Netgen's Search Extra for Ibexa CMS

Apr 22, 2022

Contents

CHAPTER 1

Reference

1.1 Reference

1.1.1 Installation instructions

To install Ibexa CMS Search Extra first add it as a dependency to your project:

```
$ composer require netgen/ibexa-search-extra:^3.0
```

Once the added dependency is installed, activate the bundle in `config/bundles.php` file by adding it to the returned array, together with other required bundles:

```
<?php

return [
    //...

    Netgen\Bundle\IbexaSearchExtraBundle\NetgenIbexaSearchExtraBundle::class => ['all
↪' => true],
]
```

1.1.2 LocationQuery criterion

`LocationQuery` criterion can be used with Content search. It allows grouping of Location criteria so that they apply together on a Location.

Note: This feature is available with both Solr and Legacy search engines.

For example, the following query will return Content of type `article` if it has hidden Location in subtree `/1/2/10/` and visible Location in some other subtree:

```
new Query([
    'filter' => new LogicalAnd([
        new ContentTypeIdentifier('article'),
        new Subtree('/1/2/10/'),
        new Visibility(Visibility::VISIBLE),
    ]),
]);
```

Using `LocationQuery` criterion you can write a query that will return Content of type `article` only when it has a visible Location in subtree `/1/2/10/`:

```
new Query([
    'filter' => new LogicalAnd([
        new ContentTypeIdentifier('article'),
        new LocationQuery(
            new LogicalAnd([
                new Subtree('/1/2/10/'),
                new Visibility(Visibility::VISIBLE),
            ])
        )
    ]),
]);
```

1.1.3 Custom Content subdocuments

This feature provides a way to index custom subdocuments *under* a Content document and a way to define criteria on them using Content search.

Note: This feature is available only for Solr search engine.

Note: It's not possible to search for custom subdocuments directly. Instead, you can define subdocument criteria as a part of Content search, using `SubdocumentQuery` criterion.

Note: Relationship between Content and it's subdocuments is not assumed. These can represent children under it's main Location, Content relations of a specific ContentType or something else altogether.

Indexing custom subdocuments

In order to index custom subdocuments, you will need to implement a subdocument mapper plugin. Two extension points are provided, depending on how you want to index subdocuments:

1. Indexing custom subdocuments per Content

To index custom subdocuments per Content, implement a service extending `Netgen\IbexaSearchExtra\Core\Search\Solr\SubdocumentMapper\ContentSubdocumentMapper` class, defining two methods:

- `accept(Content $content): bool`

  Here you will receive a Content that is being indexed as an instance of `Ibexa\Contracts\Core\Persistence\Content`. Using that object you have to decide whether you want to index custom subdocuments for it or not.

- `map(Content $content): Document`

  Again you will receive an instance of `Ibexa\Contracts\Core\Persistence\Content`, which you can use to build and return an array of `Ibexa\Contracts\Core\Search\Document` instances. These represent custom subdocuments that will be indexed under the given Content.

Code example:

```
final class MyContentSubdocumentMapper extends ContentSubdocumentMapper
{
    public function accept(Content $content)
    {
        return $content->versionInfo->contentInfo->contentTypeId === 42;
    }

    public function mapDocuments(Content $content)
    {
        return [
            new Document([
                'id' => 'unique_id',
                'fields' => [
                    new Field(
                        'document_type',
                        'content_subdocument',
                        new FieldType\IdentifierField()
                    ),
                    new Field('price', 5 new FieldType\IntegerField()),
                ],
            ]),
        ];
    }
}
```

You also have to configure the mapper in the service container configuration, tagging it with `netgen.ibexa_search_extra.solr.subdocument_mapper.content` tag so that the system can find it.

```
my_content_subdocument_mapper:
    class: MyContentSubdocumentMapper
    tags:
        - { name: netgen.ibexa_search_extra.solr.subdocument_mapper.content }
```

2. Indexing custom subdocuments per Content translation

To index custom subdocuments per Content translation, implement a service extending `Netgen\IbexaSearchExtra\Core\Search\Solr\SubdocumentMapper\ContentTranslationSubdocumentMap` class, defining two methods:

- `accept(Content $content, string $languageCode): bool`

  Here you will receive a Content being indexed and a language that it's being indexed in, as an instance of `Ibexa\Contracts\Core\Persistence\Content` and a language code string. Using these parameters you have to decide whether you want to index custom subdocuments for it or not.

- `map(Content $content, string $languageCode): Document`

  Again you receive an instance of `Ibexa\Contracts\Core\Persistence\Content` and a language code string. You can use these to build and return an array of `Ibexa\Contracts\Core\Search\Document` instances, representing custom subdocuments that will be indexed under the given translation of a Content.

Code example:

```
final class MyContentTranslationSubdocumentMapper extends ContentSubdocumentMapper
{
    public function accept(Content $content, $languageCode)
    {
        $contentTypeId = $content->versionInfo->contentInfo->contentTypeId;

        return $contentTypeId === 42 && $languageCode === 'cro-HR';
    }

    public function mapDocuments(Content $content, $languageCode)
    {
        return [
            new Document([
                'id' => 'unique_subdocument_id',
                'fields' => [
                    new Field(
                        'document_type',
                        'content_translation_subdocument',
                        new FieldType\IdentifierField()
                    ),
                    new Field('price', 5 new FieldType\IntegerField()),
                ],
            ]),
        ];
    }
}
```

You also have to configure the mapper in the service container configuration, tagging it with `netgen.ibexa_search_extra.solr.subdocument_mapper.content_translation` tag so that the system can find it.

```
my_content_translation_subdocument_mapper:
    class: MyContentTranslationSubdocumentMapper
    tags:
        - { name: netgen.ibexa_search_extra.solr.subdocument_mapper.content_
↪translation }
```

Note: It's mandatory to define `document_type` field of `IdentifierField` type, in every Document you are returning. You must not use `content` or `location` here, since these are already used by the search engine.

Using custom subdocuments in search

Indexing custom subdocuments would not be very useful without having a way to use them in search. For this `SubdocumentQuery` criterion is provided. It's constructor accepts two mandatory arguments:

1. `string $documentTypeIdentifier`

Document type identifier is used to match custom subdocument by it's type.

2. `Criterion $filter`

   Filter is an instance of a criterion, with following of the standard Ibexa CMS criteria being supported out of the box:

   - `LogicalAnd`
   - `LogicalNot`
   - `LogicalOr`
   - `CustomField`

Code example:

```
$query = new Query([
    'filter' => new LogicalAnd([
        new ContentTypeIdentifier('product'),
        new SubdocumentQuery(
            'product_variant',
            new LogicalAnd([
                new CustomField('visible_b', Operator::EQ, true),
                new CustomField('price_i', Operator::LT, 40),
            ])
        ),
    ])
]);

$searchResult = $searchService->findContent($query);
```

Implementing new criteria for `SubdocumentQuery`

If you want to implement additional criteria to use with `SubdocumentQuery` just implement is as usual. Then tag the visitor service with `netgen.ibexa_search_extra.solr.query.content.criterion_visitor.subdocument_query` tag.

1.1.4 Spellcheck suggestions

`Spellcheck suggestions` use Solr's SpellCheck component to provide inline query suggestions based on other, similar, terms.

This could be useful to provide the "did you mean" alternative to use when the search returns no results.

Note: This feature is available only with the Solr search engine.

1. Activation

In order to activate this feature, Solr has to be properly configured. First we need to create a new field type and then a new field of this type for spellcheck suggestions. Then we need to copy all textual fields to it.

solr/custom-fields-types.xml additions:

```
<fieldType name="text_suggest" class="solr.TextField" positionIncrementGap="100">
    <analyzer>
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.StandardFilterFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt
↪" />
        <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
</fieldType>

 <field name="spellcheck" type="text_suggest" indexed="true" stored="true"␣
↪multiValued="true" omitNorms="true" />

 <copyField source="*_t" dest="spellcheck" />
```

Then we need to set-up the SpellCheck component. It should already exist in solrconfig.xml but it might not be properly configured. Example configuration:

solr/solrconfig.xml additions

```
<!-- Spell Check

   The spell check component can return a list of alternative spelling
   suggestions.

   http://wiki.apache.org/solr/SpellCheckComponent
-->
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
    <str name="queryAnalyzerFieldType">text_general</str>

    <!-- Multiple "Spell Checkers" can be declared and used by this
         component
      -->

    <!-- a spellchecker built from a field of the main index -->
    <lst name="spellchecker">
        <str name="name">default</str>
        <!-- decide between dictionary based vs index based spelling suggestion, in␣
↪most cases it makes sense to use index based spell checker as it only generates␣
↪terms which are actually present in your search corpus -->
        <str name="classname">solr.IndexBasedSpellChecker</str>
        <!-- field to use -->
        <str name="field">spellcheck</str>
        <!-- the spellcheck distance measure used, the default is the internal␣
↪levenshtein -->
        <str name="distanceMeasure">org.apache.lucene.search.spell.LevensteinDistance
↪</str>
        <!-- buildOnCommit|buildOnOptimize -->
        <str name="buildOnCommit">true</str>
        <!-- $solr.solr.home/data/spellchecker-->
        <str name="spellcheckIndexDir">./spellchecker</str>
        <str name="accuracy">0.7</str>
        <float name="thresholdTokenFrequency">.0001</float>
    </lst>

    <!-- a spellchecker that can break or combine words.  See "/spell" handler below␣
↪for usage -->
    <!--
```

(continues on next page)

(continued from previous page)

```
    <lst name="spellchecker">
        <str name="name">wordbreak</str>
        <str name="classname">solr.WordBreakSolrSpellChecker</str>
        <str name="field">name</str>
        <str name="combineWords">true</str>
        <str name="breakWords">true</str>
        <int name="maxChanges">10</int>
        </lst>
        -->
</searchComponent>
```

In order to get suggestions during search, we need to tell our select request handler to use the previously configured SpellCheck component.

Example request handler configuration:

```
  <!-- SearchHandler

       http://wiki.apache.org/solr/SearchHandler

       For processing Search Queries, the primary Request Handler
       provided with Solr is "SearchHandler" It delegates to a sequent
       of SearchComponents (see below) and supports distributed
       queries across multiple shards
  -->
  <requestHandler name="/select" class="solr.SearchHandler">
  <!-- default values for query parameters can be specified, these
       will be overridden by parameters in the request
  -->
  <lst name="defaults">
       <str name="echoParams">explicit</str>
       <int name="rows">10</int>
       <!-- <str name="df">text</str> -->
       <str name="spellcheck.dictionary">default</str>
       <str name="spellcheck">on</str>
       <str name="spellcheck.extendedResults">true</str>
       <str name="spellcheck.count">10</str>
       <str name="spellcheck.alternativeTermCount">5</str>
       <str name="spellcheck.maxResultsForSuggest">5</str>
       <str name="spellcheck.collate">true</str>
       <str name="spellcheck.collateExtendedResults">true</str>
       <str name="spellcheck.maxCollationTries">10</str>
       <str name="spellcheck.maxCollations">5</str>
  </lst>
  <arr name="last-components">
       <str>spellcheck</str>
  </arr>
</requestHandler>
```

At last, our fulltext search criterion has to implement the `Netgen\IbexaSearchExtra\API\Values\Content\Query\Crit` interface.

Here's the example of a criterion which extends Ibexa CMS fulltext criterion and implements the required interface:

```
<?php

namespace AcmeBundle\API\Values\Content\Query\Criterion;
```

(continues on next page)

(continued from previous page)

```
use Ibexa\Contracts\Core\Repository\Values\Content\Query\Criterion\FullText as␣
↪BaseFullTextCriterion;
use Netgen\IbexaSearchExtra\API\Values\Content\Query\Criterion\FulltextSpellcheck;
use Netgen\IbexaSearchExtra\API\Values\Content\SpellcheckQuery;

class FullTextCriterion extends BaseFullTextCriterion implements FulltextSpellcheck
{
    /**
     * Gets query to be used for spell check.
     *
     * @return \Netgen\IbexaSearchExtra\API\Values\Content\SpellcheckQuery
     */
    public function getSpellcheckQuery()
    {
        $spellcheckQuery = new SpellcheckQuery();
        $spellcheckQuery->query = $this->value;
        $spellcheckQuery->count = 10;

        return $spellcheckQuery;
    }
}
```

Once activated, you will get the spellcheck suggestions in SearchResult object.

1.1.5 Extra fields from Solr

This feature allows you to extract additionally indexed Solr fields from each SearchHit in SearchResult. For example, you can index some fields from children content on the parent content and then get those fields during search (eg. children count).

Note: This feature is available only with the Solr search engine.

1. Usage

In order for this functionality to work, you have to use overridden `Netgen\IbexaSearchExtra\API\Values\Content\Searc` or `Netgen\IbexaSearchExtra\API\Values\Content\Search\LocationQuery` queries and use it's property *extraFields* to provide a list of additional fields that you want to extract from the Solr document. Those fields, if exist, and their values will appear in the *extraFields* property of each `SearchHit` object contained in the `SearchResult`.

2. Example

Example of a content field mapper:

```
public function mapFields(Content $content)
{
    return [
        new Field(
            'extra_field_example',
            5,
```

(continues on next page)

(continued from previous page)

```
            new IntegerField()
        ),
    ];
}
```

Search example:

```
/** @var \Netgen\IbexaSearchExtra\API\Values\Content\Search\Query $query **/
$query = new Query();

$query->extraFields = [
    'extra_field_example_i',
];

/** @var \Netgen\IbexaSiteApi\API\FindService $findService **/
$searchResult = $findService->findContent($query);

/** @var \Netgen\IbexaSearchExtra\API\Values\Content\Search\SearchHit $searchHit **/
foreach ($searchResult->searchHits as $searchHit) {
    var_dump($searchHit->extraFields);
}
```

This will output the following data:

```
array(1) { ["extra_field_example_i"]=> int(5) }
```


- *Installation instructions*
- *LocationQuery criterion*
- *Custom Content subdocuments*
- *Spellcheck suggestions*
- *Extra fields from Solr*