();
}

// Note: processing errors is much more advanced
// than you would expect. See the next section for more information.

// Because the search condition may have duplicate or redundant
// values we run them trough a list of optimizers.

$optimizer = new ChainOptimizer();
$optimizer->addOptimizer(new DuplicateRemover());
$optimizer->addOptimizer(new ValuesToRange());
$optimizer->addOptimizer(new RangeOptimizer());
$optimizer->process($searchCondition);

// Lock the condition to prevent future changes
// This is not really required but its a good practice to this
$searchCondition->getValuesGroup()->setDataLocked();

// Now the $searchCondition is ready for applying on any supported storage engine
```

4.5 Handling processing errors

When processing input its possible the input is invalid e.g. a syntax/structure error, passing an unsupported value-type to a field or missing a required field.

To not leave these situations unnoticed each processor will throw an exception in case of an error. The exception itself provides more information on what is wrong.

Please keep note of the following:

- The field-name is the resolved field name and not the alias that was used.
- The group and nesting level start at index 0 which is the root of the condition.

Tip: All exceptions have a pre-formatted message for direct usage.

So displaying an error message is as simple as `echo $e->getMessage();`.

4.5.1 GroupsNestingException

The `Rollerworks\Component\Search\Exception\GroupsNestingException` is thrown when the maximum nesting level is exceeded.

This exception provides the following properties:

- maxNesting: Maximum nesting level
- groupIdx: index of the nested-group exceeding the maximum nesting level
- nestingLevel: the nesting level at which the group is declared

4.5.2 ValuesOverflowException

The `Rollerworks\Component\Search\Exception\ValuesOverflowException` is thrown when the maximum number of values is exceeded.

This exception provides the following properties:

- fieldName: Name of the field which has to many values
- max: Maximum number of values within a field
- count: Number of values in the field
- groupIdx: index of the group at which the field was declared
- nestingLevel: the nesting level at which the field was declared

---

Note: Not all processors will give the exact number of values.

FilterQuery will stop further processing when the maximum amount of values is exceeded. But XML, JSON and Array will return the exact number of values.

4.5.3 GroupsOverflowException

The `Rollerworks\Component\Search\Exception\GroupsOverflowException` is thrown when the maximum number of groups at a nesting level is exceeded.

This exception provides the following properties:

- max: Maximum number of subgroups within a (sub)group
- count: Number of groups in the (sub)group
- groupIdx: index of the group exceeding the maximum count
- nestingLevel: the nesting level at which the group was declared

---

Note: Not all processors will give the exact number of groups.

FilterQuery will stop further processing when the maximum amount of groups is exceeded. But XML, JSON and Array will return the exact number of values.

4.5.4 UnsupportedValueTypeException

The `Rollerworks\Component\Search\Exception\UnsupportedValueTypeException` is thrown when you pass a value-type into a field which doesn't support that value-type.

This exception provides the following properties:

- fieldName: Name of the field at which the value was declared
- valueType: Type of the value which was not accepted, e.g. range, comparison or pattern-match

4.5.5 InvalidSearchConditionException

The `Rollerworks\Component\Search\Exception\InvalidSearchConditionException` is thrown when the SearchCondition has errors.

Most of these errors are eg. failed transformation or invalid range bounds.

This exception provides access to the invalid SearchCondition using `getCondition()`. The actual search value-errors are stored within the ValuesBag of each field.

The following example shows you can render these errors into a display for the user.

```
use Rollerworks\Component\Search\ValuesGroup;
use Rollerworks\Component\Search\ValuesBag;

// ..

function displaySearchErrors(ValuesGroup $group, $nestingLevel = 0)
{
    // By default hasErrors() only checks the fields in its own group.
    // But we want to check all nested groups too! so pass true to overwrite
    // this behaviour.
    if (!$group->hasErrors(true)) {
        return; // no errors so nothing do be done
    }

    $fields = $group->getFields();

    foreach ($fields as $fieldName => $values) {
        // $errors holds an array of ValuesError objects.
        //
        // A ValuesError object actually holds some very interesting information
        // including the "cause" which tells why the error occurred.
        // And a translatable message-template and parameters
        //
        // See ``Rollerworks\Component\Search\ValuesError`` for more information.

        $errors = $values->getErrors();

        if ($values->hasErrors()) {
            echo str_repeat(' ', $nestingLevel * 2).$fieldName.' has the following errors: ';

            foreach ($errors as $valueError) {
                echo str_repeat(' ', $nestingLevel * 2).' - '.$valueError->getMessage();
            }
        }

        foreach ($group->getGroups() as $subGroup) {
```

```
            displaySearchErrors($group, ++$nestingLevel);
        }
    }
}

try {
    $searchCondition = $inputProcessor->process($config, $query);
} catch (InvalidSearchConditionException $e) {
    $group = $e->getCondition()->getValuesGroup();

    displaySearchErrors($group);
}

// Caching of other exceptions has been deliberately omitted
```

You would properly want build something that is more advanced, this is just a simple verbose example to show how you get the errors.

4.5.6 InputProcessorException

The `Rollerworks\Component\Search\Exception\InputProcessorException` is thrown when a general error is hit. This is mostly used for malformed value structures.

The Exception message tells more about what is wrong, this exception does not expose any special properties.

4.5.7 QueryException

The `Rollerworks\Component\Search\Input\FilterQuery\QueryException` is only used by the FilterQuery input processor.

This exception is thrown when the provided input has a syntax error.

Example: `[Syntax Error] line 0, col 46: Error: Expected '"(" or FieldIdentification', got ')'`

The error tells that at column 46 a group opening or field-name was expected but something else was found instead.

This exception provides the following properties:

- line: Line-number at which the error occurred
- col: Column position at which the error occurred (starting from 0)
- expected: An array of tokens that were expected
- got: A Token-id, value or character that was found instead

For clarity the following token-ids are used:

- String: a unquoted string like `foo` or `12`
- QuotedString: a quoted string like `"foo"`, `"12"` or `"12.00"`
- Range: A range with lower and upper-bounds like `12-15` or `]12-15[`
- ExcludedValue: An excluded range with lower and upper-bounds like `!12-15` or `!]12-15[`
- Comparison: Mathematical comparison like `>12`, `<15` or `>="foo-bar"`
- PatternMatch: A text based pattern matcher like `~*foo`, `~!*foo`

If the "got" or "expected" property is anything else then shown above, its a literal character. For example `'>` and `(` are literal characters.

Note: QuotedString values don't actually contain the leading and trailing quotes when processing. *The processor already normalizes these.*

This is just to indicate a QuotedString could be used at the position.

4.6 Improving performance

Most search operations consist of a search condition that is being applied on a storage engine like a database or search index.

But you properly don't want to display all 500 found records on a single page. You paginate them to display a limited subset per page. And each page uses the same search-condition.

However processing a user-input to a `SearchCondition` and optimizing it can be very slow (depending on the number of fields, values and groups). And as the condition has not changed between page requests there is no point in repeating these steps!

Fortunately SearchConditions are serializable, meaning you can export (not to be confused with the exporter component) the condition to a storage friendly format for faster loading.

The following part shows an example for storing a search-condition using the PHP session system.

```php
use Rollerworks\Component\Search\Exception\ExceptionInterface;
use Rollerworks\Component\Search\ConditionOptimizer\ChainOptimizer;
use Rollerworks\Component\Search\ConditionOptimizer\DuplicateRemover;
use Rollerworks\Component\Search\ConditionOptimizer\ValuesToRange;
use Rollerworks\Component\Search\ConditionOptimizer\RangeOptimizer;
use Rollerworks\Component\Search\Input\FilterQueryInput;
use Rollerworks\Component\Search\Input\FilterQuery\QueryException;
use Rollerworks\Component\Search\Input\ProcessorConfig;
use Rollerworks\Component\Search\FieldSetRegistry;
use Rollerworks\Component\Search\Searches;

// This example uses a PHP session, but you can actually use anything.
// Just remember to NEVER store a PHP serialized object on the client-side
// as this makes it possible to inject arbitrary code!
session_start();

$searchFactory = new Searches::createSearchFactoryBuilder()
    ->getSearchFactory();

$fieldSetRegistry = new FieldSetRegistry();

$fieldset = $searchFactory->createFieldSetBuilder()
    ->add('id', 'integer')
    ->add('name', 'text')
    ->getFieldSet();

// It's important the FieldSet is registered in the FieldSetRegistry
// before serializing. Else you will get an exception thrown.
$fieldSetRegistry->add($fieldset);

// The provided query can come from anything, like $_GET or $_POST
```

```php
$query = ... ;

if (!is_string($query)) {
    exit('Expected a string.');
}

// Use an mad5 hash to generate a unique caching-key
// md5 is the fastest hashing method and provides enough uniqueness for this situation
// normally you would use something stronger like sha1 or even sha265
$searchHash = 'search_'.md5($query);

$searchConditionSerializer = new SearchConditionSerializer($fieldSetRegistry);

if (isset($_SESSION[$searchHash])) {
    $searchCondition = $searchConditionSerializer->unserialize($_SESSION[$searchHash]);
} else {
    $inputProcessor = new FilterQueryInput();
    $config = new ProcessorConfig($fieldSet);

    try {
        $searchCondition = $inputProcessor->process($config, $query);
    } catch (ExceptionInterface $e) {
        // Note: The Rollerworks\Component\Search\Exception\ExceptionInterface
        // is implemented by all the exceptions thrown by RollerworksSearch,
        // its possible (just not likely) that these messages expose sensitive
        // information about your application. See the section about error handling
        // for a better alternative.

        echo $e->getMessage();
    }

    $optimizer = new ChainOptimizer();
    $optimizer->addOptimizer(new DuplicateRemover());
    $optimizer->addOptimizer(new ValuesToRange());
    $optimizer->addOptimizer(new RangeOptimizer());
    $optimizer->process($searchCondition);

    $searchCondition->getValuesGroup()->setDataLocked();

    // Store the condition for feature usage
    $_SESSION[$searchHash] = $searchConditionSerializer->serialize($searchCondition);
}
```

Note: This example does not cover removing a search-condition when its no longer needed. Because we use a PHP Session the cached condition is automatically removed when the session expires.

CHAPTER 5

SearchConditions in action

This chapter explains how you can use search conditions in practice, what kind of results you can expect with a search condition and handy tips for getting the best result.

These examples shown below use the FilterQuery syntax as input condition (condition for short).

Remember that almost all values of a field are OR'ed, meaning that at least one value must match for the field to have a positive match. `field: value1, value2` means from the current row the value of column field1 must be e.g. value1 or value2;

Note: Excluded values are not use OR'ed, so `field: !value1, !value1;` will only match if field1 is not value1 and not value2;

5.1 Common mistakes and good to know

5.1.1 Comparison

Comparisons are applied as AND, meaning all of them must give a positive match. So using multiple comparisons may not give the expected result. `field: >10, <20` is the same as using a range like `field: ]10-20[`.

But `field: >10, <20, >30` will only give results when field is between 10 and 20, the higher then 30 part is ignored as the first part is more restrictive.

You can solve this by using ranges like: `field1: 10-20, >30;` Or by using a subgroup like `(field1: >10, <20); (field1: >30)`

5.1.2 PatternMatch

PatternMatchers are either OR'ed or applied as AND. This depends on the whether they are excluding or not.

Take the following matchers: `field: ~*foo; ~!*bar;` The first one is a "positive" matcher (field1 contains foo), the second one is a negative/excluding matcher (field1 does *not* contain bar).

If both were OR'ed we would either get a result when field1 contains "foo" or does not contain "bar", but if field one contains "foo" but also "bar" we would get an unexpected result. So the matchers are applied separately.

Caution: Don't use a regex unless there is an actual expression. `field: "^(foo|bar)"` can be easily done with field: `field1: ~>foo; field1: ~>bar"`

5.1.3 Last tip

If something is not possible because of how the condition is evaluated you can use subgroups to make it work.

Take the PatternMatcher, say we **want to** search a field that contains e.g. "foo" *or* does not contain "bar". With `field: ~*foo; ~!*bar;` this will not work, but if we use two subgroups `(field: ~*foo); (field: ~!*bar);` it will work!

5.2 What to expect with a condition

For all the examples assume we have the following records:

id	gender	reg_date	is_admin	enabled
10	male	2011-01-04	t	t
20	female	2011-01-04	f	f
30	male	2013-01-04	f	t
100	female	2013-05-04	f	f
500	male	2015-03-04	t	f

Tip: You are not limited a single table, the actual searching in a database is done by a search processor which may support searching complex structures or separated documents.

So no problem if you want to search for an invoice that has a customer relationship and you want to use the customer as leading condition.

5.3 Search for users with a specific gender

Say we want to find all users female users.

We use the following condition: `gender: female`

We will give use the following result.

id	gender	reg_date	is_admin	enabled
20	female	2011-01-04	f	f
100	female	2013-05-04	f	f

Or we can use a different approach by *excluding* male from the gender list.

```
gender: !man;
```

Which will give the same result.

Note: If we had another gender type like "N/A". Then we would have gotten all female users and users with gender "N/A".

5.4 Search for users with a specific gender and registration date

Say we want to find all users female users, that have registered in or after the year 2011 but before 2015. *Dates are in date notation year/month/day.*

The following conditions will all produce the same result, but use different methods to get the result.

All conditions will give the following result.

id	gender	reg_date	is_admin	enabled
20	female	2011-01-04	f	f
100	female	2013-05-04	f	f

Note: Notice that the date is between ", this is because any value part that is not a single word or number must be quoted in the FilterQuery syntax.

Female is a single word so this doesn't require quoting.

5.4.1 Explicit range

Find where gender is female and date is (inclusive) between "2011/01/01" and "2014/12/31".

```
gender: female; date: "2011/01/01" - "2014/12/31";
```

5.4.2 Explicit range with exclusive bounds

Sometimes the upper-value is not really predictable, for example you want to search for a date that falls in a leap year. Instead of figuring out the last day of the month you can use an exclusive upper-bound.

Find where gender is female and date is between (inclusive) "2011/01/01" and (exclusive) "2014/12/31".

The lower bound is inclusive (by default) meaning it will only match a value that is equal or higher than "2011/01/01".

The the upper-bound of the range is marked exclusive meaning it will only match values that are lower than "2015/01/01".

```
gender: female; date: "2011/01/01" - ]"2015/01/01";
```

And same thing can be done for the lower-bound.

```
gender: female; date: ["2012/12/31" - ]"2015/01/01";
```

The lower bound is now exclusive meaning it will only match a value that is higher than "2011/01/01".

5.4.3 Implicit range with Comparisons

Using ranges is just one method, it's also possible to use multiple comparisons, which is better known as an "implicit range". It has the same effect as a range, but is defined differently.

Caution: Implicit ranges can't (currently) be optimized, so if you have a value which is overlapping in a range this will not be optimized.
So avoid using implicit ranges whenever possible.

Find where gender is female and date is higher than "2011/01/01" and lower than "2014/12/31".

```
gender: female; date: >"2011/01/01", <"2015/01/01";
```

5.4.4 Multiple single values

So far we have only used ranges, but did you know it's also possible to use multiple single values? OK, this may seem a bit crazy but it's not uncommon, when you select a list of checkboxes all of these are technically single values.

For our date example this will result in 1460 single values (which for logical reason are not all shown here, this example only shows 4 dates).

```
gender: female; date: "2011/01/02", "2011/01/03", "2011/01/04", "2011/01/05";
```

Tip: The system already has an optimizer that can convert incremented values to ranges. So don't worry about the 1460 single values, in the end this this is simply converted into a single range.

But you are properly are gonna hit the maximum values per field limit. So it's best to avoid this whenever possible.

5.4.5 Subgroup range

Using subgroups in this case is just an example, normally you would use one of the methods described above.

Find where gender is female and subgroup 0 is matching, subgroup 0 matches when date is (inclusive) between "2011/01/01" and "2014/12/31".

```
gender: female; (date: "2011/01/01" - "2014/12/31";)
```

5.5 Search for users which either have admin access or are disabled

In the previous section we only used conditions where all the fields must match. But what if we want to search with an OR condition? We want to search for users which either have admin access **or** are disabled.

This is where we can use an OR'ed group. In an OR'ed group at least one field must match but the other fields are *optional*.

Using condition:

```
* is_admin: t; enabled: f;
```

Will give the following result.

id	gender	reg_date	is_admin	enabled
10	male	2011-01-04	t	t
20	female	2011-01-04	f	f
100	female	2013-05-04	f	f
500	male	2015-03-04	t	f

Lets analyze this result a bit further.

The first row matches because the user is an admin, the user is enabled but we can ignore this because we already have a positive match.

The second row matches, the user is not an admin it's disabled so the second field has a positive match.

Note: The OR'ed symbol works only on groups, because the condition always starts with a group the OR'ed symbol is only valid at the start of a condition or subgroup. So the following is invalid: `is_admin: t; * enabled: f;`

But this is valid: `is_admin: t; *(enabled: f)` and marks subgroup 0 as OR'ed.

5.6 Search for users which either "have admin access and are disabled" or female

Using OR'ed subgroups is great if want at least one field to match and mark the rest as optional. But this will not work if want all the fields to match but but just not together.

This is where subgroup (finally) come into play. Each subgroup can have it's own condition which is applied secondary to the parent-group and only fields within the subgroup will make it matching.

Using condition:

```
(is_admin: t; enabled: f); (gender: female);
```

Note: Subgroups are always OR'ed to each other, but at **least one must match** for the group it's in! A group can be meant as the condition's root (the root group) or a nested subgroup.

Will give the following result.

id	gender	reg_date	is_admin	enabled
20	female	2011-01-04	f	f
100	female	2013-05-04	f	f
500	male	2015-03-04	t	f

Lets analyze this result a bit further.

The first and second rows match because the user is a female, the second subgroup does not match but as subgroups are OR'ed this is no problem.

The last row matches because first subgroup matches, the user is an admin and is disabled, the second subgroup does not match and so is ignored.

Caution: Note that we used two subgroups, if we the placed either of the fields in the root of the condition like `gender: female; (is_admin: t; enabled: f);` We would have gotten a completely different result. The first subgroup must match as subgroups are only OR'ed to each other.
So in practice using `gender: female; (is_admin: t; enabled: f);` is the same as using `gender: female; is_admin: t; enabled: f;`

CHAPTER 6

Metadata

Class metadata is used by the `FieldSetBuilder` to populate a `FieldSet` instance field configuration with the metadata of a Model class.

To actually use the metadata component you first need a compatible metadata loader.

RollerworksSearch doesn't come bundled with a metadata loader, but you can use the RollerworksSearch Metadata extension as compatible metadata loader.

Note: The RollerworksSearch JMS Metadata extension is deprecated and will no longer be supported in future versions.

You only need to update the PHP, all metadata already defined is still compatible. But the `required` flag is no longer available and needs to be removed.

CHAPTER 7

Field Type

Field Types are used for configuring a search field.

As you probably know, searching with the system works by defining searching conditions. But in order to known how the value should be handled each field has a type specialized in handling the value type/format.

The type is used for configuring the field so you don't have to apply all the configuration for each field individually.

For a list of build-in types see Field Types Reference.

Creating you're own field type is also possible.

See How to Create a Custom Search Field Type for more information.

CHAPTER 8

The Cookbook

8.1 Type

8.1.1 How to Use Data Transformers

You'll often find the need to transform the data the user entered in a field into something else for use in your search processor. You could do this in the processor but then you will loose some power full features (including removing duplicates and validating ranges).

Say you have an invoice number that is provided in a special format like '2015-134', the first part is the year and second a number relative to the year. 2015-134 means number 134 of the year 2015.

Passing this value to the system it a bit troubling, because the system has no idea how to handle this format. It looks like a number but it's not.

It would be better if this value was in an InvoiceNumber value object, so the system can work with. This is where Data Transformers come into play.

The InvoiceNumber object is known as a normalized value.

InvoiceNumber value class

First create the InvoiceNumber class that holds an invoice number.

This technique is known as a 'value class', the InvoiceNumber is immutable meaning its internal values can't be changed.

```
// src/Acme/Invoice/InvoiceNumber.php

namespace Acme\Invoice;

class InvoiceNumber
{
    private $year;
    private $number;

    private function __construct($year, $number)
    {
        $this->year = $year;
        $this->number = $number;
    }

```

```
    public static function createFromString($input)
    {
        if (!is_string($input) || !preg_match('/^(?P<year>\d{4})-(?P<number>\d+)$/s', $input, $matche
            throw new \InvalidArgumentException('This not a valid invoice number.');
        }

        return new InvoiceNumber((int) $matches['year'], (int) ltrim($matches['number'], '0'));
    }

    public function equals(InvoiceNumber $input)
    {
        return $input == $this;
    }

    public function isHigher(InvoiceNumber $input)
    {
        if ($this->year > $input->year) {
            return true;
        }

        if ($input->year === $this->year && $this->number > $input->number) {
            return true;
        }

        return false;
    }

    public function isLower(InvoiceNumber $input)
    {
        if ($this->year < $input->year) {
            return true;
        }

        if ($input->year === $this->year && $this->number < $input->number) {
            return true;
        }

        return false;
    }

    public function __toString()
    {
        // Return the invoice number with leading zero
        return sprintf('%d-%04d', $this->year, $this->number);
    }
}
```

Creating the Transformer

Create an `InvoiceNumberTransformer` class - this class will be responsible for converting to and from the invoice number and the `InvoiceNumber` object:

```
// src/Acme/Invoice/Search/DataTransformer/InvoiceNumberTransformer.php

namespace Acme\Invoice\Search\DataTransformer;

use Acme\Invoice\InvoiceNumber;
```

```php
use Rollerworks\Component\Search\DataTransformerInterface;
use Rollerworks\Component\Search\Exception\TransformationFailedException;
use Rollerworks\Component\Search\Exception\UnexpectedTypeException;

class InvoiceNumberTransformer implements DataTransformerInterface
{
    public function transform($value)
    {
        if (!$value instanceof InvoiceNumber) {
            throw new UnexpectedTypeException($value, 'Acme\Invoice\InvoiceNumber');
        }

        return (string) $value;
    }

    public function reverseTransform($value)
    {
        if (null === $value) {
            return null;
        }

        try {
            return InvoiceNumber::createFromString($value);
        } catch (\Exception $e) {
            throw new TransformationFailedException('This not a valid invoice number.')
        }
    }
}
```

Tip: If the transformer is unable to transform the input to an `InvoiceNumber` a `TransformationFailedException` is thrown to indicate an invalid value.

The error message that is shown to user can be controlled with the `invalid_message` field option.

Note: When `null` is passed to the `transform()` method, your transformer should return an equivalent value of the type it is transforming to (e.g. an empty string, 0 for integers or 0.0 for floats).

Using the Transformer

Now that you have the transformer built, you just need to add it to your invoice field type.

```php
// src/Acme/Invoice/Search/Type/InvoiceNumberType.php

namespace Acme\Invoice\Search\Type;

use Acme\Invoice\Search\DataTransformer\InvoiceNumberTransformer;
use Rollerworks\Component\Search\AbstractFieldType;
use Rollerworks\Component\Search\Exception\InvalidConfigurationException;
use Rollerworks\Component\Search\FieldConfigInterface;

class InvoiceNumberType extends AbstractFieldType
{
    public function buildType(FieldConfigInterface $config, array $options)
    {
```

```
        $config->addViewTransformer(new InvoiceNumberTransformer());
    }

    public function getName()
    {
        return 'invoice_number';
    }
}
```

Cool, you're done! Your user will be able to enter an invoice number into the field and it will be transformed back into an InvoiceNumber object. This means after a successful transformation, the system will pass an `InvoiceNumber` object instead of a string value.

And when exporting the value the original format is shown.

So why Use the Data Transformer?

Transforming user input into a normalized value is done for multiple reasons.

- It makes using the value in the processor easier.
- It allows for localized input (like numbers on a none-latin charset).
- And most of all it makes it possible to optimize the search condition.

One very good example for usage is Value Comparisons.

8.1.2 How to Use Value Comparisons

A powerful feature of RollerworksSearch is the ability to optimize search conditions and perform basic validation of user input.

But in order to do this the system needs to understand which values are equal or lower/higher to other values. Especially when you are working with objects.

Note: Fields with range support enabled must have a Value Comparer in order to work properly. A missing Comparer will mark every range in the field invalid!

Assuming you have a field that handles invoice numbers as an InvoiceNumber and you configured that the field supports ranges.

InvoiceNumber value class

First create the InvoiceNumber class that holds an invoice number.

This technique is known as a 'value class', the InvoiceNumber is immutable meaning its internal values can't be changed.

```
// src/Acme/Invoice/InvoiceNumber.php

namespace Acme\Invoice;

class InvoiceNumber
{
    private $year;
```

```
    private $number;

    private function __construct($year, $number)
    {
        $this->year = $year;
        $this->number = $number;
    }

    public static function createFromString($input)
    {
        if (!is_string($input) || !preg_match('/^(?P<year>\d{4})-(?P<number>\d+)$/s', $input, $matche
            throw new \InvalidArgumentException('This not a valid invoice number.');
        }

        return new InvoiceNumber((int) $matches['year'], (int) ltrim($matches['number'], '0'));
    }

    public function equals(InvoiceNumber $input)
    {
        return $input == $this;
    }

    public function isHigher(InvoiceNumber $input)
    {
        if ($this->year > $input->year) {
            return true;
        }

        if ($input->year === $this->year && $this->number > $input->number) {
            return true;
        }

        return false;
    }

    public function isLower(InvoiceNumber $input)
    {
        if ($this->year < $input->year) {
            return true;
        }

        if ($input->year === $this->year && $this->number < $input->number) {
            return true;
        }

        return false;
    }

    public function __toString()
    {
        // Return the invoice number with leading zero
        return sprintf('%d-%04d', $this->year, $this->number);
    }
}
```

Tip: See How to Use Data Transformers on how to transform a user input to an InvoiceNumber.

Creating the Comparer

Create an `InvoiceNumberComparison` class - this class will be responsible for comparing values for equality and lower/higher `InvoiceNumber` objects:

```php
// src/Acme/Invoice/Search/ValueComparison/InvoiceNumberComparison.php

namespace Acme\Invoice\Search\ValueComparison;

use Acme\Invoice\InvoiceNumber;
use Rollerworks\Component\Search\ValueComparisonInterface;

class InvoiceNumberComparison implements ValueComparisonInterface
{
    public function isHigher($higher, $lower, array $options)
    {
        return $higher->isHigher($lower);
    }

    public function isLower($lower, $higher, array $options)
    {
        return $lower->isLower($higher);
    }

    public function isEqual($value, $nextValue, array $options)
    {
        return $value->equals($nextValue);
    }
}
```

Tip: A comparison method will only receive values that are returned by the field's data transformer.

You don't have to check if the input is what you expect, but if the input is invalid you need look at the configured data transformers.

Note: When `isLower()` and `isHigher()` are not supported, then both methods should be return `false`.

Using the Comparer

Now that you have the comparer built, you just need to add it to your invoice field type.

```php
// src/Acme/Invoice/Search/Type/InvoiceNumberType.php

namespace Acme\Invoice\Search\Type;

use Acme\Invoice\Search\DataTransformer\InvoiceNumberTransformer;
use Rollerworks\Component\Search\AbstractFieldType;
use Rollerworks\Component\Search\Exception\InvalidConfigurationException;
use Rollerworks\Component\Search\FieldConfigInterface;
use Rollerworks\Component\Search\ValueComparisonInterface;
use Rollerworks\Component\Search\ValuesBag;

class InvoiceNumberType extends AbstractFieldType
{
```

```
    private $valueComparison;

    public function __construct(ValueComparisonInterface $valueComparison)
    {
        $this->valueComparison = $valueComparison;
    }

    public function buildType(FieldConfigInterface $config, array $options)
    {
        $config->setValueComparison($this->valueComparison);
        $config->setValueTypeSupport(ValuesBag::VALUE_TYPE_RANGE, true);
        $config->setValueTypeSupport(ValuesBag::VALUE_TYPE_COMPARISON, true);

        $config->addViewTransformer(new InvoiceNumberTransformer());
    }

    public function getName()
    {
        return 'invoice_number';
    }
}
```

Cool, you're done! Input processors can now validate the bounds of ranges and optimizers can optimize the generated search condition.

Optimizing incremented values

Now that your type supports comparing values, you can extend the comparer with the ability to calculate increments.

Calculating increments is helps with optimizing single incremented values. For example: `1, 2, 3, 4, 5` can be converted to a `1-5` range which will decrease the size of the search condition and speed-up the search operation.

Note: Optimizing incremented values is done by the :class:`Rollerworks\\Component\\Search\\ConditionOptimizer\` optimizer. So make sure its enabled.

Instead of implementing the `ValueComparisonInterface` implement the `ValueIncrementerInterface` which extends the `ValueComparisonInterface` and adds the `getIncrementedValue` method for calculating increments.

```
// src/Acme/Invoice/Search/ValueComparison/InvoiceNumberComparison.php

namespace Acme\Invoice\Search\ValueComparison;

use Acme\Invoice\InvoiceNumber;
use Rollerworks\Component\Search\ValueIncrementerInterface;

class InvoiceNumberComparison implements ValueIncrementerInterface
{
    public function isHigher($higher, $lower, array $options)
    {
        return $higher->isHigher($lower);
    }

    public function isLower($lower, $higher, array $options)
    {
        return $lower->isLower($higher);
```

```
    }

    public function isEqual($value, $nextValue, array $options)
    {
        return $value->equals($nextValue);
    }

    public function getIncrementedValue($value, array $options, $increments = 1)
    {
        return new InvoiceNumber($value->getYear(), $value->getNumber() + $increments);
    }
}
```

Note: Technically it's possible to optimize `"2015-099", "2015-100", "2015-0001"` to `"2015-099"-"2015-0001"`, but only when we know if "2015-100" is the last invoice for the year 2015.

Cool, you're done! The new `InvoiceNumberComparison` can be registered as any normal value comparer.

8.1.3 How to Create a Custom Search Field Type

RollerworksSearch comes with a bunch of core field types available for building FieldSets. However there are situations where you may want to create a custom field type for a specific purpose. This recipe assumes you need a field definition which holds specially formatted Client ID's, based on the existing integer field. This section explains how the field is defined, and how you can register it for usage in your application.

The Client ID begins with a 'C' prefix, and is prepended with zeros until it's at least 4 digits.

Example: `C30320, C0001, C442482, C0020`.

Defining the Field Type

In order to create the custom field type, first you have to create the class representing the type. In this situation the class holding the field type will be called ClientIdType and the file will be stored in the default location for search field types, which is `<VendorName>\\Search\\Type`. Make sure the field extends `AbstractFieldType`:

```
// src/Acme/Client/Search/Type/ClientIdType.php

namespace Acme\Client\Search\Type;

use Acme\Client\Search\DataTransformer\ClientIdTransformer;
use Rollerworks\Component\Search\AbstractFieldType;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ClientIdType extends AbstractFieldType
{
    public function buildType(FieldConfigInterface $config, array $options)
    {
        // The integer-type configures the field by setting a DataTransformer
        // to transformer localized input to an integer.
        //
        // Our custom type only accepts input in a special format,
        // so DataTransformers set by integer are no longer needed here.
        $config->resetViewTransformers();

```

```
        $config->addViewTransformer(new ClientIdTransformer());
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(
            array(
                'precision' => 0
            )
        );
    }

    public function getName()
    {
        return 'client_id';
    }

    public function getParent()
    {
        return 'integer';
    }
}
```

Tip: The location of this file is not important - the `Search\\Type` directory is just a convention.

Here, the return value of the `getParent` function indicates that you're extending the `integer` field type. This means that, by default, you inherit all of the logic and rendering of that field type. To see some of the logic, check out the IntegerType class. There are three methods that are particularly important:

`buildType()` Each field type has a `buildType` method, which is where you configure and build any field(s).

`buildView()` This method is used to set any extra variables you'll need when rendering your field in a template. For example, in IntegerType, a `precision` variable is set and used in the template to set the `precision` attribute on the `input` field.

`configureOptions()` This defines options for your field type that can be used in `buildType()` and `buildView()`. There are a lot of options common to all fields (see field Field Type), but you can create as any others as needed.

Creating the ClientIdTransformer

Because the type handel's client ID's in a special format, the type needs a How to Use Data Transformers to transform the input back into a regular integer.

```
// src/Acme/Client/Search/DataTransformer/ClientIdTransformer.php

namespace Acme\Client\Search\DataTransformer;

use Rollerworks\Component\Search\DataTransformerInterface;
use Rollerworks\Component\Search\Exception\TransformationFailedException;

class ClientIdTransformer implements DataTransformerInterface
{
    public function transform($value)
    {
        return sprintf('C%04d', $value);
```

```
    }

    public function reverseTransform($value)
    {
        if (null !== $value && !is_scalar($value)) {
            throw new TransformationFailedException('Expected a scalar.');
        }

        return ltrim('C0');
    }
}
```

Using the Field Type

Now that the type is created, the Search system needs a way to find it.

This can be done in to ways;

You can choose to use your custom field type immediately, simply by creating a new instance of the type in one of your FieldSets:

```
use Acme\Client\Search\Type\ClientIdType;
use Rollerworks\Component\Search\Searches;

$searchFactory = new Searches::createSearchFactoryBuilder()->getSearchFactory();

$fieldset = $searchFactory->createFieldSetBuilder('clients')
    ->add('id', new ClientIdType())
    ->add('name', 'text')
    ->getFieldSet()
;
```

Or the by registering your field type in a `SearchExtension`.

Tip: Registering the type in a `SearchExtension` is the recommended way when you want to reuse the type in multiple FieldSets or when you need some additional parameters to the class constructor.

```
// src/Acme/Client/Search/ClientExtension.php

namespace Acme\Client\Search;

use Rollerworks\Component\Search\AbstractExtension;

class ClientExtension extends AbstractExtension
{
    protected function loadTypes()
    {
        return array(
            new Type\ClientIdType(),
        );
    }
}
```

And then register it at system using the FactoryBuilder.

```
...

$searchFactory = new Searches::createSearchFactoryBuilder()
    ->addExtension(new ClientExtension())
    ->getSearchFactory();
```

Now that type can be used for any field by type name the corresponds with the value returned by the `getName` method defined earlier.

```
use Rollerworks\Component\Search\Searches;

$searchFactory = new Searches::createSearchFactoryBuilder()->getSearchFactory();

$fieldset = $searchFactory->createFieldSetBuilder('clients')
    ->add('id', 'client_id')
    ->add('name', 'text')
    ->getFieldSet()
;
```

Further reading

Creating a field type is fun and easy, but did you know much more is possible than what is shown here? Learn more at: How to Use Data Transformers and How to Use Value Comparisons and it is also a good idea to test your types: How to Unit Test your Field Types

8.1.4 How to Create a Field Type Extension

Custom search field types are great when you need field types with a specific purpose, such as a gender selector, or a VAT number input.

But sometimes, you don't really need to add new field types - you want to add features on top of existing types. This is where field type extensions come in.

Field type extensions have 2 main use-cases:

1. You want to add a **generic feature to several types** (such as adding a "help" text to every field type);
2. You want to add a **specific feature to a single type** (such as adding a user friendly date picker feature to the "date" field type).

In both those cases, it might be possible to achieve your goal with custom field rendering, or custom search field types. But using field type extensions can be cleaner (by limiting the amount of business logic in templates) and more flexible (you can add several type extensions to a single field type).

Field type extensions can achieve most of what custom field types can do, but instead of being field types of their own, **they plug into existing types**.

Imagine you want to add a range selector to the `Number` type, but this requires a min and max value to select from, and maybe a step value.

You could of course do this by customizing how this field is rendered in a template. But field type extensions allow you to do this in a nice DRY fashion.

Defining the Field Type Extension

Your first task will be to create the field type extension class (called `RangeTypeExtension` in this article). By standard, field extensions usually live in the `Search\Type\Extension` directory of one of your applications/libraries.

When creating a field type extension, you can either implement the `FieldTypeExtensionInterface` interface or extend the `AbstractFieldTypeExtension` class. In most cases, it's easier to extend the abstract class:

```
// src/Acme/Client/Search/Type/Extension/RangeTypeExtension.php
namespace Acme\Client\Search\Type\Extension;

use Rollerworks\Component\Search\AbstractFieldTypeExtension;

class RangeTypeExtension extends AbstractFieldTypeExtension
{
    /**
     * Returns the name of the type being extended.
     *
     * @return string The name of the type being extended
     */
    public function getExtendedType()
    {
        return 'file';
    }
}
```

The only method you **must** implement is the `getExtendedType` function. It is used to indicate the name of the field type that will be extended by your extension.

Tip: The value you return in the `getExtendedType` method corresponds to the value returned by the `getName` method in the field type class you wish to extend.

In addition to the `getExtendedType` function, you will probably want to override one of the following methods:

- `buildType()`
- `buildView()`
- `configureOptions()`

For more information on what those methods do, you can refer to the Creating Custom Field Types cookbook article.

Adding the extension Business Logic

The goal of your extension is to display a range slider instead of an input field. As most JavaScript libraries already provide a simple way to trigger the rendering the extension only needs to pass the options to view for usage.

The template rendering the view can then use the configured options to trigger the rendering done by JavaScript.

Your field type extension class will need to do two things in order to extend the `number` field type:

1. Override the `configureOptions` method in order to add the `min_value`, `max_value` and `step_value` options;
2. Override the `'buildView` method in order to pass the options to the view.

```
// src/Acme/Client/Search/Type/Extension/RangeTypeExtension.php
namespace Acme\Client\Search\Type\Extension;

use Rollerworks\Component\Search\AbstractFieldTypeExtension;
use Rollerworks\Component\Search\SearchFieldView;
use Rollerworks\Component\Search\FieldConfigInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

```

```php
class RangeTypeExtension extends AbstractFieldTypeExtension
{
    /**
     * Returns the name of the type being extended.
     *
     * @return string The name of the type being extended
     */
    public function getExtendedType()
    {
        return 'number';
    }

    /**
     * Add the min_value, max_value, step_value options.
     *
     * @param OptionsResolver $resolver
     */
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setOptional(array('min_value', 'max_value', 'step_value'));
    }

    /**
     * Pass the image options to the view.
     *
     * @param SearchFieldView      $view
     * @param FieldConfigInterface $field
     * @param array                $options
     */
    public function buildView(SearchFieldView $view, FieldConfigInterface $field, array $options)
    {
        foreach (array('min_value', 'max_value', 'step_value') as $key) {
            if (array_key_exists($key, $options)) {
                $view->vars[$key] = $options[$key];
            }
        }
    }
}
```

Using the Field Type

Now the type extension is created, the Search system needs to know it exists, just like field types this can be done in to ways;

You can choose to register the extension using the `FactoryBuilder`

```php
use Acme\Client\Search\Type\Extension\RangeTypeExtension;
use Rollerworks\Component\Search\Searches;

$searchFactory = new Searches::createSearchFactoryBuilder()
    ->addTypeExtension(new RangeTypeExtension())
    ->getSearchFactory()
;
```

Or the by registering the type in a `SearchExtension`.

```php
// src/Acme/Client/Search/ClientExtension.php

namespace Acme\Client\Search;

use Rollerworks\Component\Search\AbstractExtension;

class ClientExtension extends AbstractExtension
{
    protected function loadTypeExtensions()
    {
        return array(
            new Type\Extension\RangeTypeExtension(),
        );
    }
}
```

And then register it at system using the FactoryBuilder.

```php
/* ... */

$searchFactory = new Searches::createSearchFactoryBuilder()
    ->addExtension(new ClientExtension())
    ->getSearchFactory();
```

Now that type can be used for any field by type name the corresponds with the value returned by the `getName` method defined earlier.

From now on, when adding a field of type `number` in your field, you can specify the `min_value`, `max_value` and `step_value` options that will be used to display an the range selector. For example:

```php
/* ... */

$fieldset = $searchFactory->createFieldSetBuilder('products')
    ->add('name', 'text')
    ->add('size', 'number', 'min_value' => 1, 'max_value' => 100)
    ->getFieldSet()
;
```

8.1.5 How to Unit Test your Field Types

A Field consists of 3 core object: a field type (implementing `FieldTypeInterface`) the `SearchField` and the `SearchFieldView`.

The only class that is usually manipulated by programmers is the field type class which serves as a field blueprint. It is used to generate the `SearchField` and the `SearchFieldView`. You could test it directly by mocking its interactions with the factory but it would be complex. It is better to pass it to SearchFactory like it is done in a real application. It is simple to bootstrap and you can trust the Search components enough to use them as a testing base.

There is already a class that you can benefit from for simple FieldTypes testing: `FieldTypeTestCase`. It is used to test the core types and you can use it to test your types too.

Note: Depending on the way you installed RollerworksSearch the tests may not be downloaded. Use the `--prefer-source` option with Composer if this is the case.

The Basics

The simplest `FieldTypeTestCase` implementation looks like the following:

```
// src/Acme/Invoice/Tests/Search/Type/InvoiceNumberTypeTest.php
namespace Acme\Invoice\Tests\Search\Type;

use Rollerworks\Component\Search\Test\FieldTypeTestCase;
use Acme\Invoice\Search\Type\InvoiceNumberType;
use Acme\Invoice\Search\ValueComparison\InvoiceNumberComparison;
use Acme\Invoice\InvoiceNumber;

class InvoiceNumberTypeTest extends FieldTypeTestCase
{
    public function testValidInvoiceNumber()
    {
        $field = $this->getFactory()->createField('invoice', 'invoice_number');

        $expectedOutput = new InvoiceNumber(2015, 20);
        $expectedView = '2015-0020';

        $this->assertTransformedEquals($field, $expectedOutput, '2015-0020', $expectedView);
        $this->assertTransformedEquals($field, $expectedOutput, '2015-020', $expectedView);
        $this->assertTransformedEquals($field, $expectedOutput, '2015-20', $expectedView);
    }

    public function testWrongInputFails()
    {
       $field = $this->getFactory()->createField('invoice', 'invoice_number');

        $this->assertTransformedFails($field, '201-0020');
        $this->assertTransformedFails($field, '2015-');
        $this->assertTransformedFails($field, '201500');
    }

    protected function getTestedType()
    {
        return 'invoice_number';
    }

    protected function getTypes()
    {
        return array(
            new InvoiceNumberType(
                new InvoiceNumberComparison()
            )
        );
    }
}
```

So, what does it test? Here comes a detailed explanation.

First you verify if the `FieldType` compiles. This includes basic class inheritance, the `buildField` function and options resolution. This should be the first test you write:

```
$type = new TestedType();
$form = $this->getFactory()->create($type);
```

This test checks that none of your data transformers used by the field failed. The `assertTransformedEquals`

checks that the value-input is transformed properly to the expected output and that the reverse transforming is what you expect:

```
$this->assertTransformedEquals($field, $expectedOutput, '2015-0020', $expectedView);
$this->assertTransformedEquals($field, $expectedOutput, '2015-020', $expectedView);
$this->assertTransformedEquals($field, $expectedOutput, '2015-20', $expectedView);

$form->submit($formData);
$this->assertTrue($form->isSynchronized());
```

Note: The expected view result is not required, but its a good practice to ensure the field transformers work properly.

Next, verify that invalid values are not transformed:

```
$this->assertTransformedFails($field, '201-0020');
```

Caution: Make sure to only call `getFactory` method and not use the private `factory` property to get the factory.
To access the factory builder (before calling the `getFactory` method) use the `factoryBuilder` property.

Adding a Type your Type Depends on

Your field type may depend on other types that are not registered by default. It might look like this:

```
// src/Acme/Invoice/Search/Type/TestedType.php

// ... the getParent method
return 'my_custom_type';
```

To create your type correctly, you need to make the other type available to the search factory in your test. The easiest way is to register it manually before creating the child type using the `getTypes` method:

```
// src/Acme/Test/Tests/Search/Type/TestedTypeTest.php
namespace Acme\Test\Tests\Search\Type;

use Rollerworks\Component\Search\Test\FieldTypeTestCase;
use Acme\Test\Search\Type\ParentType;
use Acme\Test\Search\Type\TestedType;
use Acme\Test\ValueObject;

class TestedTypeTest extends FieldTypeTestCase
{
    public function testValidValueTransforms()
    {
        $field = $this->getFactory()->createField('field_name', 'tested_type');

        $expectedOutput = new ValueObject(10, 20, 50);
        $expectedView = '{10, 20, 50}';

        $this->assertTransformedEquals($field, $expectedOutput, '{10, 20,50}', $expectedView);
    }

    protected function getTestedType()
    {
        return 'tested_type';
```

```
    }

    protected function getTypes()
    {
        return array(
            new ParentType(),
            new TestedType(),
        );
    }
}
```

Caution: Make sure the parent type you add is well tested. Otherwise you may be getting errors that are not related to the type you are currently testing but to its children.

Adding custom Extensions

It often happens that you use some options that are added by type extensions. One of the cases may be the Symfony `ValidatorExtension` with its `constraints` option. The `FieldTypeTestCase` loads only the core form extension so an "Invalid option" exception will be raised if you try to use it for testing a class that depends on other extensions. You need add those extensions to the factory object:

```
// src/Acme/Test/Tests/Search/Type/TestedTypeTest.php
namespace Acme\Test\Tests\Search\Type;

use Rollerworks\Component\Search\Test\FieldTypeTestCase;
use Rollerworks\Component\Search\Extension\Symfony\ValidatorExtension;

class TestedTypeTest extends FieldTypeTestCase
{
    protected function getTypeExtensions()
    {
        return array(
            new ValidatorExtension(),
        );
    }

    // ... your tests
}
```

Note: The Symfony `ValidatorExtension` class is provided by a separate package. See Installing the Library for more information to install this extension.

Testing against different Sets of Data

If you are not familiar yet with PHPUnit's data providers, this might be a good opportunity to use them:

```
// src/Acme/Test/Tests/Search/Type/TestedTypeTest.php
namespace Acme\Test\Tests\Search\Type;

use Rollerworks\Component\Search\Test\FieldTypeTestCase;
use Acme\Test\Search\Type\TestedType;
use Acme\Test\ValueObject;
```

```php
class TestedTypeTest extends FieldTypeTestCase
{
    protected function getTypes()
    {
        return array(
            new TestedType(),
        );
    }

    protected function getTestedType()
    {
        return 'tested_type';
    }

    /**
     * @dataProvider getValidTestData
     */
    public function testValidDataTransforms($input, $expected, $viewExpected = null)
    {
        $field = $this->getFactory()->createField('field_name', 'tested_type');
        $this->assertTransformedEquals($field, $expectedOutput, $input, $expectedView);
    }

    public function getValidTestData()
    {
        return array(
            array('{10, 20,50}', new ValueObject(10, 20, 50), '{10, 20, 50}'),
            array('{10, 20, 50}', new ValueObject(10, 20, 50), '{10, 20, 50}'),
            array('{10,20,50}', new ValueObject(10, 20, 50), '{10, 20, 50}'),
        );
    }
}
```

The code above will run your test three times with 3 different sets of data. This allows for decoupling the test fixtures from the tests and easily testing against multiple sets of data.

8.2 How to Create a Custom Input processor

RollerworksSearch already provides input processors for a wide range of formats, including XML, JSON, PHP Array's and the user-friendly FilterQuery.

But sometimes you need support something else, whether you want to make a simple string processor, use a binary file or fixed-length string syntax or anything you can think of, everything is possible.

Each processor follows a very simple principle, accept a user-input and transform this to SearchCondition object. That's it. You don't have to enforce groups field-names or even support ranges!

This example shows a simple processor which will accept any value and will place it in a list of configured fields, no support for ranges, comparisons or pattern-matchers.

Tip: In the future we hope to add support for an input processor that will work similar to what we are building here. Find out more at: SmartQuery - GitHub issue tracker

---

For example we provide the following input: `foobar, 2012-12-05, "bar"`. The values will be placed as single-values on the configured fields.

Say we have two fields that will be used for condition: field1 and field2. The created SearchCondition `ValuesGroup` will look like:

```
$valuesGroup = new ValuesGroup(ValuesGroup::GROUP_LOGICAL_OR);

$valuesBag = ValuesBag();
$valuesBag->addSingleValue(new SingleValue('foobar'));
$valuesBag->addSingleValue(new SingleValue('2012-12-05'));
$valuesBag->addSingleValue(new SingleValue('bar'));

$valuesGroup->addField('field1', $valuesBag);
$valuesGroup->addField('field2', $valuesBag);
```

First lets create a custom `ProcessorConfig` class for configuring the mapping fields that need to be used.

```
namespace Acme\Search\Input;

use Rollerworks\Component\Search\Input\ProcessorConfig;

class SingleValuesInputConfig extends InputProcessorInterface
{
    private $processingFields;

    public function setProcessingFields(array $processingFields)
    {
        $this->processingFields = $processingFields;
    }

    public function getProcessingFields()
    {
        return $this->processingFields;
    }
}
```

```
namespace Acme\Search\Input;

use Rollerworks\Component\Search\InputProcessorInterface;
use Rollerworks\Component\Search\Value\SingleValue;
use Rollerworks\Component\Search\ValuesBag;
use Rollerworks\Component\Search\ValuesGroup;

class SingleValuesInput implements InputProcessorInterface
{
    public function process(ProcessorConfig $config, $input)
    {
        if (!$config instanceof SingleValuesInputConfig) {
            throw new \InvalidArgumentException(
                sprintf(
                    'Expected 1 argument of type "%s", "%s" given',
                    'Acme\\Search\\Input\\SingleValuesInputConfig'
                     get_class($config)
                )
            );
        }

        if (!is_string($input)) {
            throw new \InvalidArgumentException(
                sprintf(
                    'Expected 2 argument of type "string", "%s" given',
```

```
                    gettype($input)
                )
            );
        }

        // Instead of using a complex regex or something we can simply use str_getcsv()
        // and run array_map() over the returned array to remove leading and trailing whitespace
        $values = array_map('trim', str_getcsv($input, ',', '"', '"'));

        $valuesGroup = new ValuesGroup(ValuesGroup::GROUP_LOGICAL_OR);
        $valuesBag = ValuesBag();

        foreach ($values as $value) {
            $valuesBag->addSingleValue(new SingleValue($value));
        }

        $processingFields = $config->getProcessingFields();

        // Each field gets all the values exactly once.
        foreach ($processingFields as $fieldName) {
            if (!$config->getFieldSet()->has($fieldName)) {
                throw new \RuntimeException(
                    sprintf('Unable to processing unregistered field "%s"', $fieldName)
                );
            }

            $valuesGroup->addField($fieldName, $valuesBag);
        }

        return $condition = new SearchCondition(
            $config->getFieldSet(),
            $valuesGroup
        );
    }
}
```

That's it, a very simple straightforward input processor, you can extent this functionality by also detecting ranges and other operands.

Need more inspiration? Take a look at one of the already provided input processors.

Tip: For this example we are using the `InputProcessorInterface` but it's also possible to leverage the `AbstractInput` which provides some helper methods for field alias resolving and type support validating.

Now that we have an input processor, it may be a good idea to create an exporter that can deal with search conditions within the input format.

See more at: How to Create a custom condition exporter

8.3 How to Create a custom condition exporter

The main reason you need an exporter is to export a search condition to a format that can later be processed by an input processor.

Same as the input processor an exporter follows a simple principle, it exports a search condition to a processable

output.

Note: It's possible that an exporter is us unable to export all search conditions it receives. Sometimes the condition is just too complex to be exported. *So don't force yourself to support all conditions.*

This example assumes you have created a custom input processor (as described in How to Create a Custom Input processor).

```php
namespace Acme\Search\Exporter;

use Rollerworks\Component\Search\ExporterInterface
use Rollerworks\Component\Search\SearchConditionInterface;

class SingleValuesExporter implements ExporterInterface
{
    public function exportCondition(SearchConditionInterface $condition)
    {
        $fields = $condition->getValuesGroup()->getFields();
        $values = array();

        foreach ($fields as $valuesBag) {
            $values = array_merge(array_map(array($this, 'valueExtractor'), $valuesBag->getSingleValu
        }

        return implode(', ', array_unique($values));
    }

    /**
     * @internal
     */
    public function valueExtractor(SingleValue $value)
    {
        return $this->exportValue($value->getViewValue());
    }

    /**
     * @internal
     */
    public function exportValue($value)
    {
        if ('' === $value) {
            throw new \InvalidArgumentException(
                'Unable to export empty view-value. Please make sure there is a view-value set.'
            );
        }

        if (!preg_match('/^([\p{L}\p{N}]+)$/siu', $value)) {
            return '"'.str_replace('"', '""', $value).'"';
        }

        return $value;
    }
}
```

That's it, a very simple straightforward condition exporter.

Need more inspiration? Take a look at one of the already provided exporters.

Tip: For this example we are using the `ExporterInterface` but it's also possible to leverage the `AbstractExporter` which provides some helper methods for field label resolving and PatternMatchType exporting.

Note that the `AbstractExporter` requires you to implement the `exportGroup` method rather than `exportCondition`.

CHAPTER 9

Field Types Reference

9.1 field Field Type

See `FieldType`.

The `field` type predefines a couple of options that are then available on all fields.

The most common one is the `SimpleValueComparison` to check for duplicated values.

9.2 text Field Type

The text field handles the most basic input text field.

Options	• *trim*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`TextType`

9.2.1 Field Options

trim

type: `bool` **default**: `true`

If true, the whitespace of the submitted string value will be stripped via the `trim()` function when the data is transformed. This guarantees that if a value is submitted with extra whitespace, it will be removed before the value is stored in the `ValuesBag`.

9.2.2 Inherited Options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.3 integer Field Type

Handles an input "number" field.

This field has different options on how to handle input values that aren't integers. By default, all non-integer values (e.g. 6.78) will round down (e.g. 6).

Options	• *rounding_mode* • *precision* • *grouping*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`IntegerType`

9.3.1 Field Options

precision

type: `integer` **default**: Locale-specific (usually around `3`)

This specifies how many decimals will be allowed until the field rounds the submitted value (via `rounding_mode`). For example, if `precision` is set to `2`, a submitted value of `20.123` will be rounded to, for example, `20.12` (depending on your `rounding_mode`).

rounding_mode

type: `integer` **default**: `IntegerToLocalizedStringTransformer::ROUND_DOWN`

By default, if the user enters a non-integer number, it will be rounded down. There are several other rounding methods, and each is a constant on the `IntegerToLocalizedStringTransformer`:

- `IntegerToLocalizedStringTransformer::ROUND_DOWN` Round towards zero.
- `IntegerToLocalizedStringTransformer::ROUND_FLOOR` Round towards negative infinity.
- `IntegerToLocalizedStringTransformer::ROUND_UP` Round away from zero.
- `IntegerToLocalizedStringTransformer::ROUND_CEILING` Round towards positive infinity.
- `IntegerToLocalizedStringTransformer::ROUND_HALF_DOWN` Round towards the "nearest neighbor". If both neighbors are equidistant, round down.
- `IntegerToLocalizedStringTransformer::ROUND_HALF_EVEN` Round towards the "nearest neighbor". If both neighbors are equidistant, round towards the even neighbor.
- `IntegerToLocalizedStringTransformer::ROUND_HALF_UP` Round towards the "nearest neighbor". If both neighbors are equidistant, round up.

grouping

type: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class.

Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

9.3.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.4 money Field Type

A field specialized in money data.

This field type allows you to process money with a currency. There are also several other options for customizing how the input and output of the data is handled.

Output Data Type	`Rollerworks\Component\Search\Extension\Core\Model\`
Options	• *default_currency* • *divisor* • *precision* • *grouping*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`MoneyType`

9.4.1 Field Options

default_currency

type: `string` **default**: `EUR`

Specifies the default currency that the money is being specified in. This value is only used when no currency symbol is detected.

This can be any 3 letter ISO 4217 code. You can also set this to false to enforce an explicit currency symbol.

divisor

type: `integer` **default**: `1`

If, for some reason, you need to divide your starting value by a number before passing it to the storage later, you can use the `divisor` option.

precision

type: `integer` **default**: `2`

For some reason, if you need some precision other than 2 decimal places, you can modify this value. You probably won't need to do this unless, for example, you want to round to the nearest dollar (set the precision to `0`).

grouping

type: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class.

Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

9.4.2 Inherited Options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.5 number Field Type

A field specialized in number as input.

This type offers different options for the precision, rounding, and grouping that you want to use for your number.

Options	• *rounding_mode* • *precision* • *grouping*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`NumberType`

9.5.1 Field Options

precision

type: `integer` **default**: Locale-specific (usually around `3`)

This specifies how many decimals will be allowed until the field rounds the submitted value (via `rounding_mode`). For example, if `precision` is set to `2`, a submitted value of `20.123` will be rounded to, for example, `20.12` (depending on your `rounding_mode`).

rounding_mode

type: `integer` **default**: `NumberToLocalizedStringTransformer::ROUND_HALFUP`

If a submitted number needs to be rounded (based on the `precision` option), you have several configurable options for that rounding. Each option is a constant on the `NumberToLocalizedStringTransformer`:

- `NumberToLocalizedStringTransformer::ROUND_DOWN` Round towards zero.
- `NumberToLocalizedStringTransformer::ROUND_FLOOR` Round towards negative infinity.
- `NumberToLocalizedStringTransformer::ROUND_UP` Round away from zero.
- `NumberToLocalizedStringTransformer::ROUND_CEILING` Round towards positive infinity.
- `NumberToLocalizedStringTransformer::ROUND_HALF_DOWN` Round towards the "nearest neighbor". If both neighbors are equidistant, round down.
- `NumberToLocalizedStringTransformer::ROUND_HALF_EVEN` Round towards the "nearest neighbor". If both neighbors are equidistant, round towards the even neighbor.
- `NumberToLocalizedStringTransformer::ROUND_HALF_UP` Round towards the "nearest neighbor". If both neighbors are equidistant, round up.

grouping

type: `integer` **default**: `false`

This value is used internally as the `NumberFormatter::GROUPING_USED` value when using PHP's `NumberFormatter` class.

Its documentation is non-existent, but it appears that if you set this to `true`, numbers will be grouped with a comma or period (depending on your locale): `12345.123` would display as `12,345.123`.

9.5.2 Inherited Options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.6 choice Field Type

A multi-purpose field used to allow the user to "choose" one or more options.

To use this field, you must specify *either* the `choice_list` or `choices` option.

Output Data Type	can be various types depending on the choice value
Options	• *choices* • *choice_list* • *label_as_value*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`ChoiceType`

9.6.1 Example Usage

The easiest way to use this field is to specify the choices directly via the `choices` option. The key of the array becomes the value that's actually set on your underlying object (e.g. `m`), while the value is what the user choices/types on the input (e.g. `Male`).

```
$builder->add('gender', 'choice', array(
    'choices'   => array('m' => 'Male', 'f' => 'Female'),
));
```

You can also use the `choice_list` option, which takes an object that can specify the choices.

9.6.2 Field Options

choices

type: `array` **default**: `array()`

This is the most basic way to specify the choices that should be used by this field. The `choices` option is an array, where the array key is the item value and the array value is the item's label:

```
$builder->add('gender', 'choice', array(
    'choices' => array('m' => 'Male', 'f' => 'Female')
));
```

choice_list

type: `Rollerworks\Component\Search\Extension\Core\ChoiceList\ChoiceListInterface`

This is one way of specifying the options to be used for this field. The `choice_list` option must be an instance of the `ChoiceListInterface`. For more advanced cases, a custom class that implements the interface can be created to supply the choices.

label_as_value

type: `bool` **default**: `false`

Each choice has a label and value, by default the value is used for transforming from view to the choice.

To use the label as value set this to `true`.

9.6.3 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.7 country Field Type

The `country` type is a subset of the choice type that allows the user to select from a large list of countries of the world.

The "value" for each country is the two-letter country code.

Note: The locale of your user is guessed using `Locale::getDefault()`

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses all of the countries of the world. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Overridden Options	• *choices*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`CountryType`

9.7.1 Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getRegionBundle()->getCountryNames()`

The country type defaults the `choices` option to the whole list of countries. The locale is used to translate the countries names.

9.7.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.8 language Field Type

The `language` type is a subset of the choice type that allows the user to select from a large list of languages. As an added bonus, the language names are displayed in the language of the user.

The "value" for each language is the *Unicode language identifier* (e.g. `fr` or `zh-Hant`).

Note: The locale of your user is guessed using `Locale::getDefault()`

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses all of the countries of the world. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Overridden Options	• *choices*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`LanguageType`

9.8.1 Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getLanguageBundle()->getLanguageNames()`.

The choices option defaults to all languages. The default locale is used to translate the languages names.

9.8.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.9 locale Field Type

The `locale` type is a subset of the `ChoiceType` that allows the user to select from a large list of locales (language+country). As an added bonus, the locale names are displayed in the language of the user.

The "value" for each locale is either the two letter ISO639-1 *language* code (e.g. `fr`), or the language code followed by an underscore (_), then the ISO3166 *country* code (e.g. `fr_FR` for French/France).

Note: The locale of your user is guessed using `Locale::getDefault()`

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of locales. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Overridden Options	• *choices*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`LocaleType`

9.9.1 Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getLocaleBundle()->getLocaleNames()`

The choices option defaults to all locales. It uses the default locale to specify the language.

9.9.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.10 timezone Field Type

The `timezone` type is a subset of the `ChoiceType` that allows the user to select from all possible timezones.

The "value" for each timezone is the full timezone name, such as `America/Chicago` or `Europe/Istanbul`.

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of timezones. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Overridden Options	• *choices*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`TimezoneType`

9.10.1 Overridden Options

choices

default: `Rollerworks\\Component\\Search\\Extension\\Core\\Type\\TimezoneType::getTimezones()`

The choices option defaults to all languages. The default locale is used to translate the languages names.

9.10.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.11 currency Field Type

The `currency` type is a subset of the choice type that allows the user to select from a large list of 3-letter ISO 4217 currencies.

Unlike the `choice` type, you don't need to specify a `choices` or `choice_list` option as the field type automatically uses a large list of currencies. You *can* specify either of these options manually, but then you should just use the `choice` type directly.

Overridden Options	• *choices*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`CurrencyType`

9.11.1 Overridden Options

choices

default: `Symfony\Component\Intl\Intl::getCurrencyBundle()->getCurrencyNames()`

The choices option defaults to all currencies.

9.11.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.12 date Field Type

A field to capture date input.

The provided input can be provided localized. The underlying data is stored as a `DateTime` object with UTC as timezone.

Output Data Type	`DateTime`
Options	• *format*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`DateType`

9.12.1 Field Options

format

type: `integer` or `string` **default**: IntlDateFormatter::MEDIUM

Option passed to the `IntlDateFormatter` class, used to transform user input into the proper format. This is critical when the input is passed localized format, and will define how the user will input the data. By default, the format is determined based on the current user locale: meaning that *the expected format will be different for different users*. You can override it by passing the format as a string.

For more information on valid formats, see Date/Time Format Syntax

9.12.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.13 datetime Field Type

A field to capture datetime input.

The provided input can be provided localized. The underlying data is stored as a `DateTime` object.

Output Data Type	`DateTime`
Options	• *with_seconds* • *with_minutes* • *model_timezone* • *input_timezone*
Parent type	field
Class	`DateTimeType`

9.13.1 Field Options

format

type: `integer` or `string` **default**: IntlDateFormatter::MEDIUM

Option passed to the `IntlDateFormatter` class, used to transform user input into the proper format. This is critical when the input is passed localized format, and will define how the user will input the data. By default, the format is determined based on the current user locale: meaning that *the expected format will be different for different users*. You can override it by passing the format as a string.

For more information on valid formats, see Date/Time Format Syntax

with_seconds

type: `bool` **default**: `false`

Whether or not to include seconds in the input.

with_minutes

type: `bool` **default**: `false`

Whether or not to include minutes in the input.

model_timezone

type: `string` **default**: system default timezone

Timezone that the storage data is stored in. This must be one of the PHP supported timezones.

input_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the PHP supported timezones.

9.13.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```php
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.14 time Field Type

A field to capture time input.

The provided input can be provided localized. The underlying data is stored as a `DateTime` object with UTC as time zone.

Output Data Type	`DateTime`
Options	• *with_seconds* • *with_minutes* • *model_timezone* • *input_timezone*
Inherited options	• *invalid_message* • *invalid_message_parameters*
Parent type	field
Class	`TimeType`

9.14.1 Field Options

with_seconds

type: `bool` **default**: `false`

Whether or not to include seconds in the input.

with_minutes

type: `bool` **default**: `false`

Whether or not to include minutes in the input.

model_timezone

type: `string` **default**: system default timezone

Timezone that the storage data is stored in. This must be one of the PHP supported timezones.

input_timezone

type: `string` **default**: system default timezone

Timezone that the input data is stored in. This must be one of the PHP supported timezones.

9.14.2 Inherited options

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

9.15 birthday Field Type

A date field that specializes in handling birthday or age data. Can accept both birthdate or age.

This type is essentially the same as the date type, but with a special option for handling an age in years.

Output Data Type	can be `DateTime` or `integer`
Options	• *allow_future_date*
Inherited Options	• *format* • *allow_age* • *allow_future_date*
Parent type	date
Class	`BirthdayType`

9.15.1 Field Options

allow_age

type: `bool` **default**: `true`

Allow age (in years) as accepted format.

> **Caution:** This will be produce a mixed result for the field values, as some maybe integer while other are `\DateTime` objects.

allow_future_date

type: `bool` **default**: `true`

Allow dates that are in the future.

9.15.2 Inherited options

These options inherit from the date type:

format

type: `integer` or `string` **default**: IntlDateFormatter::MEDIUM

Option passed to the `IntlDateFormatter` class, used to transform user input into the proper format. This is critical when the input is passed localized format, and will define how the user will input the data. By default, the format is determined based on the current user locale: meaning that *the expected format will be different for different users*. You can override it by passing the format as a string.

For more information on valid formats, see Date/Time Format Syntax

These options inherit from the field type:

invalid_message

type: `string` **default**: `This value is not valid`

This is the validation error message that's used if the data entered into this field doesn't make sense (i.e. fails validation).

This might happen, for example, if the user enters a nonsense string into a time field that cannot be converted into a real time or if the user enters a string (e.g. `apple`) into a number field.

Normal (business logic) validation (such as when setting a minimum length for a field) should be set using validation messages with your validation rules.

Validation is provided by eg the rollerworks/search-symfony-validator or any other supported SearchCondition validator.

invalid_message_parameters

type: `array` **default**: `array()`

When setting the `invalid_message` option, you may need to include some variables in the string. This can be done by adding placeholders to that option and including the variables in this option:

```
$builder->add('some_field', 'some_type', array(
    // ...
    'invalid_message'            => 'You entered an invalid value - it should include %num% letters',
    'invalid_message_parameters' => array('%num%' => 6),
));
```

A FieldSet is composed of *fields*, each of which are built with the help of a field *type* (e.g. a `text` type, `choice` type, etc). RollerworksSearch comes standard with a large list of field types that can be used.

9.16 Supported Field Types

The following field types are natively available in RollerworksSearch:

9.16.1 Text Fields

- text
- integer
- money
- number

9.16.2 Choice Fields

- choice
- country
- language
- locale
- timezone
- currency

9.16.3 Date and Time Fields

- date
- datetime
- time
- birthday

9.16.4 Base Fields

- field

CHAPTER 10

Input

The input component processes a condition to a `SearchCondition` instance.

Note: Only fields registered in the `FieldSet` will be processed, other fields are simple ignored.

The input can be provided in a wide range of formats.

Tip: The FilterQuery works similar to a spreadsheet formula's syntax and is perfect single input conditions.

10.1 Values limit

To prevent overloading your system the provided input can limited a by values (per group), maximum amount of groups and group nesting level.

Configuring happens using a `Rollerworks\Component\Search\Input\ProcessorConfig` object instance.

By default the input is limited to 10000 values per group and 100 groups in total, with a nesting level of 100 levels deep.

Changing these limits can be done by calling `setLimitValues()`, `setMaxGroups` and `setMaxNestingLevel()` respectively.

Unless you want to/must support a large number of values its best to not set these values too high.

Caution: Allowing users to pass a large number of values can result in a massive performance hit or even crashing of the application.
Setting the nesting level to high may require you to increase the `xdebug.max_nesting_level` value.

10.1.1 FilterQuery

Processes input in the FilterQuery format.

The FilterQuery consists is a user-friendly syntax for providing search conditions. Note that input is accepted in any local including none latin characters.

Tip: Spaces are ignored so that you can keep whitespace between condition characters for better readability.

`field-name:  value1, value2;` and `field-name:value1,value2;` Are technically the same.

The syntax works using whats called query-pairs consisting of a field-name and list of values. Values are separated using a single coma (`,`).

```
username: value1, value2;
```

"username" is the field-name, "value1" and "value" are the values of the username field.

The `;` character at the end of the query-pair indicates that the values list has ended, and a new query-pair or subgroup can be started. When the query-pair is not followed by anything (except optional space) the `;` can be omitted.

```
field1: value1, value2; field2: value1, value2;
```

Can shorted to.

```
field1: value1, value2; field2: value1, value2
```

Field-name

A field-name is limited in formatting, it must start with an alphabetic character or word from any language. So a to z or "" (price) in Simplified Chinese.

After this it may be followed be a number, alphabetic character, word or a dash `-` or underscore `_` any given number of times.

The following fields-names are valid:

-
- `price`
- `price0`
- `total_price`
- `total-price`

The following field-names are invalid:

- `0K` (must not start with a number)
- `0` (must not start with a number)
- `0` (must not start with a number)
- `_price` (must not start with an underscore)
- `-price` (must not start with an dash)
- `total-price:` (must not end with an double colon `:`).

Values

A value can be anything from a word, a sentence, number or specially formatted value like a date. But not all values can be used directly, if the value contains special characters (including spaces) the value must be quoted using double quotes (`"`).

Note: Decimal numbers must be quoted as well like `"10.00"`. `10.00` unquoted is considered invalid!

If the value itself contains double quotes these must be escaped by duplicating theme, so `va"lue` becomes `va""lue` and `va""lue` becomes `va""""lue`.

Escaped quotes will then be normalized when processing the input.

> **Caution:** Be careful with quotes at the beginning, `""foo"` might seem valid, but the first quote is used to indicate a quoted value.
> The correct usage is `""""foo"` which is transformed to `"foo`

Using a value like described above is whats called a "single value". If there are more we call them of a list of single values.

But as single values alone are not really useful, you can also use excluded single values, (excluded) ranges, comparisons and pattern matchers.

To mark a value as being excluded prefix it with an `!`, this works for almost all value types except comparisons and pattern matcher which have there own syntax.

```
field: !value, !1 - 10;
```

As some values are part of an expression like a range, the value is referred to as a value-part.

Ranges

A range consists of two sides, a lower and upper bound (inclusive by default). Each side is considered a value-part and must follow the value convention (as described above).

The following condition is seen as: field1 is between (inclusive) 1 and 100, and field2 is between (inclusive) -1 and 100.

```
field: 1-100; field2: "-1" - 100
```

Each side is inclusive by default, meaning the 'value' itself and anything lower/higher then it. To mark a value exclusive (everything between, but not the actual value) use the outer turning square brace `]` for the lower-bound and `[` for the upper-bound.

- `]1-100` is equal to (higher then) 1 and (equal to or lower then) 100)
- `[1-100` is equal to (equal to or higher then) 1 and (equal to or lower then) 100)
- `[1-100[` is equal to (equal to or higher then) 1 and (lower then) 100)
- `]1-100[` is equal to (higher then) 1 and (lower then) 100)

You can also mark a bound explicitly inclusive using `[` for lower-bound and `]` for the upper-bound to mark the. But as bounds inclusive by default you don't have to do this, it's just for explicitness.

Comparison

Comparisons are very straightforward, each comparison starts with an operator followed by a value-part.

Supported operators are:

- < (lower then)
- <= (lower then or equal to)
- <> (not higher or lower then (same as marking the value as excluded))
- > (higher then)

- >= (higher then or equal to)

```
field: >=1, < "-10", date: >"06/02/2015";
```

Tip: When ever possible try to use ranges instead of multiple comparisons, because ranges can be optimized.

PatternMatch

PatternMatchers work similar to Comparisons, everything starting with tilde (~) is considered a pattern-matcher.

Supported operators are:

- ~* (contains)
- ~> (starts with)
- ~< (ends with)
- ~? (regex matching)
- ~= (equals)

And not the excluding equivalent.

- ~!* (does not contain)
- ~!> (does not start with)
- ~!< (does not end with)
- ~!? (does not match regex)
- ~!= (equals)

Example: `field:  ~>foo, ~*"bar", ~?"^foo|bar$";`

To mark the pattern case insensitive add an 'i' directly after the '~'.

Example: `field:  ~i>foo, ~i!*"bar", ~i?"^foo|bar$";`

Note: The regex is limited to simple POSIX expressions. Actual usage is handled by the storage layer, and may not fully support complex expressions.

Most matchers can be easily solved without regexes, always try to use a normal matcher before trying a regex.

Caution: In most languages the Regex would start and end with a delimit, but in filter-query this is not the case.

Subgroups

For more complex conditions you can nest query-pairs inside subgroups. Subgroups are separated the same way as query-pairs, using the `;` character. And when the group closing character is not followed by anything (except optional space) the last `;` can be omitted.

```
(field-name: value1, value2;); (field-name: value1, value2)
```

Or in combination with query-pairs.

```
field-name: value1, value2; (field-name: value1, value2);
```

Tip: Notice that query-pair in the second subgroup does end with a `;`? That's because the processor is smart enough to know that the group has ended here and it can simply ignore the missing `;` and continue. If there was a second query-pair or nested subgroup an `;` is required.

By default all groups are marked as logical AND, meaning all the fields within the group must give a positive match. For explicitness you can use this to mark the group as logical AND.

```
&(field1: values; field2: values);
```

To change a group and make it OR'ed (at least one field must give a positive match), prefix the group with an `*` character.

```
*(field1: values; field2: values);
```

If you want to head-group (the condition itself) OR'ed or AND (default) use `*` or `&` as the first character in the condition.

```
*field1: values; field2: values;
```

```
&field1: values; field2: values;
```

Caution: The OR'ed symbol works only on groups, because the condition always starts with a group the OR'ed symbol is only valid at the start of a condition or subgroup. So the following is invalid: `is_admin: t; * enabled: f;`
But this is valid: `is_admin: t; *(enabled: f)` and marks subgroup 0 as OR'ed.

10.1.2 JsonInput

Processes input in the JSON format.

The required input structure is the same as ArrayInput

The advantage of using the JsonInput processor instead decoding the JSON object yourself is at the JsonInput processor does a linting on the provided input, ensuring the JSON input is valid and will give a more detail message on whats wrong with the input.

10.1.3 ArrayInput

Processes input provided as a PHP Array.

The provided input must be structured as follow;

Each entry must contain an array with either 'fields' and/or groups. Optionally the array can contain `'logical-case' => 'AND'` to make it AND-cased.

The "groups" array contains numeric groups with the structure as described above (fields and/or groups).

The fields array is an associative array where each key is the field-name and the values as follow.

All the keys are optional, but at least one must exists.

```
 array(
    'single-values'   => array('value1', 'value2')
    'excluded-values' => array('my value1', 'my value2')
    'ranges'          => array(array('lower'=> 10, 'upper' => 20))
    'excluded-ranges' => array(array('lower'=> 25, 'upper' => 30))
    'comparisons'     => array(array('value'=> 50, 'operator' => '>'))
    'pattern-matchers' => array(array('value'=> 'foo', 'type' => 'STARTS_WITH'))
)
```

The type of 'pattern-matchers' must either one of the following:

- CONTAINS
- STARTS_WITH
- ENDS_WITH
- REGEX
- NOT_CONTAINS
- NOT_STARTS_WITH
- NOT_ENDS_WITH
- NOT_REGEX

Full example:

```
array(
    'fields' => array(
        'field1' => array(
            'ranges' => array(
                array('lower' => 10, 'upper' => 20),
                array('lower' => 30, 'upper' => 40),
                array('lower' => 50, 'upper' => 60, 'inclusive-lower' => false),
                array('lower' => 70, 'upper' => 80, 'inclusive-upper' => false),
            )
        )
    ),
    'groups' => array(
        array(
            'logical-case' => 'AND'
            'fields' => array(
                'field1' => array(
                    'single-values' => array('value', 'value2', 'value3', 'value4', 'value5')
                )
            )
        )
    )
)
```

10.1.4 XmlInput

Processes input as an XML document.

See the XSD in 'src/Input/schema/dic/input/xml-input-1.0.xsd' for more information about the schema.

Any node/leave inside the `<field />`-node is optional. But at least one must exists.

Caution: Because of the way XSD validates the `<fields>` node must be provided before the `'<groups>` node.

```
<?xml version="1.0" encoding="UTF-8"'.'?'.'>
<search>
    <fields>
        <field name="field1">
            <single-values>
                <value>value</value>
                <value>value2</value>
            </single-values>

            <excluded-values>
                <value>value</value>
                <value>value2</value>
            </excluded-values>

            <ranges>
                <range>
                    <lower>55</lower>
                    <upper inclusive="false">60</upper>
                </range>
                <range>
                    <lower inclusive="false">70</lower>
                    <upper>80</upper>
                </range>
            </ranges>

            <excluded-ranges>
                <range>
                    <lower>10</lower>
                    <upper>20</upper>
                </range>
            </excluded-ranges>

            <comparisons>
                <compare operator="&gt;">10</compare>
                <compare operator="&lt;">50</compare>
            </comparisons>

            <pattern-matchers>
                <pattern-matcher type="contains">foo</pattern-matcher>
                <pattern-matcher type="ends_with" case-insensitive="true">bar</pattern-matcher>
            </pattern-matchers>
        </field>
    </fields>

    <groups>

        <group>
            <fields>
                <field name="field1">
                    <single-values>
                        <value>value</value>
                        <value>value2</value>
                    </single-values>
                </field>
            </fields>
```

```
        </group>

        <group>
            <fields>
                <field name="field1">
                    <single-values>
                        <value>value3</value>
                        <value>value4</value>
                    </single-values>
                </field>
            </fields>

            <!--<groups> ... </groups>-->

        </group>
    </groups>

</search>
```

F

I

T