doofinder Bundle

Installation

doofinder

composer

in your composer.json file add the following repo under your

```
"repositories": [
    {
      "type": "vcs",
      "url":  "git@bitbucket.org:asioso/pimcore-doofinder-module.git"
    }
  ],
```

after that make sure you have access to the repo and added your ssh key to your bitbucket account. test if composer can find the package.

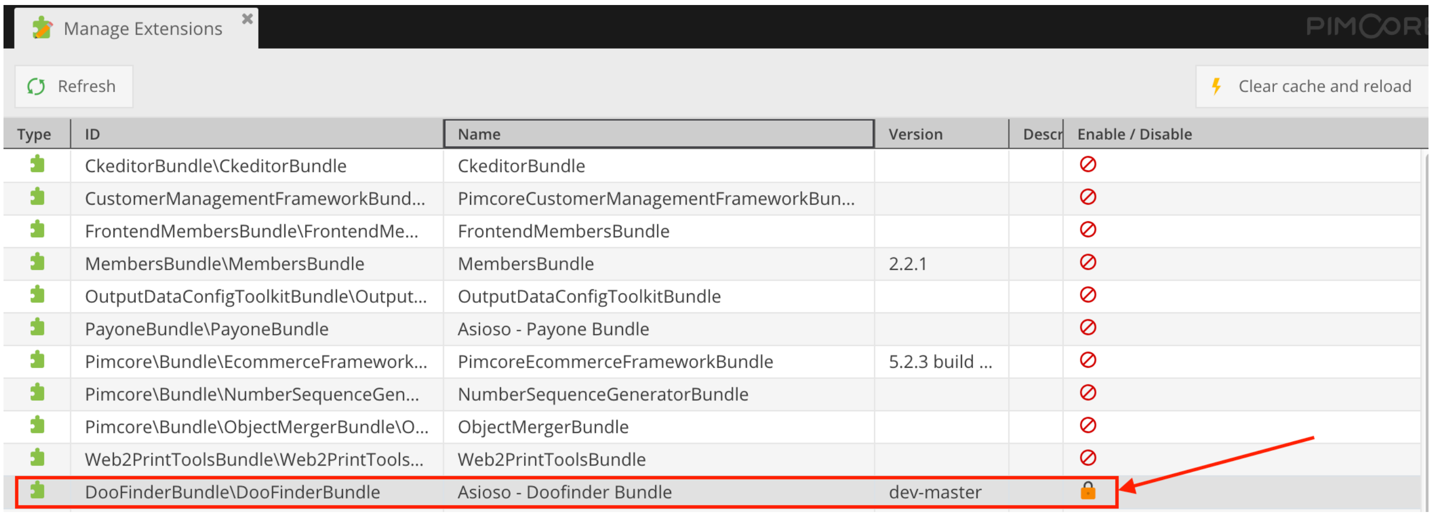
```bash
composer search asioso

>>>>>
asioso/pimcore-doofinder-module A bundle to help with dooFinder
```

add the bundle to composer.json with

```bash
composer require asioso/pimcore-doofinder-module:dev-master
```

Extension manager

no need to enable the bundle in the pimcore extension manager after you have run composer,, it's automatically taken care of.



Type	ID	Name	Version	Descr	Enable / Disable
🧩	CkeditorBundle\CkeditorBundle	CkeditorBundle			⊘
🧩	CustomerManagementFrameworkBund...	PimcoreCustomerManagementFrameworkBun...			⊘
🧩	FrontendMembersBundle\FrontendMe...	FrontendMembersBundle			⊘
🧩	MembersBundle\MembersBundle	MembersBundle	2.2.1		⊘
🧩	OutputDataConfigToolkitBundle\Output...	OutputDataConfigToolkitBundle			⊘
🧩	PayoneBundle\PayoneBundle	Asioso - Payone Bundle			⊘
🧩	Pimcore\Bundle\EcommerceFramework...	PimcoreEcommerceFrameworkBundle	5.2.3 build ...		⊘
🧩	Pimcore\Bundle\NumberSequenceGen...	NumberSequenceGeneratorBundle			⊘
🧩	Pimcore\Bundle\ObjectMergerBundle\O...	ObjectMergerBundle			⊘
🧩	Web2PrintToolsBundle\Web2PrintTools...	Web2PrintToolsBundle			⊘
🧩	DooFinderBundle\DooFinderBundle	Asioso - Doofinder Bundle	dev-master		🔒

It might be possible that you have to add the minimal configuration in any project related config.yml file first, otherwise the Kernel might not boot.

What's next?

After defining your configuration, to generate your datafeed you will need to define a pimcore/symfony command to do the heavy lifting for you.

If you are familiar with Pimcore you will know that you have multiple possibilities to query for objects * use Listings via the API * write your own SQL query * using an Index

You can find a implementation of such a command here, which uses both default object listings and *AdvancedMysql* Index Service and runs with efficient ressource management in mind.

Configuration

what's absolutely necessary to be added to your config.yml file:

```
doo_finder:
    search_api_key: "<search-api-key>"
    management_api_key: "<management-api-key>"
```

Define your Engine Configurations

Take a look here for a deeper insights on the internal workings of this bundle, but in short: doofinder let's you define multiple search engines for your account. Each of these engines might have several **types** (so to say indices) for content. Each **type**'s content might greatly differ in structure and content, however they all must share the same *language*-property (also *currency*!) from their parent engine.

So why does this matter?

For example you want to put your webshop's products from pimcore to doofinder. Let's say you have active localization for German (DE) und English (EN) and Prices in Euro(€) and Pound(£). That would mean you would still have to create two search engines in doofinder, one for each language respectively.

So you will have add the following configurations (simplified)

```
- de_engine (DE , €)
    + de - products

- en_engine (EN , £)
    + en -products
```

Given the case you would like to offer both currencies in both languages, you will need four engines!

```
- de_eur_engine (DE , €)
    + de-eur - products

- en_eur_engine (EN , €)
    + en-eur -products

- de_pound_engine (DE , £)
    + de_pound - products

- en__pound engine (EN , £)
    + en_pound -products
```

Keep this in mind, but apparently that's how doofinder is designed.

Note before you continue reading, I'm aware that there is some potential in mixing up terminology, because of the following: * in doofinder: * an engine is just a *search engine* with any number of types (indices) attached. * in this bundle: * an engine refers to exactly one engine and one type.

This is due to design and let's us determine an engine and type specific representation per Object. This is important, you can read more on this in the internals.

Some Example Configuration

Below you can see a configuration we have been using for a project. We are using 3 **types** for one search engine. So each bundle-engine here uses the same but references different *types*.

- *products* with type: **test_products**
- *products_special* with type: **test*products*special**
- *content* with type: **test_content**

way more interesting are the engine's item definitions. *class* defines the object's classname it's "listening" to, the *field* set defines the feed representation.

- **dfAttribute**: is the attribute's name obviously (this will end up in feeds header line)
- **classAttribute**: either:
    - 'self' with combination of a merger
    - the attributes Name. e.g 'description' - we are using PathProperty and Reflection to retrieve the value
- **merger**: originally designed to merge arrays, but can be used to do other things as well. just use it as a callback where you can control the output. see here for more details
- **locale**: very useful if the classAttribute is a localized field
- **getter**: if classAttribute itself is another object, then *getter* will be executed.

This settings and behavior came up during development, so there is still potential to do this better and more user friendly. Configuration can be quite redundant right now for example.

```
doo_finder:
        search_api_key: "<search-api-key>"
        management_api_key: "<management-api-key>"
    search_engines:
       engine:
            name: "products"
            type: "test_product"
            site_url: "https://example.de/"
            language: "German"
            currency: "Euro"
            hashId: "<hashId>"
            baseURL: "https://example.de"
            user: ~
            item:
                class: "AppBundle\\Model\\DefaultProduct"
                listing: "Pimcore\\Model\\DataObject\\Product\\Listing"
                fields:
                    -  {dfAttribute: "title", classAttribute: "self", merger: [ { cl
                    -  {dfAttribute: "description", classAttribute: "description", m
                    -  {dfAttribute: "image link", classAttribute: "self", locale: '
                    -  {dfAttribute: "product_type", classAttribute: "categories", l
                    -  {dfAttribute: "link",  url: [ { class: "AppBundle\\doofinder\
                    -  {dfAttribute: "attributes", classAttribute: "self", locale: '
                    -  {dfAttribute: "price",  classAttribute: "self", merger: [ { c
                    -  {dfAttribute: "uvp price",  classAttribute: "self", merger: [
                    -  {dfAttribute: "id", classAttribute: "id"}
                    -  {dfAttribute: "brand", classAttribute: "brand", locale: "de"
                    -  {dfAttribute: "sku_field", classAttribute: "sku" }
                    -  {dfAttribute: "sku2", classAttribute: "skuTwo" }
```

```
                     -  {dfAttribute: "sku3", classAttribute: "skuThree",}
                     -  {dfAttribute: "manufacturer", classAttribute: "manufacturer",

        engine_product_special:
             name: "products_special"
             type: "test_product_special"
             site_url: "https://example.de/"
             language: "German"
             currency: "Euro"
             hashId: "<hashId>"
             baseURL: "https://example.de"
             user: ~
             item:
                 class: "AppBundle\\Model\\DefaultProductSpecial"
                 listing: "Pimcore\\Model\\DataObject\\Product\\Listing"
                 fields:
                     -  {dfAttribute: "title", classAttribute: "name", locale: "de"
                     -  {dfAttribute: "description", classAttribute: "description", m
                     -  {dfAttribute: "image_link", classAttribute: "self", locale: '
                     -  {dfAttribute: "product type", classAttribute: "categories", l
                     -  {dfAttribute: "link",  url: [ { class: "AppBundle\\doofinder\
                     -  {dfAttribute: "id", classAttribute: "id"}

        engine_content:
                 name: "content"
                 type: "test_content"
                 site_url: "https://example.de/"
                 language: "German"
                 currency: "Euro"
                 hashId: "<hashId>"
                 baseURL: "https://example.de"
                 user: ~
                 item:
                     class: "\\Pimcore\\Model\\Document"
                     listing: "\\Pimcore\\Model\\Document\\Listing"
                     listing_arguments: [unpublished: "false", condition: "`parentId`
                     fields:
                         -  {dfAttribute: "id", classAttribute: "id"  }
                         -  {dfAttribute: "title", classAttribute: "title", locale: '
                         -  {dfAttribute: "description", classAttribute: "descriptior
                         -  {dfAttribute: "metadata", classAttribute: "metadata", loc
                         -  {dfAttribute: "link",  url: [ { class: "AppBundle\\doofir
                         #-  {dfAttribute: "image_link", classAttribute: "self", loca
                         #-  {dfAttribute: "content", classAttribute: "self", merger:
```

Internals

What is doofinder

If you are not familar with doofinder it's best to take a look at their website. BUT basically they provide a search engine for your content/products etc., which is accessible either by API or customizable search layer to you and your clients.

All you have to do is to make your content available to doofinder and include their search snippet.

Bundle Architecture

This Pimcore Bundle should help you with bringing your content to doofinder's search engines. There are two approaches to connect do doofinder:

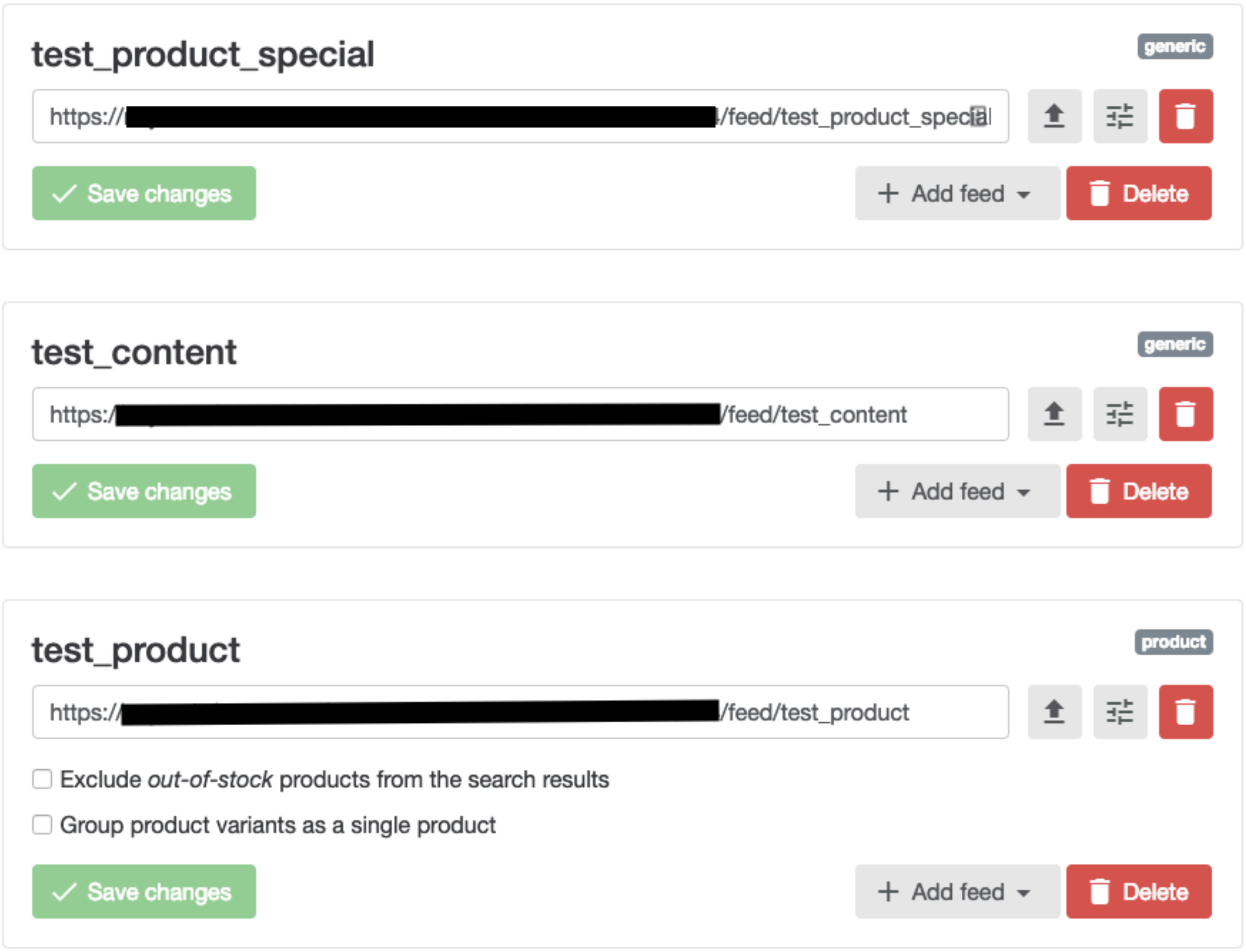
API

You could say this is still under development, since we turned our focus on the feed strategy because of the sheer number of objects we had to work with for the moment.

The original idea was to hook into the publishing process of pimcore DataObjets and push the updated data to doofinder then (the eventListener is still there, but deactived ;)), but we ran into api quota restrictions and timeouts, beside the fact that we needed a mechanism to do batch processeing anyway, which is now realized by the data feed.

Data Feed

The datafeed concept is that you supply doofinder with a number of urls where a crawler can find your feeds representing your content.


test_product_special

generic

https://[redacted]/feed/test_product_special

test_content

generic

https:/[redacted]/feed/test_content

test_product

product

https://[redacted]/feed/test_product

☐ Exclude *out-of-stock* products from the search results

☐ Group product variants as a single product

If you are new to this topics, lease read more about this in the [doofinder documentation](#) in short doofinder expects for example a *.txt* file with a header line, separated by a pipe ('|') character. other formats like xml would also be possible, but we used txt because it would get less bloated in file size in comparison to xml.

```
title|link|description|id|price|image link|product type
```

entries then will look like the following way:

```
LG Flatron M2262D 22" Full HD LCD|http://www.example.com/electronics/tv/LGM2262D.htm
```

We use the bundles configuration (see here: [Configuration](#)) to determine what should be in the file.

so basically all you have to do is to call this (this is taken from [our example command](#):

PHP

```php
/**
 * @var DooFinderBundle\DependencyInjection\DooFinder\IDooFinderServiceHandler
 */
$doofinder = $this->dooFinder;

/*
 * returns an array with <engineHashId> => [ data ]
 * e.g.:   [
 *          'hash1'=>['data' => ['id'=> 1, 'name'=> 'en_name'], 'engine'=> 'hash
 *          'hash2'=>['data' => ['id'=> 1, 'name'=> 'de_name'], 'engine'=> 'hash
 *         ]
 */
$feed = $dooFinder->getValuesForEngine($object);

foreach ($feed as $engineKey => $feedContent) {

    //append the data line to each engine's type-feed file, $header is only requ
    $doofinderFile->writeToFile($feedContent['engine'], $feedContent['type'], $f
}
```

Search Layer

When doofinder is fed with content you can include a configured search layer (configuration takes place in the search engine's configuration). You can use a controller action we wrote to render all required javascript. example below:

```
    <!-- PHP --->
    <?php if($this->getLocale() == "en"){
        $params = array(
            "hashID"=> "<en_engine_hash>",
            "locale"=> "en",
            "selector" => "#s",
            "zone" => "eu1",
        );
    }else{
        $params = array(
            "hashID"=> "<de_engine_hash>",
            "locale"=> "de",
            "selector" => "#s",
            "zone" => "eu1",
        );
    }
    ?>
    <?php echo $this->action("doofinderLayer", "DooFinder", "DooFinder", $params );
```

```
{#TWIG#}
{% if app.request.locale == 'en'%}
    {{ render(controller('DooFinderBundle:DooFinder:doofinderLayer',{"hashID": "<en_
{% else %}
    {{ render(controller('DooFinderBundle:DooFinder:doofinderLayer',{"hashID": "<de_
{% endif%}
```

Mergers

You might ask: "What the Hell?" But they offer more flexibility and additional processing callbacks. For example take a look at the configuration below, where we want to get a product's type (s) following doofinder's documentation (More than one category? Right on!)

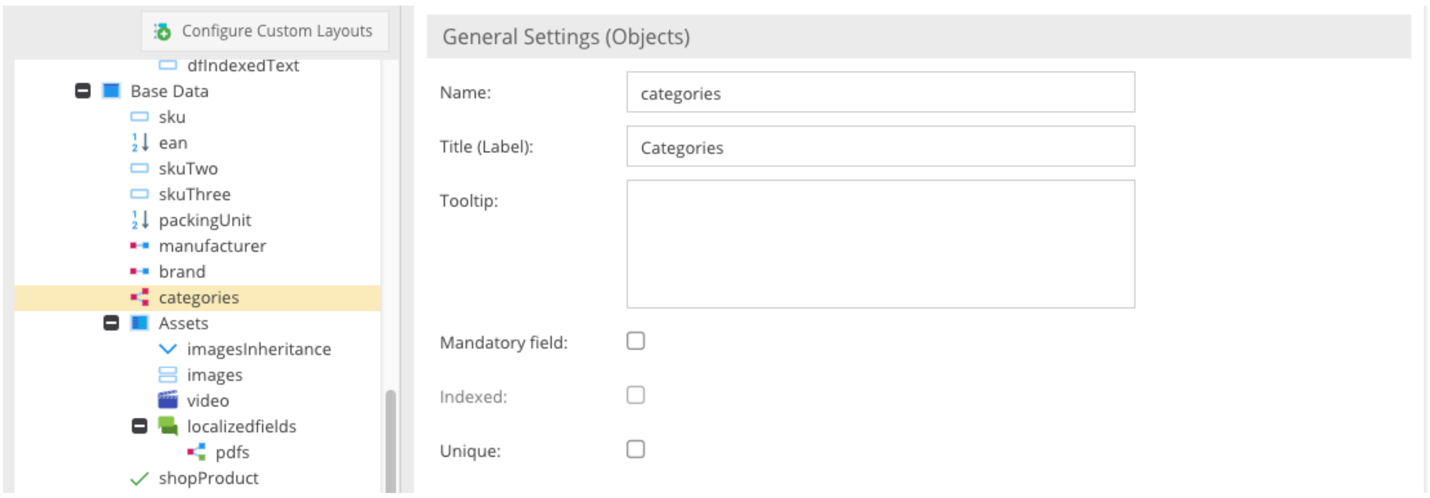
The Category Tree Example

```
...
-  {dfAttribute: "product_type", classAttribute: "categories", locale: "de", merger:
...
```

To just quote doofinder here:

> You can specify **several category trees** for an item. A sports shoe could belong to both the *Sports > Clothes > Shoes and Snickers > Jogging*, for instance. You can do this with either a xml or txt feed: * For a XML feed: Just add more than one tag. * For a plain text feed: Use the **%%** character to indicate another category tree. i.e.: *Sports > Clothes > Shoes %% Snickers > Jogging*.

Our Product class does store categories as a m to n relation to a ShopCategory class. Addtionally we have to differentiate between shop categories and internal categories (e.g. discounts, shipping type, etc.)



General Settings (Objects)

Name: categories

Title (Label): Categories

Tooltip:

Mandatory field: ☐

Indexed: ☐

Unique: ☐

So the resulting Merger class taking care of all that can be seen here, which generates exactly the category trees we wanted for *shop* categories and applies additionally removes unwanted characters.

PHP

```php
<?php

namespace AppBundle\DooFinder;


use AppBundle\Model\ShopCategory;
use DooFinderBundle\Merger\IDooFinderItemMerger;

class CategoriesMerger implements IDooFinderItemMerger
{

    protected $_badChars = array('"',"\r\n","\n","\r","\t", "|");
    protected $_repChars = array(""," "," "," "," ", "");

    public function merge( $objects, $options = array() )
    {
        if($objects == null){
            return "";
        }

        $data = array();

        foreach($objects as $object){
            if(!$object instanceof ShopCategory ){
                continue;
            }
            /**
             * @var $object ShopCategory
             */
            if($object->isShopCategory()){
                $catLine = array();
                $list = $object->getParentCategoryList($object);
                foreach($list as $cat){
                    $line = $cat->getName($options['locale']);
                    $line = str_replace($this->_badChars, $this->_repChars, $line);
                    $catLine[] = $line;

                }
                $data[] = trim(implode(" > ", $catLine));
            }
        }
        return trim(implode(" %% ", $data));
    }
}
```

Generate *flat* Text

A product's row in the datafeed must be a single line without, but what about attributes that hold multiline descriptions in html? The merger below shows an example how to achieve that:

```php
PHP
<?php

namespace AppBundle\DooFinder;


use AppBundle\Model\DefaultProduct;
use DooFinderBundle\Merger\IDooFinderItemMerger;

class DescriptionMerger implements IDooFinderItemMerger
{

    public function merge($object, $options = array())
    {
        if($object instanceof DefaultProduct)
        {
            $locale = $options[0]['locale'];
            $description = $object->getDescription($locale);

            //remove all tags
            $description = strip_tags($description);

            //remove line breaks
            $description = preg_replace('/\s+/S', " ", $description);

            return $description;
        }
    }
}
```

to apply this merger, again just put it in the configuration:

```
...
-  {dfAttribute: "description", classAttribute: "description", merger: [ { class: "A
...
```

URL Provider

You will need a way to build URLs for your products during datafeed generation.

below you can see an example, which mimics the way product URLs are generated within the pimcore e-commerce bundle default product implementation.

```php
<?php

namespace AppBundle\DooFinder;


use AppBundle\Model\DefaultProduct;
use DooFinderBundle\Merger\AbstractURLProvider;
use DooFinderBundle\Merger\IURLProvider;

class DefaultProductURLProvider extends AbstractURLProvider implements IURLProvider
{

    /**
     * @param $object
     * @param $locale
     * @param $route
     * @param null $prefix
     * @return string
     */
    public function getUrlForObject($object, $locale, $route, $prefix = null): strir
    {
        /**
         * @var $object DefaultProduct
         *
         */

        // add id
        //if (!array_key_exists('product', $params)) {
            $params['product'] = $object->getId();
        //}

        //add prefix / by default language/shop
        if (!array_key_exists('prefix', $params)) {
            if ($params['document']) {
                $params['prefix'] = substr($params['document']->getFullPath(), 1);
            } else {
                $language = $locale;
                #$language = \Pimcore::getContainer()->get('request_stack')->getCurr
                $params['prefix'] = substr($language, 0, 2) . '/shop';
            }
        }

        // add name
        if (!array_key_exists('name', $params)) {
            // add category path
            $category = $object->getFirstCategory();
            if ($category) {
                $path = $category->getNavigationPath($params['rootCategory'], $paran
                $params['name'] = $path."/";
            }

            // add name
            $name = \Pimcore\File::getValidFilename($object->getOSName());
            $params['name'] .= preg_replace('#-{2,}#', '-', $name);
        }

        unset($params['rootCategory']);
        unset($params['document']);

        // create url
        $urlHelper = \Pimcore::getContainer()->get('pimcore.templating.view_helper.p

        return $this->getBaseURL() . $urlHelper($params, $route);


    }

}
```

To use this Provider class again just put it in the configuration:

```
...
-  {dfAttribute: "link",  url: [ { class: "AppBundle\\doofinder\\DefaultProductURLPr
...
```

Feeds

If you are familiar with Pimcore you will know that you have multiple possibilities to query for objects

- use Listings via the API
- write your own SQL query
- using an Index

You can find a implementation of such a command here, which uses both default object listings and *OptimizedMysql* Index Service and runs with efficient ressource management in mind.

run the command like this:

```bash
$bin/console doo:build --force --process --notify=<example@email.com> --notify=<othe
```

options:

- --force: disables interactive questions in the command flow
- --notify: sends out a email to the defined recipient, when the command is done. multiple recipients possible!
- --process: notifies doofinder to process all new datafeeds, using the management api
- --gzip: compress your feed files

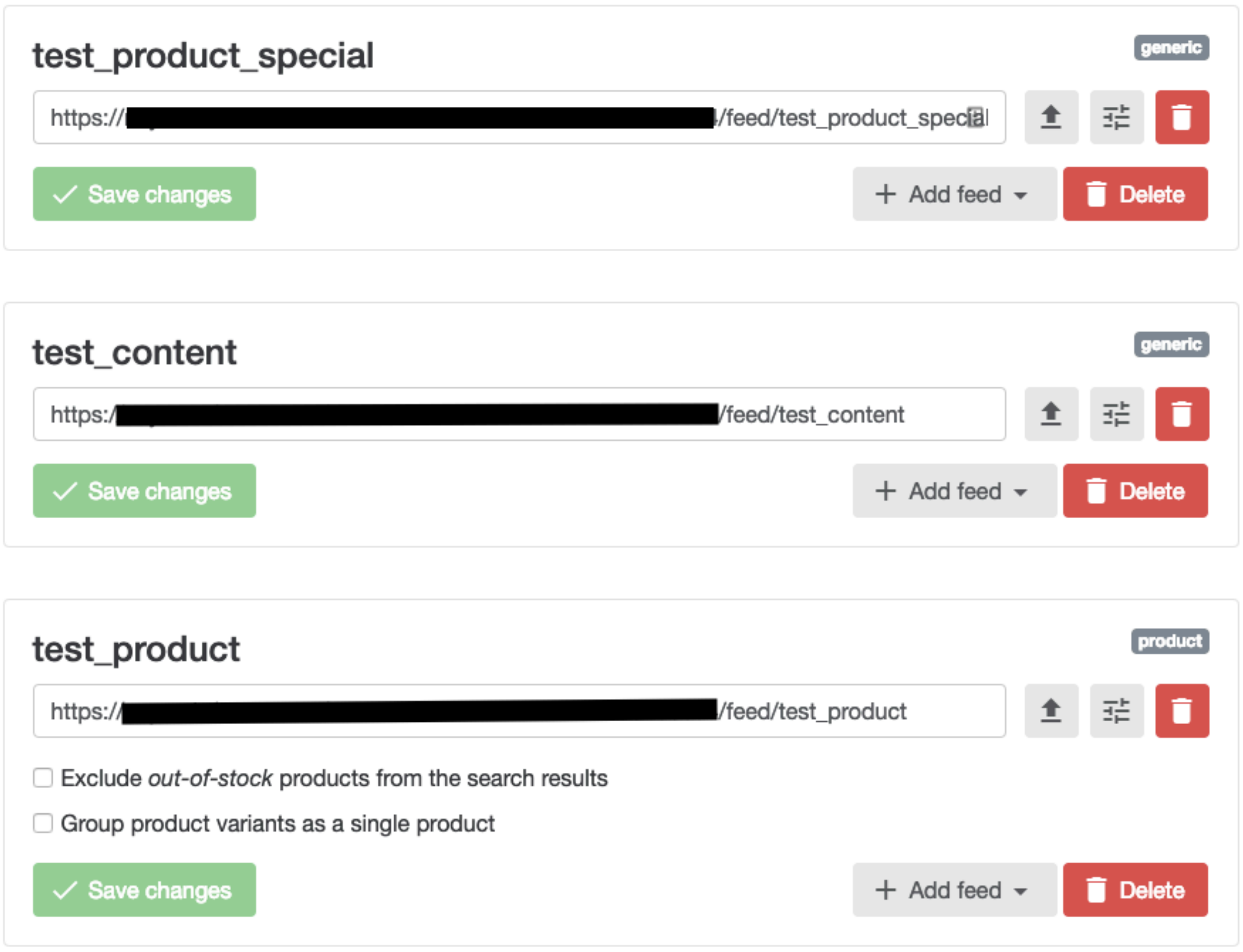
as a result there should be a */data* folder in your project's root directory with one file for every configured doofinder search engine and type combination prefixed with a timestamp.

> **naming convention:** {Ymd}_{Hi}_feed_{engine_hash}_{type}.txt
>
> e.g: 20181116_101605_feed_<someHash>_test_product.txt

There is a controller action dedicated to serve the latest datafeed file under the following route:

```
/dooFinder/{hashId}/feed/{type}
```

this would check against the feed urls we have configured in doofinder, e.g.:


test_product_special

generic

https://[redacted]/feed/test_product_special

Save changes

Add feed

Delete

test_content

generic

https://[redacted]/feed/test_content

Save changes

Add feed

Delete

test_product

product

https://[redacted]/feed/test_product

☐ Exclude *out-of-stock* products from the search results

☐ Group product variants as a single product

Save changes

Add feed

Delete

Examples

Take a look at the [/examples]() folder. Here is a collection of code samples we came up with to achieve all the things we needed.