

Blackbit Data Director Bundle

Import XML, CSV, JSON, Excel files to Pimcore objects, assets, documents + Export feeds + create REST API without any programming

For an overview how to use this plugin, please see our [tutorial videos](#).

Installation

Composer

To get the plugin code you have to [buy the plugin](#) or write an email to info@blackbit.de.

You then either get access to the bundle's [Bitbucket repository](#) or you get the plugin code as a zip file. Accessing the Bitbucket repository has the advantage that you will always see changes to the plugin in the pull requests and are able to update to a new version yourself - please visit [this page](#) if this sounds interesting to you - if it does, please send us the email address of your BitBucket account so we can allow access to the repository.

When we allow your account to access our repository, please add the repository to the `composer.json` in your Pimcore root folder (see [Composer repositories](#)):

```
"repositories": [  
    {  
        "type": "vcs",  
        "url": "git@bitbucket.org:blackbitwerbung/pimcore-plugins-data-director"  
    }  
]
```

(Please [add your public SSH key to your Bitbucket account](#) for this to work)

Alternatively if you received the plugin code as zip file, please upload the zip file to your server - e.g. create a folder `bundles` in the Pimcore root folder) and add the following to your `composer.json`:

```
"repositories": [  
    {  
        "type": "artifact",  
        "url": "./bundles/"  
    }  
]
```

Beware that when you put the zip directly in the Pimcore root folder, and add `"url": "./"` it will still work but Composer will scan *all* files under the Pimcore root recursively to find bundle zip files (incl. assets, versions etc) - which will take quite a long time.

Then you should be able to execute `composer require blackbit/data-director` (or `composer update blackbit/data-director --with-dependencies` for updates if you already have this bundle installed) from CLI.

At last you have to enable and install the plugin, either via browser UI or via CLI `bin/console pimcore:bundle:enable BlackbitDataDirectorBundle` && `bin/console pimcore:bundle:install BlackbitDataDirectorBundle`

You can always access the latest version by executing `composer update blackbit/data-director --with-dependencies` on CLI.

Migrations / Updates

From version to version we extend the bundle continually. So there will be some modifications on the database scheme. Those migrations get executed automatically, so you do not have to care for this.

Nevertheless, if you do not see the Data Director main menu icon in the Pimcore backend, it may be caused by a failed migration. You can execute the migrations manually via `bin/console pimcore:bundle:install BlackbitDataDirectorBundle` to see which migration fails.

Quick overview


Importing will run in two phases:

1. Parse source file, import data into a flat database table

In the first phase data does not get transformed. This is purely a fast way to import source data to an intermediate data container named **raw data table**.

You have to [specify which fields shall be extracted from the import source](#) to the raw data table.

2. Map data from intermediate table to object / asset / document fields / attributes.

In the attribute mapping panel all fields of the target class are listed. For each field you can assign the raw data field from 1. to be used. Additionally / alternatively you can use multiple raw data fields or modify the source data via callback functions if necessary - for example parse numbers, find related objects, transform the data to the expected data type. These mapping settings can be configured by clicking the  button.

Also in the field mapping settings window you can mark one or multiple fields as **key fields**. During import it is first tried to find already existing objects which have the same values in the key fields - if one or multiple elements get found, they get updated instead of new elements being created.

When multiple fields are set to be used as key fields, they are connected by **AND**, so objects only get updated if all key fields match the assigned raw data fields.

In the dataport panel settings under **Advanced Options** you can set the **import mode**, available are:

- Create and edit objects
- Create new objects, do not edit existing ones
- Edit existing objects, do not create new ones

You can [skip single raw data items](#) from being processed by returning **null** for a key field.

How to import data to objects

You can start an import on multiple ways:

- manually in the Pimcore backend on the panel **History & manual import** with the buttons **Start rawdata import**, **Import raw data to objects** and **Start complete import**
- manually by right-clicking the target folder in the data objects tree or on the import resource (if Pimcore assets are used as import resource)
- via CLI by executing one of these scripts:

- `bin/console data-director:extract <Dataport-ID> [--rm]` - Parse source resource (e.g. file or URL) and write data to a flat database table (replace with the real ID).

If `--rm` is set, the imported file will get deleted after raw data import. For cronjob imports the `--rm` flag has to be set because otherwise the importer always imports from the same file.

- `bin/console data-director:process <Dataport-ID> [Rawdata-ID] [--force]` - Maps raw data to Pimcore objects, updates existing objects or creates new ones as needed.

By default a raw data item gets only imported if the raw data for the found object changed since the last import (for performance reasons and to not overwrite manually edited values with old import data). By setting `--force` (or `-f`) you can bypass this check.

- you can additionally use `-vvv` to get all log / error messages

- To see all parameters and a description of these commands you can use `bin/console data-director:extract --help` or `bin/console data-director:process --help`.

- `bin/console data-director:complete <Dataport-ID>` combines raw data extraction and data processing - it first executes `data-director:extract` and afterwards `data-director:process`

- start import automatically if import source changes (configurable via checkbox **Run automatically on new data** in dataport settings). If activated the import gets started as soon as a file / object gets saved which is inside the configured **import file path** (for Pimcore asset based imports) or which matches the given filter criteria (SQL condition for Pimcore-based imports). In this case the import gets started **only for the modified file / object** (not for all files / objects which match the path / filter criteria).

Automatic imports also get triggered if an element is referenced by the saved object.

Example: you have an import (with **Run automatically on new data** enabled) which shall update an assets metadata field **product name** to the product's name of a data object which the asset is referenced in. As soon as the product gets saved, the asset metadata import gets started for all assets which are referenced by this product.

This is also useful when you dynamically create text data from product attributes via [placeholders](#). This way you will always have the updated text even if you have placeholders which fetch data from referenced objects.

If you do not want to execute the dataport also for dependent objects, you can add `{% if id %} o_id = '{{ id }}' {% endif %}` to the dataport's SQL condition. As soon as an object gets saved this [placeholder](#) gets replaced by the just saved object's ID and thus the dataport will only get automatically executed for this one object.

- start imports via [REST API call](#)

Dataport settings

Import resources

Importing data from folders

You can specify a folder path as import source instead of a single file. In this case files in this folder (and subfolders) are imported subsequently (sorted by modification time - oldest first). Already imported files are stored in an `archive` folder within the import folder.

In addition to the source file fields you can use the field `__source` to obtain the source file name. This can be useful if you want to process the source file name in the object import (e.g. to call a dependent import with the same import file). It will contain a local file copy of the import resource to ensure that even if you use the `--rm` option the file will be available to follow-up imports.

Moreover, there is the special field selector `__updated` which provides you the modification time (Unix timestamp) of the import file.

You can also specify a Pimcore asset folder instead of a file system folder. This enables you to enable the checkbox to automatically start imports if files in this folder change or are uploaded. In this case the import is only run with the changed file not with all files in the given folder.

Importing files with glob expression (wildcards)

It is possible to enter an expression like `/path/to/import/files/*.xml` - or any other glob expression, see [Wikipedia Glob](#) to see all options for wildcards. Also a relative path expression based on the Pimcore root folder or on Pimcore assets is possible.

Importing data from URLs

You can specify a URL instead of a file.

Authorization is supported, e.g. `https://username:password@example.org/file.csv`

You can also provide FTP and SFTP URLs:

```
ftp://username:password@example.org/folder/file.csv
sftp://username:password@example.org/folder/file.csv
```

Importing remote data via SOAP, authentication token or other complex methods

You can use a `curl` command as import resource when you want to access data from an API. Examples: SOAP:

```
curl --username:password \
  --header "Content-Type: text/xml; charset=UTF-8" \
  --header "SOAPAction: ACTION_YOU_WANT_TO_CALL" \
  --data <FILE_NAME> <SOAP_WEB_SERVICE_ENDPOINT_URL>
```

Accessing REST-API via token authentication (e.g. OAuth):

```
curl "<Authentication Token URL>" --header 'Authorization: Basic ABC...' \
  | jq -r '.access_token' \
  | xargs -I {access_token} curl -s --header 'Authorization: Bearer {access_token}' "<Data UR
```

Alternatively you can use the integrated [REST API](#) to *push* data from an external system into Pimcore - or in other words: to trigger the import by sending an HTTP request which includes the data to be imported in its message body.

Importing data from zip file

When you use a zip file as import resource (be it local or remote) the bundle extracts the files and determines the actual import file based on the file extensions, e.g. *.csv for dataports with source data type CSV*, *.xml for XML* etc.). All other files get recognized as assets which can be accessed by their file path within the zip file.

Example: You have a zip file `abc.zip` which contains the following files:

- `import.csv`
- `image.jpg`
- `documents/manual.pdf`

When you create an import with source data type `CSV` and use `abc.zip` as import resource, you can access the files for example with the following CSV file:

```
Product name;Main image;Manual
My product;image.jpg;documents/manual.pdf
```

The attribute mapping works the same as if the files would actually exist unzipped in a folder.

Importing data via PHP script

You can provide the path to a PHP script which gets used as import source. This script can either **return** the data to be imported or **return** a file path / URL where the data to be imported can be found. You can either enter the script directly as import resource in the UI or you can provide the path to the script (either relative based on the Pimcore root folder or as absolute file system path - whereas the relative paths are recommended because absolute paths may differ in different environments or when hosting changes).

You can even provide arguments to this PHP script, e.g. `import.php Supplier1 Supplier2`. In your script you can access these arguments via `$arguments`.

Importing data from Pimcore objects

Sometimes Pimcore's built-in mass-data editing is not suitable for certain requirements:

- due to max execution time you can only edit a certain number of objects at the same time
- when you want dynamic values depending on another object field
- data of some field types cannot be edited in grid view

In these cases you can choose "Pimcore" as data source to import data from Pimcore data objects / assets / documents. After choosing a source class all methods starting with "get" are provided as raw data fields. You can extend this list by [overriding the data model class](#).

Another use-case for this feature is when you want to change a data type of a class / brick / classification store / field collection field without losing existing data. You can first import existing data as raw data, then change the class / brick / classification store / field and thereafter reimport the data to the new field.

And last but not least you can use this to implement a save hook for your objects - e.g. to only publish objects which fulfill some quality criteria (has a title, has images etc.). Of course you could do this via [model overriding](#) but an import has the advantage that all things which edit an object's state can be reviewed from the Pimcore backend - while overridden model classes often are a magic black box for Pimcore users.

The objects to be used can be restricted with the **SQL condition**. You can access here

- all fields of the source class database table
- localized fields with `fieldname#<language-code>`, e.g. `name#en='example'`
- object brick fields with `<brick name>.fieldname`, e.g. `Chairs.legCount=3`
- relational fields, e.g. with source data class "Product" which has a relational field "categories" to class "Category" which has a field "name" you can have SQL condition: `categories.name LIKE 'a%'` and will only get those products which have at least one category assigned whose name begins with "a"

To create raw data from Pimcore objects you can enter a [data query selector](#).

Importing data from Pimcore reports

You can use [Pimcore custom reports](#) as import source. Just select the report which you want to import data from, select which columns of the report to use and you are done.

Importing data from Pimcore reports has especially 3 use cases:

1. You can use the [result callback function](#) to check for certain deviations in your report's data and to create a response document (e.g. text or CSV) and with the [result document action](#) send this data as an email. This way you can get notified about certain problems even without continuously looking at the reports.
2. Use custom report adapters like the Google Analytics Adapter which gets shipped with Pimcore or the adapter to import data from external databases to import data from external services / systems.
3. Export data from any Pimcore database table like Quantity Value Units, translations etc.

Exporting data from Pimcore objects

Exporting works nearly the same as [Pimcore-based imports](#). You can specify the fields which shall be exported via [data query selectors](#) - with the addition of being able to set the names of intermediate return fields.

Example: You use a `Category` class as source class for the dataport. This class contains a many-to-many relation field `products` to the class `Product`. Class `Product` contains an *advanced* many-to-many relation `images` (allowed types: assets).

```
products:each:(name;images:each as images:(Element:fullpath as path;Element:thumbnail#my-thumb
```

This would result in:

```
[
{
  "name":"Product A",
  "images": [
    {
      "path":"/path/to/asset/file1.jpg",
```

```

        "url":"/var/tmp/image-thumbnails/0/1234/thumb__my-thumbnailedefinition/file1.jpg"
    },
    {
        "path":"/path/to/asset/file2.jpg",
        "url":"/var/tmp/image-thumbnails/0/1235/thumb__my-thumbnailedefinition/file2.jpg"
    }
]
},
{
    "name":"Product B",
    "images": [
        {
            "path":"/path/to/asset/file1.jpg",
            "url":"/var/tmp/image-thumbnails/0/1234/thumb__my-thumbnailedefinition/file1.jpg"
        },
        {
            "path":"/path/to/asset/file3.jpg",
            "url":"/var/tmp/image-thumbnails/0/1236/thumb__my-thumbnailedefinition/file3.jpg"
        }
    ]
}
]

```

Together with [result callback functions](#) this can be used to individually configure exports.

When you enable the **Run automatically on new data** checkbox for export dataports, raw data will always get updated as soon as qualified objects get saved. Only the saved object gets processed during automatic raw data import, not all data objects. This way raw data is always up to date and exports become very fast as the step to fetch the desired data from the data objects is not necessary anymore in the moment of export request. You can also consider automatic exports as a CQRS system: Data to be queried gets saved in an optimized way for reading and gets fetched from the write model (Pimcore data objects) only when data changes.

Automatic exports also get triggered if an element of the dataport's target class (which matches the configured SQL condition) is referenced by the saved object.

When you enabled **Run automatically on new data** for your export dataport, another checkbox **Incremental Export** becomes available. Non-incremental exports first fetch the data from all data objects which match the configured SQL condition and thereafter update this export data when a related object gets saved. Incremental backups do not fetch the data of all matching objects. Instead they fetch the data only from the saved objects and export this data. As soon as the [result callback function](#) does not trigger any error for the exported items and does not return **false** the raw data gets automatically deleted. Then when saving other objects the whole process is repeated.

Exporting data from grid configuration

This bundle ships with a grid operator **Data Query Selector** to use [data query selectors](#) to show data from data objects in the Pimcore grid / folder view. This is especially interesting for complex fields like relations, field collections, object bricks as it is really difficult to extract the desired data from such complex fields with Pimcore's default grid operators.

All the grid columns can then be directly exported as CSV, JSON, XML without creating a dataport first (in the background an ad-hoc dataport will be automatically created).

Another option is to create a dataport manually and choose **grid configuration** as source type. The raw data fields of such a dataport cannot be changed but get dynamically fetched from the configured

saved grid configuration. So when the grid configuration gets changed, automatically also the raw data fields of the dataport get changed. In attribute mapping you can then define the export format as for usual [object-based imports](#).

Importing data from filesystem

The import type “filesystem” uses files on the server as import source (no additional CSV, XML or other meta file needed). The raw data fields result of CLI commands which get executed on the files. For example you could have a raw data field with the CLI command `md5sum "$file" | cut -d ' ' -f 1` to get the md5 checksum of the file imported file. As placeholders for the file you can use `$file`, `$filename`, `{{file}}` and `{{filename}}`.

The absolute file path gets available when you add a raw data field named `file` or `filename` - you do not need any CLI function in this case. Another option to get the absolute file path is to use `$file`, `$filename`, `{{file}}` or `{{filename}}` in the CLI command field - in this case the field name does not matter.

Dynamic import resource / parametrized imports and exports

In the import resource field you can use placeholder variables via `{{ variableName }}`. For example you could set up an import with `http://example.org/api/products/{{ itemCode }}` as import resource. You have several ways to fill this variable:

1. Call import via CLI: `bin/console dd:complete <Dataport id> --parameters="itemCode=1234"`
2. Call import via REST API POST `http://example.org/api/rest/import/dataport-name?itemCode=1234`
3. Configure the dataport to be executed automatically. When there is a field `itemCode` in the dataport's target class, and a data object of this class gets saved, the dataport will get executed with the `itemCode` of the just saved object.
4. Set environment variable `itemCode=1234` and start dataport without setting the parameter explicitly
5. Configure a [website setting](#) `itemCode` with a certain value. Localized website settings can be accessed via `{{ itemCode#en }}` (english setting of `itemCode`)

Actually the variables are [data query selectors](#), this means that you can also access object fields here, e.g. when you have separate configuration data objects where you keep your API credentials, your import resource could be `http://example.org/api/products/?apiKey={{ ApiConfiguration:path:/configuration/api/xyz:apiKey }}`. This then would try to find a Pimcore data object of class `ApiConfiguration` whose path is `/configuration/api/xyz` and from this object retrieve the content of field `apiKey`. So in the end the import resource would be `http://example.org/api/products/?apiKey=1234` if the retrieved object has `apiKey=1234`.

And as those variables are [data query selectors](#) you can also use logic operations with [Twig syntax](#). For example you could have `http://example.org/api/products/{{ itemCode|default('1234') }}` to set a default value which makes developing the import easier because you will get preview data for your raw data fields and in attribute mapping.

Another example is conditions:

```
http://example.org/api/products/
{% if APP_ENV == "prod" %}
    {{ ApiConfiguration:path:/configuration/api/xyz-live:apiKey }}
{% else %}
    {{ ApiConfiguration:path:/configuration/api/xyz-sandbox:apiKey }}
{% endif %}
```


Dynamic raw data field selector

You can use `__all` selector to get all fields of the current raw data item in one raw data field. This way you can set up dynamic imports which use differently structured import files (e.g. CSV files with different columns according to the fields to be edited), e.g. once there is a CSV with `SKU` and `name`, next time the import CSV has columns `SKU` and `price`. In this case the `__all` raw data field will contain all raw data `{SKU: 123, price: 3.99}` and can be processed in the attribute mapping callback functions.

Archive folder

When you set a Pimcore asset folder or a filesystem folder as **archive folder** the import files which get used for a certain import will afterwards get saved to that folder. This makes it is easier to analyze import errors in future. If you use a Pimcore asset folder, the archive file will be directly linked from the history panel so whenever there is an error, you can directly jump to the corresponding import file in the archive folder.

By default files older than 30 days get automatically deleted from the archive folder to not waste hard disk space. You can change this by overriding the parameter `blackbit_pim.importstatus.cleanup_interval` in your `app/config/parameters.yml`.

Raw data fields

Raw data fields define which data shall be extracted from the import resource. You can specify the data to be extracted with selector queries depending on the format of the import source:

Input format	Field selection by
XML / HTML	XPath
CSV	Column index or column heading
JSON	JMESPath / JSON Pointer
Excel	Column name or column heading
Fixed-length Files	Field length
Pimcore objects	Data query selector
Pimcore reports	Report column names
Pimcore Grid configuration	Dynamic field selection via grid configuration
Files	CLI commands


Define order of import

When processing raw data the order of raw data fields defines the order of data processing. At first the raw data item with the “lowest” value in the uppermost field gets imported. If multiple raw data items have the same value in the uppermost field, then the import order is defined by the second field, etc. - just like SQL's `ORDER BY column1, column2` feature. This is especially useful if raw data items depend on each other (e.g. master slave data - you have to import masters first to get correct hierarchy, or for category hierarchy you have to import main categories first and afterwards sub categories). You can adjust the raw data field order by drag & drop the raw data fields to the desired position.

Side note: For imports raw data gets first sorted by modification timestamp of the import files to guarantee correct import order when importing data from multiple files at once. Raw data items with the same modification timestamp are ordered by raw data fields as described above.

When you want to import the items exactly as they are in the import file, you can use the special field `__index`. When you add this as your first raw data field, the items are processed in the same order as they are in the import file.

Attribute mapping

In the attribute mapping panel you will see all fields of the dataport's configured target class or of the export document. In the column **Raw data field** you can assign a raw data field. If you want to alter the value of the raw data field during the import / export, you can add a [callback function](#) by clicking the  icon.

Modifying data via callback functions

In the settings dialog for each field in the attribute mapping panel, you can specify a callback function to modify / transform the data that will be processed during the import.

You only have to enter the function body in the callback function code editor field. All parameters passed to the function are properties of the `$params` object.

```
$params['value'] *= (1+$params['rawItemData']['tax']['value']);  
  
return $params['value'];
```

You will find a list of all available variables on the right side of the code editor.

Data passed into the callback function as argument (properties of the `$params` object)

- **value** - The raw data value as imported from the source file
- **rawItemData** - Array with all raw data values for this item. Access data by field index as specified in the data parsing configuration or by raw data field name, e.g.: `rawItemData['field_1']` or `rawItemData['name']`
- **currentValue** - The attribute value currently set in the object which got found by set **key fields**
 - class inheritance and fallback languages do **not** get used for this field
- **currentObjectData** - Array with all current values from the found object. Access data by field name from the class definition:
 - `currentObjectData['price']` for class fields
 - `currentObjectData['name#de']` for localized class fields
 - `currentObjectData['brickContainerField']['brickName']['brickField']` for object brick fields
 - If the value in the dependent field gets changed during the import, **currentObjectData** gets updated (import field order gets adjusted according to mapping dependencies - when in the callback function of field **name#en** `currentObjectData['name#de']` gets accessed, **name#de** gets imported before **name#en**)
 - class inheritance and fallback languages get used for this variable
- **keyValues** - Associative array with key values indexed by key field names, especially important when [creating / editing multiple objects from one raw data item](#)
- **request** - Request object which may contain [URL / CLI parameters](#)
- **transfer** - Object for storing data to be reused / processed on later raw item datasets, the object is initially empty

Skipping items

You can skip items by returning `null` in the callback function of a key field. If you have multiple key fields and at least for one of them `null` gets returned, the raw data item is skipped.

Importing to non-scalar field types

For most non-scalar fields this bundle provides templates which you just have to select. Only for more complex logic you would need to edit the callback functions after selecting a template.

Object hierarchy / parent element

With the attribute mapping for field `Path` you can specify the parent element of the imported element. You have multiple options to specify the parent element:

- via [data query selector](#), e.g. with callback function `return 'Category:category_id:'.$params['value'];`
- via string (or by just assigning a raw data field without callback function):
 - if this string's first character is not a `/` then the configured target folder for this dataport will get prepended (mapped path is relative to configured target path), e.g. you return `fashion` and the dataport has `/categories` as target folder, then the imported object will be put under `/categories/fashion`
 - if it starts with a `/` then the configured target path gets ignored (absolute path)

If no object is found, then it will be created. If you use a data query selector, then an object of the given class (the part before the first colon in the data query selector) will be created. This object has as the only filled field the filter field used in the data query selector. This way it will get found again if you repeat the import or if also other raw items refer to this object by data query selector. When you later import the real object the auto-created object gets updated. If you return a string (and not a data query selector) and there isn't any folder with this path, then it will get created.

Single object relation (Href / Many-to-one relation)

Manual Assignment To assign objects based on raw data it is necessary to provide some information how to find these objects. In addition to the raw data field value Pimcore needs to know of which class the desired object is and which field to use for querying, example:

```
return 'ClassName:fieldName:'.$params['value'];
```

This code tries to find an object of class "ClassName" which has `fieldName` = imported raw data field.

It is also possible to find objects by a value in a localized fields:

```
return 'ClassName:fieldName#locale:'.$params['value'];
```

If there are colons within the query value, you should wrap the value with quotes:

```
return 'ClassName:fieldName:'.$params['value'].'"';
```

Automatic assignment With the checkbox **Automatically assign** you can automatically assign elements to the relation field based on artificial intelligence. The training data consists of the data of all mapped fields of this dataport of all published objects of the same class as the imported object. This can e.g. be used to automatically assign categories of a product based on the product name and description.

Many-to-many object relation

Assigning data to a many-to-many object relation works the same as single object relation but you have to return an array.

```
return [
  'ClassName:fieldName:'.$params['value'],
  'ClassName:fieldName:'.$params['rawItemData']['otherRawDataField']['value']
];
```

Object relation with metadata (Advanced many-to-many object relation)

The definition how to find the object to be assigned works the same as with normal many-to-many object relation with the only difference that the return value has to contain the object query string in the field **query**. Other meta columns can be set via key-value pairs within this JS object. Example:

```
return [
  [
    'query' => 'Person:email:'.$params['value'],
    'metadataFieldName' => $params['rawItemData']['rawDataColumn']['value']
  ]
];
```

This adds a relation to an object of class **Person** which has the given raw data value in its field **email**. Additionally the metadata column **metadataFieldName** is set to the value of raw data column **rawDataColumn**.

Asset relation (Image, image gallery, Many-to-many relation)

To assign an asset to an image / image gallery / many-to-many relation field you have to provide an array of file paths or URLs under which the assets are accessible. The importer tries to load it and imports the asset to the target folder (configurable under dataport settings). The structure of the array to be returned follows the same logic as for [image fields](#), e.g.

```
$images = explode(',', $params['value']);
return array_map(
  function($imageUrl) {
    return [
      'url' => $imageUrl,
      'filename' => 'somePrefix_'.basename($imageUrl)
    ];
  }, $images
);
```

If you want to search for existing assets, you can provide a [data query selector](#) instead of the **url**:

```

$images = explode(',', $params['value']);
return array_map(
    function($imagePath) {
        return [
            'query' => 'Asset:path:'.$imagePath
        ];
    }, $images
);

```

Image field

```

return $params['rawItemData']['imageUrl']['value'];

```

The given image file is tried to be found in the *configured asset source folder*. But you can also return absolute path to the file.

You can also retrieve files via URL:

```

return 'https://example.org/'.$params['rawItemData']['imageUrl']['value'];

// or, if you want to save the file under a different name or folder:
return [
    'url' => 'https://example.org/'.$params['rawItemData']['imageUrl']['value'],
    'filename' => $params['rawItemData']['name']['value']
];

```

Providing a `filename` does also support dynamic folders. So when you provide `/images/abc.jpg` in the `filename` parameter, the asset will get the name `abc.jpg` and will be created in / moved to the asset folder `/images`. When you provide a relative path like `images/abc.jpg` the image will be put in the folder `images` below the dataport's configured asset target directory.

You can also assign already existing Pimcore assets via:

```

return [
    'query' => 'Image:path:'.$params['value']
];

```

It is also possible to find assets by custom metadata:

```

return [
    'query' => 'Image:external_system_id:'.$params['value']
];

```

`external_system_id` is a custom metadata field of the asset.

Optionally you can return other fields than `query`, `url` and `filename` - those will be assigned as *custom metadata* to the image asset, e.g.

```

return [
    'url' => $params['value'],
    'product_sku' => $params['rawItemData']['sku']['value'],
    'product_name' => [
        'de' => $params['rawItemData']['name de']['value'],
        'en' => $params['rawItemData']['name en']['value'],
    ]
];

```

This example would result in the asset being created from the filename in the assigned raw data field. To this asset the `custom metadata product_sku` gets set as well as the localized custom metadata fields `product_name`. This way you can for example find the asset by searching for the product name.

Image gallery

Assigning images to image gallery fields works the same as for `images` - except that you can return multiple items in an array, e.g.:

```
return [
    $params['rawItemData']['image 1']['value'],
    $params['rawItemData']['image 2']['value'],
];
```

or as a more complex example:

```
return [
    [
        'url' => $params['rawItemData']['image 1']['value'],
        'filename' => $params['rawItemData']['sku']['value'].'_1.jpg',
        'product_sku' => $params['rawItemData']['sku']['value'],
    ],
    [
        'url' => $params['rawItemData']['image 2']['value'],
        'filename' => $params['rawItemData']['sku']['value'].'_2.jpg',
        'product_sku' => $params['rawItemData']['sku']['value'],
    ]
];
```

Many-to-many relation fields

For many-to-many-relation fields the same format as for image fields applies but you can provide multiple entries in an array:

```
return [
    'https://example.org/'.$params['rawItemData']['imageUrl_1']['value'],
    'https://example.org/'.$params['rawItemData']['imageUrl_2']['value']
];

// or, if you want to save the file under a different name:
return [
    [
        'url' => 'https://example.org/'.$params['rawItemData']['imageUrl_1']['value'],
        'filename' => $params['rawItemData']['name']['value'].'_1.jpg'
    ],
    [
        'url' => 'https://example.org/'.$params['rawItemData']['imageUrl_2']['value'],
        'filename' => $params['rawItemData']['name']['value'].'_2.jpg'
    ]
];
```

Asset relation with metadata / Advanced many-to-many relation

To assign an asset together with some metadata you have to return an array of objects. The asset filename / URL has to be in field `url` of this object. To find already existing assets you can use the field `query` (as described above for asset relation fields). Metadata fields can be set with key-value pairs in the returned object. Example:

```
return [
  [
    "url" => $params['rawItemData']['image-url-field']['value'],
    "metaField1" => "ABC",
    "metaField2" => $params['rawItemData']['rawDataColumn']['value']
  ]
];
```

If you want to change the filename of the asset which gets created by the import you can use the field `filename`:

```
return [
  [
    "url" => $params['rawItemData']['image-url-field']['value'],
    "filename" => "my-new-file.jpg"
  ]
];
```

It is also possible to retrieve existing Pimcore assets via:

```
return [
  [
    'query' => 'Image:path:'. $params['value']
  ]
];
```

For asset querying same rules apply as for object querying: `<Asset class>:<Field to filter by>:<Filter value>`.

Video

Pimcore's video field type supports different sources of the video. In general videos can be imported via

```
return [
  'type' => 'asset', // or youtube, vimeo, dailymotion
  'data' => 'path to file / URL' // or video ID
];
```

This works the same as for [image fields](#).

You can also provide the URL of the video, then it will automatically get parsed, e.g. you can provide strings like: - `https://youtu.be/nyhKJTzTq-4` -> will get parsed as type "youtube" and data "nyhKJTzTq-4" - `https://another-pimcore.org/var/assets/my-video.wmv` -> will load the video from the remote system and create it as a local asset

Quantity value / Input quantity value

To fill a quantity value field you have to return an array with the value and the unit abbreviation in the callback function. Alternatively you can return a string with value and unit separated by a whitespace. For example:

```
return [$params['value'], 'mm'];  
return $params['value'].' mm';
```

If the given unit does not exist yet, there is an option to automatically create units. If possible, the unit gets created with base unit and conversion factor / offset - ready to support [automatic unit conversion](#).

If you do not return an array or set the second array item to an empty value, the default unit of the field gets used (if configured in the field definition).

Field collections

Field collection values can be assigned by providing a JSON array with objects whose keys represent the fields of the field collection, for example:

```
return [  
    'fieldCollectionName' => [  
        [  
            "field1" => $params['rawItemData']['field_1']['value'],  
            "field2" => $params['rawItemData']['field_2']['value'],  
        ],  
        [  
            "field1" => $params['rawItemData']['field_3']['value'],  
            "field2" => $params['rawItemData']['field_4']['value'],  
        ]  
    ]  
];
```

The importer tries every field collection type which is allowed in the target class field. If it encounters a field returned by the callback function which does not exist in the field collection it tries the next allowed field collection type.

Field collection items only get added when at least one field is filled and if no other currently existing item has the exact same field values. In cases where you want to keep some of the current items while updating others, please enable **Truncate before import** checkbox and then loop through currently assigned field collections items via `$params['currentValue']`. The ones which you want to keep, you put into an array. To this array you also add the ones which the import shall create. Then you return the array and have your desired items in the field collection.

Localized fields within field collection

```
return [  
    'fieldCollectionName' => [  
        [  
            "field1" => $params['rawItemData']['field_1']['value'],  
            "field2" => $params['rawItemData']['field_2']['value'],  
            "field3_localized#de" => $params['rawItemData']['Text_DE']['value'],  
            "field3_localized#en" => $params['rawItemData']['Text_EN']['value']  
        ]  
    ]  
];
```

```

]
];

```

Asset relation within field collection

```

return [
  'fieldCollectionName' => [
    [
      "many-to-many_relation_field" => [
        "Image:path:/path/to/image/" . $params['value']
      ]
    ]
  ]
];

```

Asset / object relation in localized field in field collection

```

return [
  'fieldCollectionName' => [
    [
      "many_relation_field#de" => [
        "Image:path:/path/to/image/" . $params['value']
      ],
      "many_relation_field#en" => [
        "Image:path:/path/to/other/image/" . $params['value']
      ]
    ]
  ]
];

```

Quantity value field within field collection

```

return [
  'fieldCollectionName' => [
    [
      "quantity_value_field" => [[123, 'EUR']]
    ]
  ]
];

```

Blocks

Block field values can be imported with a key-value array:

```

return [
  [
    'field1' => 'first block item, value for field1',
    'field2' => 'first block item, value for field2',
    'relationalField' => 'PriceList:customerNo:1234'
  ],
  [
    'field1' => 'second block item, value for field1',
    'field2' => 'second block item, value for field2',

```

```

        'relationalField' => 'PriceList:customerNo:2345'
    ]
];

```

The provided values are parsed through the corresponding field type interpretation. For this reason assigning an object to the field `relationalField` via a data query selector works in above example.

Object bricks / Classification Store

All the single fields of the allowed object bricks / classification store groups get displayed in the attribute mapping panel and can be mapped individually. You can treat them like normal class fields so all the things listed here for other field types are valid also for object brick / classification store fields. As soon as at least for one of the fields of an object brick / classification store group data gets imported the brick / group will get added to the current object.

Alternatively you can directly map your data to the object brick / classification store container field by returning an associative array whose keys are the brick / classification store group names:

PHP:

```

return [
    'brickName' => [
        'field1' => $params['rawItemData']['field_1']['value'],
        'field2' => $params['rawItemData']['field_2']['value'],
        'name#de' => $params['rawItemData']['name de']['value'], // name is a localized field
        'name#en' => $params['rawItemData']['name en']['value'],
    ],
];

```

Convenience features like resolving data query selectors are applied to all the provided fields, e.g.

```

return [
    'technicalAttributes' => [
        'cableLength' =>
            'AttributeValue:externalId:'.$params['rawItemData']['id']['value'].':value'
    ],
];

```

This will search for an object of the class `AttributeValue` whose `externalId` equals the value from the raw data field `id` and import this to the field `cableLength` of the object brick `technicalAttributes`.

Or if the field name is part of the import data you can assign it dynamically:

```

return [
    'technicalAttributes' => [
        $params['rawItemData']['attributeName']['value'] =>
            $params['rawItemData']['attributeValue']['value']
    ],
];

```

If a field is mapped in the object brick / classification store container field *and* as a single field, only the single field mapping will get applied.

If a field is mapped as a single field but the container mapping does not return the brick / classification store group name of the mapped single field, the single field will get skipped.

Example:

Mapping for container field returns:

```
[
  'Shoes' => [
    'size' => 43,
    'color' => 'blue'
  ]
]
```

When the field `Shoes/size` is not mapped as a single field, the value from the container field will get used.

When there is a single field mapping for `Shoes/color`, this single field mapping will get used instead of the value from the container field. This way you can set import options like `Automatically create missing options` if `color` is a select field.

When there is a mapping for `Clothing/color` (= for a field of another brick / classification store group), this will get skipped because the callback function for the object brick / classification store container does not return brick / group `Clothing`. Of course this only applies if the container field is mapped. If you only map single object brick / classification store fields, they will get applied anyway.

Links

The data of a `Link` field can be set by returning an array whose keys refer to the properties of the `Pimcore\Model\DataObject\Data\Link` class, for example:

```
return [
  'path' => 'http://example.org/page',
  'target' => '_blank',
  'text' => 'Link text'
];
```

Geographic Point

To set a value to a geographic point field you have to provide the geographic coordinates of the point. Several formats are supported, e.g.

- 40:26:46N,079:56:55W
- 40:26:46.302N 079:56:55.903W
- 40°26'47"N 079°58'36"W
- 40d 26' 47" N 079d 58' 36" W
- 40.446195N 79.948862W
- 40.446195, -79.948862
- 40° 26.7717, -79° 56.93172

Also geocoding is supported. When the bundle does not recognize the given value as valid coordinates it assumes that it is an address and tries to convert this to coordinates. This is done either with

- [Google Maps Geocoding Service](#). For this to work you have to provide an [API key](#) in `app/config/config.yml` (Pimcore <= 6) or `/config/config.yaml` (Pimcore >= 10):

```
pimcore:
  services:
    google:
      simple_api_key: <Your API key>
```

- or with [Nominatim Open Street map geocoding service](#). This service is free and usable without credentials or an API key but [limited under some circumstances](#).

Before Pimcore 10 you can also set this value in the Pimcore backend under **System settings**. [Information how to get a Google Maps Geocoding API key](#).

Select / Multiselect fields

To assign a value to a select field you should return the option's value. If no option with the given value can get found, it is tried to find an option with the given label.

For multiselect fields the same logic gets applied but you have to provide the option separator. Or if you use a callback function, you have to return an array of values:

```
return ['option 1', 'option 2'];
```

The same applies for all field types which extend select or multiselect like User, Language, Country, Country (Multiselect) etc.

If you enable the checkbox **Automatically create missing options** in the attribute mapping settings, new options will automatically get created. For example this makes sense when another system is the leading system for the available options of this field.

URL slug

For URL slug fields you can either return a string (or only design a raw data field without any callback function) to assign the URL slug for all sites.

Alternatively you can return an array with the key being one of the configured domains of a site and the value being the URL slug:

```
return [
    'example.org' => '/en/my-product',
    '' => '/my-product' // fallback URL slug for all sites without a specific value
];
```

Automatic translation

In attribute mapping there is an option for automatic translation of text fields (Input, Wysiwyg, Textarea). You can assign a raw data field with a certain text and define its language in the text field mapping options to get the raw data text automatically translated to the target language of the localized field (for objects) or the language of the document (for document imports).

Translation Provider Configuration

DeepL You will need an [DeepL API key](#) to use this feature. Please add your API key to `app/config/parameters.yml`:

```
parameters:
    ...
    blackbit_pim.deepl_api_key: <Your API key>
```

The DeepL API supports to ignore certain words for translation. This can be useful for brand names, product names or other phrases which shall not be translated. By default you can mark such phrases by wrapping them into `<x>` tags. You can change the tag for ignored phrases by overriding the parameter `blackbit_pim.skip_translation_tag` in your `app/config/parameters.yml`:

```
parameters:
    ...
    blackbit_pim.skip_translation_tag: dfn
```

Afterwards clear the cache (via Pimcore UI or `bin/console cache:clear`).

Amazon (AWS) Translate To use Amazon Translate you need to [set up your AWS credentials](#) - if you decide to configure credential information inside Pimcore's configuration files, please put your access key and secret in `app/config/parameters.yml`:

```
parameters:
    ...
    blackbit_pim.aws_translate_access_key: <Your access key>
    blackbit_pim.aws_translate_secret_key: <Your secret key>
```

Afterwards clear the cache (via Pimcore UI or `bin/console cache:clear`).

Exclude terms from translation / Manually translate phrases

When you want to exclude some phrases from translation or are not satisfied with the translation, you can exclude phrases from being translated or provide custom translations. To achieve that you have to create translations in [Pimcore's shared translations](#) whose key starts with `translate`. (the rest of the key does not matter for the Data Director, use whatever you want). The entered translation in the source language will trigger exclusion / manual translation:

- When in the target language no translation is set, the phrase will be excluded from translation, so the original phrase is kept.
- When there is a translation in the target language, this will get used (internally the phrase in the source language gets ignored and after translation it gets replaced with the custom translation from Pimcore's shared translations)

Translate complex data / Manipulate translated content

In some cases it might become necessary to translate certain strings via a function call. For example this is useful for complex datatypes like when you want to translate asset metadata or field collection data. This way you can implement your custom logic for filtering which fields have to be translated and then call the `\Blackbit\DataDirectorBundle\lib\Pim\Item\Importer::translate` function:

```
// Translate asset metadata from german to other languages

use \Blackbit\DataDirectorBundle\lib\Pim\Item\Importer;

$sourceLanguage = 'de';
return array_map(function($value) use ($sourceLanguage) {
    if($value['language'] === $sourceLanguage) {
        $newMetaData = [];
        foreach(\Pimcore\Tool::getValidLanguages() as $targetLanguage) {
            if($targetLanguage === $sourceLanguage || $value['type'] !== 'input') {
```

```

        continue;
    }

    $translation = Importer::translate(
        $value['data'],
        $targetLanguage,
        $sourceLanguage
    );

    $newMetaData[] = [
        'name' => $value['name'],
        'data' => $translation,
        'type' => $value['type'] ?? 'input',
        'language' => $targetLanguage
    ];
}
return $newMetaData;
}, $params['value']));

```

You can also use this function when you want to manipulate the translation afterwards, e.g. uppercase first letter of the translation for product names:

```

use \Blackbit\DataDirectorBundle\lib\Pim\Item\Importer;

$targetLanguage = 'de';
$sourceLanguage = 'en';

$translation = Importer::translate($params['value'], $targetLanguage, $sourceLanguage);
return ucfirst($translation);

```

Map languages

Normally it is tried to use Pimcore's configured language including locale. If the translation provider does not support the given locale, it will fall back to the main language.

But when for instance in Pimcore **English** got added as language, it is not clear which locale shall be used for translation. In such a case you can map Pimcore languages to translation target languages in `config/config.yaml` (or `app/config/config.yml` on Pimcore versions ≤ 6):

```

blackbit_data_director:
    translation:
        languages:
            en: 'en-gb'

```

Text generation

Rule-based

There are 2 ways to generate texts based on rules:

1. Set up an automatic Pimcore-based import with the fields, which you want to use in your text generation rules, as raw data fields. In the callback function of the target field you can generate the text.

2. Add a field of type “Textarea / WYSIWYG field with variables”. This allows to access the content of other fields directly from the data object editing view. It supports logic operations (like conditions, loops etc.) via [Twig template engine syntax](#). The rules may be inherited to child objects, so you can have a different set of rules for different sub-trees.

Artificial Intelligence

For textarea and wysiwyg fields there is a checkbox “Automatically generate” in the field settings in Data Director attribute mapping. With this you can connect to a text generation API (like OpenAI). The API needs some input to know what to do - the more data you provide, the better the API knows what to do.

Infer technical data from texts After creating the target state in your data model, it is often a lot of work to extract this information for your existing data. Artificial intelligence can help to extract information for single fields from textual information.

Example:

Input: > * Product name: Wrangler Men’s Casual Shirt > * Product description: Long-sleeved cotton shirt with navy and orange stripes

Result: > * Material: cotton > * colors: orange, blue (because the **colors** field does not allow **navy** but it allows **blue**) > * gender: male

This data can now be used for filters or structured exports.

Create texts based on technical data Especially manufacturers often have the challenge that they know all details of their products exactly but struggle with creating good texts. But also for distributors / resellers it is a big challenge to create good texts because multiple distributors get the same texts from a certain supplier - so how to stand out?

Example:

Input: > * Name: Wrangler Men’s Casual Shirt > * Material: cotton, > * Colors: Blue, Orange > * Gender: Male

Output: > The Wrangler Men’s Casual Shirt is a comfortable and stylish choice for any man. Made from high-quality cotton, it offers breathable comfort that will keep you feeling great all day long. Available in two vibrant colors - blue and orange - this shirt is perfect for casual wear or dressier occasions. Designed with the male physique in mind, it provides a flattering fit that looks great on every body type. Whether you’re heading to work or out with friends, the Wrangler Men’s Casual Shirt has got you covered!

Modify the prompt Behind the scenes to configured input data in only one part of the prompt. Some context information like class name, object name etc. will automatically get added. Also limitations (e.g. select options, length limit) get added automatically.

If you prefer to create the prompt on your own, you can add **<no-prompt-extension>** together with your custom prompt in the callback function.

Tags

You can import one or multiple [tags](#). Tag levels should be separated by /. You can either just assign a raw data field or use a callback function like this

```
return ['Tag-Name','hierarchical/tag'];
```

Tags which do not exist yet, get automatically created.

Properties

You can import [properties](#) by returning a key-value array:

```
return [  
  'Name' => 'Product A',  
  'imported' => time(),  
  'product name' => 'MyDataObjectClass:path:/path/to/object:fieldA', // data query selectors g  
  'product' => 'Product:sku:'.$params['rawItemData']['sku']['value'] // for relations correct  
];
```

This will end up in the 3 properties being assigned to the imported element.

Alternatively you can provide all supported property attributes:

```
return [  
  ['name' => 'parameter name', 'data' => 'Parameter value', 'inheritable' => true],  
  ['name' => 'parameter 2', 'data' => '123', 'inheritable' => false]  
];
```

Dependency graphs

When a certain field gets changed by multiple dataports it can be difficult to keep track of all the dependencies. In this case it might be helpful to visualize all the dataports which use a certain field. This can be accessed by clicking the field name in the attribute mapping panel, the result looks like this:

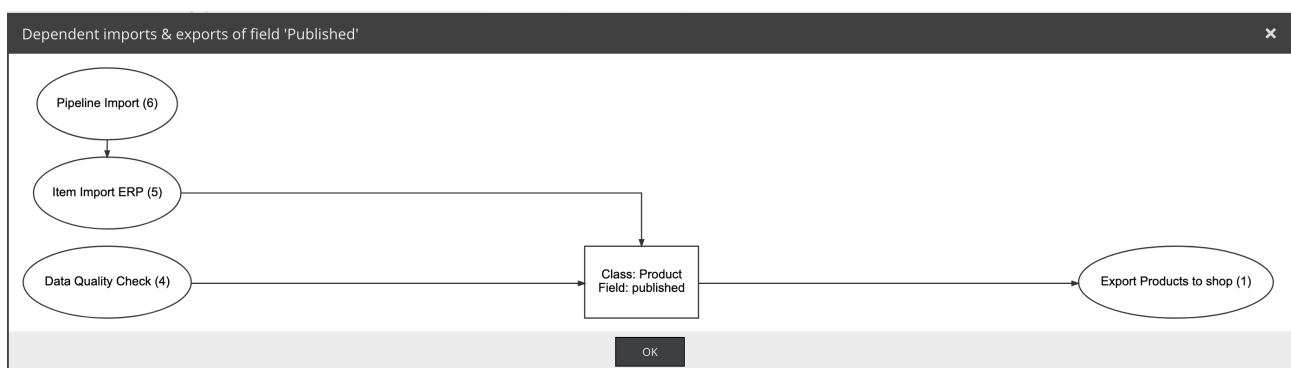


Figure 0.1: Dependency graph for field published

In this example the dependency graph for the field `published` of class `Product` is shown. In this case 2 dataports (`Item Import ERP`, `Data Quality Check`) import to the field `published`. `Item Import ERP` might also be called as a dependent import of dataport `Pipeline import`. On the right side we see that the dataport `Export products to shop` accesses the field `published`.

Import pipelines

In the [result callback function](#) there is a template for setting up a dependent dataport which can be called during a run of the current dataport - either after every raw data item or at the end. If necessary you can provide [parameters](#) to be used in the import resource or callback functions of the follow-up dataport.

Edge cases

Importing data to objects whose newest version is unpublished

For objects with their newest version being unpublished for every raw data item import gets executed twice:

1. Update currently published version und create a new published version
2. Update newest currently unpublished version

This way the unpublished changes do not get overwritten while the object data from the import gets applied for both versions.

Creating / Editing multiple objects with one raw data item

Returning an array for a key field will be treated as if there were multiple raw data items with those key field values.

Example: You have an XML import with products:

```
<products>
  <product>
    <name>Product A</name>
    <colors>
      <color>red</color>
      <color>blue</color>
    </colors>
  </product>
</products>
```

With item XPath `product` and raw data field `color` having the XPath `colors/color` you would get an array in the raw data field `color`. This shall be your key field for objects of the class `Color`. In this case the array in the key field `color` will be treated as if there were two raw data items - one with `color: red` and one with `color: blue`. Consequently this will result in two `Color` objects being created - one for each color.

If you have multiple key fields and one or more return arrays, all combinations will get created.

Example: If above XML import had another key field `size` with the values `['S','M','L']` the import would create objects for all combinations:

1. color: red, size: S
2. color: red, size: M
3. color: red, size: L
4. color: blue, size: S
5. color: blue, size: M
6. color: blue, size: L

Data Query Selectors

Syntax

Data query selectors simplify fetching other objects (e.g. for assigning them to relational fields) and fetching other objects' data. To fetch an object you can use the following expression:

```
Class:filterField:filterValue
```

This means that you look for an element of class `Class` which has the value `filterValue` in its field `filterField`. In reality this could be `Product:sku:1234` - this returns the `Product` object which has `sku=1234` - if this exists.

When you want to fetch data of the same or another object you can use the following syntax:

```
Class:filterField:filterValue:dataField
```

This would query a data object of class `Class` which has `filterField = filterValue`. From the found object you get the value in the field `dataField` returned. If the getter method needs parameters, those can be provided as follows:

```
Class:filterField:filterValue:dataField#argument1,argument2
```

You can even access data of other objects within larger text snippets. With the following callback function you will get the text "Look at these cool car products" - if the corresponding raw data item refers to the car category.

```
Look at these cool {{ Category:key:". $params['rawItemData']['categoryKey']['value']+":name#de
```

As you can see in this example to get the german name (`name#de`) getter arguments can be provided comma-seperated via suffix `#argument1,argument2,...`

Query with wildcard value

You can also use wildcard placeholders at the beginning or end of the `filterValue`, e.g.

```
// will find Product whose name begins with "car" like "car red" and "card game"
Product:name:car*
```

```
// will find a product whose name ends with "bike" like "Trekking Bike" (case does not matter)
Product:name:*bike
```

```
// will find a product whose name contains "late" like "Chocolate Cake", "Chocolate", "Late Ma
Product:name:*late*
```

Query in subset

The `filterValue` can also be a comma-separated list of values. In this case an `IN (...)` SQL will get executed. To enable this mode, wrap the `filterValue` in square brackets like

```
// will find Product whose name begins with "car" like "car red" and "card game"
Product:id:[123,234]
```

Data query selector chaining

You can chain these data selectors as long as you want. For example `Product:articleNo:123:crossSellingProducts` first searches for an object with `articleNo=123`. From this object the first item of the `crossSellingProducts` many-to-many relation gets fetched. From this object we fetch the assigned manufacturer object (many-to-one relation) whose german name finally gets returned.

It is also possible to apply PHP functions to the returned values, for example to strip HTML tags from a wysiwyg field:

```
Class:filterField:filterValue:wysiwygField:strip_tags
```

This way the content of the wysiwyg field gets piped as first argument to the `strip_tags` function. If you do not want to provide the value as first parameter or want to call a function with multiple parameters, you can use `%s` as placeholder for the value and comma as parameter separator. For example to get the first 100 characters of a text field:

PHP:

```
Class:filterField:filterValue:textField:substr#%s,0,100
```

If you want to use a comma as argument, please enclose it with quotes: PHP:

```
Class:filterField:filterValue:multiSelectField:implode#"","%s
```

Also, those function calls can be chained, for example to strip HTML tags and afterwards convert line breaks to `
` you could use:

PHP:

```
Class:filterField:filterValue:wysiwygField:strip_tags:nl2br
```

For more complex tasks you can also call your own service methods from a data query selector:

```
// via Fully qualified class name
```

```
Class:filterField:filterValue:imageGallery:items:\AppBundle\Service\MyService::methodName
```

```
// via Symfony service
```

```
Class:filterField:filterValue:imageGallery:items:@MyService::methodName
```

Grouping fields

When you want to fetch values from multiple fields you can use the following data query selector: PHP:

```
Class:filterField:filterValue:(dataField1;dataField2)
```

This can especially be useful for exports.

Field aliases

For exports it sometimes is useful to define an alias for a value which got queried with a complex data query selector. PHP:

```
Class:filterField:filterValue:(category:image:thumbnail#mainImage as thumbnail)
```

This way you can access the fetched category image thumbnail with `$params['value']['thumbnail']` instead of `$params['value']['category:image:thumbnail#mainImage']`

Placeholders within data query selectors

It is possible to access data from the request or the source class object and use it as parameter for the data query selector.

For instance when you set up a read API for thumbnails you can create an assets-based dataport with a raw data field whose data query selector is `thumbnail#{{ definition }}:path`. The thumbnail definition to be used can be provided via URL parameter `definition`. Of course this also works with environment variables, Pimcore website settings or CLI parameters. Default values can be provided via [Twig syntax](#) `thumbnail#{{ definition|default('default') }}:path` - in this case if parameter `definition` has not been provided the thumbnail definition `default` will get used.

You can even access data via data query selectors, e.g. `myMethod#{{ sku }}`. This would call method `myMethod` with the content of the field `sku` of the currently processed source class object. URL / CLI / env parameters have higher priority than data object fields, so when in this case a parameter `sku` gets provided, this will get used instead of the data object field. Chained dynamic data query selectors like `myMethod#{{ categories:name }}` are also possible.

This way you can also find objects with the same value as the current object in a certain field via `Class:supplierSku:{{ supplierSku }}:each:(images)`. Such selectors can be useful for golden record use-cases.

Shortcuts

For convenience there are also some shortcut versions for data query selectors:

- if you enter `.` as `Class` the import's target class gets used
 - e.g. `Product:articleNo:123` is equal to `.:articleNo:123` for imports with target class `Product`
 - this can lead to the problem that there is a field `Product` in the target class, then it is not clear if `Product:articleNo:123` means:

- * fetch content of field **Product** and from the result fetch field **articleNo** and following data query selector chain or
- * fetch **Product** objects which have **articleNo = 123**
- * -> in such cases where there is a target class field as well as a class with the name before the first colon, the first of above examples has priority. If you want to query data like in second case, you can use **Product:getByArticleNo:123**
- if you enter a **.** for **filterField** and **filterValue**
 - the currently imported object gets used
 - * e.g. **Product:....name:de** is equal to **Product:articleNo:123:name:de** when the import currently processes a **Product** object with **articleNo=123**
- **\$** is a shortcut for **....**:
 - e.g. you can use **\$name** in a text and get this replaced by the content of the field **name** of the current object.
 - * **\$categories:0:name** fetches the name of the first assigned object in the relation **categories** of the object which currently gets imported.
- you can fetch data from reverse-related objects via **Category:products:..** - for an export with source class **Product** this will find corresponding **Category** objects which contain the currently processed **Product** object in the relation **products**. Combine it with **each** to get data from all found reverse-related objects, e.g. **Category:products:..each:(name)** will return an array with the names of all category objects which refer to the current product in their man-to-many object relation field **products**.

Example: For a Pimcore-based import you can access **....:name** as raw data field to get the name of the object being currently imported. You could for example use this as placeholder variable in a description text template which gets filled by a Pimcore-based import. Alternatively you can use the even shorter version **\$name**.

Helpers

Get label from select field option

If you want to access the label of a select / multiselect field option you can access this via **:label** suffix, for example **Product:articleNo:123:mySelectField:label** whereas **Product:articleNo:123:mySelectField:** and **Product:articleNo:123:mySelectField** both return the selected option id. To access the translated option label you can call **Product:articleNo:123:mySelectField:label#de** (or any other locale) - this accesses the admin translation of the label.

Handle arrays

Loop through all items Array items can either be processed individually by index via **myManyToManyRelation:0:name**, **myManyToManyRelation:1:name** or via the **each** helper: **myManyToManyRelation:each:(name)**. This fetches the **name** of all items in the relation field and returns them as an array. You can also chain the data query selector after the **all** selector, for example **myManyToManyRelation:each(manufacturer:name)** (beware the braces).

Taking this concept further you can also group multiple data query selectors by using **myManyToManyRelation:each:(articleNumber;name;manufacturer:name)**. This returns the following (here written in JSON notation):


```
[
  {
    "articleNumber": 123,
    "name": "Product A",
    "manufacturer:name": "Manufacturer A"
  },
  {
    "articleNumber": 234,
    "name": "Product B",
    "manufacturer:name": "Manufacturer B"
  }
]
```

This can be useful for XML or JSON exports with nested tags (the raw data field with above data query selector is named `products` in this case):

```
<item>
  <other_field>any value</other_field>
  <products>
    <item>
      <articleNumber>123</articleNumber>
      <name>Product A</name>
      <manufacturer_name>Manufacturer A</manufacturer_name>
    </item>
    <item>
      <articleNumber>234</articleNumber>
      <name>Product B</name>
      <manufacturer_name>Manufacturer B</manufacturer_name>
    </item>
  </products>
</item>
```

Access previous version's data

before With the `before` helper you can access data of the previous version of an object, e.g. `before:published` will return if the object was published in the version before the currently published one. This can be used to trigger actions when certain fields change, for instance you could set up an automatic dataport with condition `o_published=1` and raw data field `before:published`. As soon as someone publishes an object (so that SQL condition `o_published=1` is true) which was not published before (if this is the case you see in the raw data field with data query selector `before:published`), you can now trigger some action.

You can fetch another version than the next-to-last version via `before#2024-01-01`. This becomes especially handy for workflows where the last review date is stored in a `date` or `datetime` field. For example for a review date field named `reviewedAt` the data query selector `before#{ reviewedAt }` will fetch the version which was valid at that date.

diff For workflows it may be useful to know exactly what has changed between two versions. The `diff` data query selector helper outputs the difference of a field value. In combination with the `before` helper this is really powerful: For example `before:diff#name` will return the diff of the field `name` between current and the version before as HTML code

Old

Name

New

Name

changes The `changes#<fieldName>` selector can be used to ompare the current value of a certain field to a previous time. For example `changes#name` will return a JSON object

```
{
  "date": "2023-09-14 13:49:31",
  "oldValue": "ABC123",
  "newValue": "ABC123-DEF",
  "user": "john.doe",
  "versionId": 123
}
```

Localized fields are supported via `lastChange#name#de`.

To only retrieve changes after the object got reviewed last time, please store the review date into a `date` or `datetime` field `reviewDate` (or any other name but use the same name in the following data query selector) and then you can use `changes#name,{ { reviewDate:timestamp } }`.

Group fields Aliasing field names does not only work with the `each` helper but also when you want to extract data from multiple fields in one raw data field: `description as html;description:strip_tags as text` returns for example

```
{
  "html": "Our <b>new</b> product",
  "text": "Our new product"
}
```

It is also possible to fetch multiple fields and assign an array function to this group, e.g. `(title;description):implode#
,%s`. This would fetch the title and description of the object and concatenate them using an HTML line-break.

Enable / Disable inheritance for single fields With the helper functions `withInheritance` / `withoutInheritance` it is possible to enable / disable inheritance for single data query selectors / raw data fields. The set status stays active until the opposite function gets called or the processing of the current data query selector is finished.

For example the data query selector `withoutInheritance:sku` would fetch the SKU of the product object. If the field of the current object is empty and the dataport and data object class support inheritance and the parent object has `sku` filled, you would normally get this inherited id which makes it difficult to tell if the given `sku` is from the current object or inherited from one of its parent. With `withoutInheritance` you can disable inheritance this way and really get the empty string.

Translations

With the `translate` helper function you can translate terms in data query selectors. This can be used to translate fields which are not in localized fields. You could for example use this to translate an object's key with the data query selector `key:translate`. This will translate the key of an exported element to the requested language. If you want to always translate the item to the same language regardless of the requested language you can provide the target language with `#<locale>` suffix - in above's example this would be `key:translate#de` to export the object key translated to german.

First it is tried to find an admin translation in the requested language. If none exists, shared translations get used. If neither exists the original value gets returned.

Get all languages for localized field

While you can request a specific language for example with the data query selector `name#en`, you can get the value of the field in all configured languages via `name#all`. This will give you

```
{ "de": "Fahrrad", "en": "Bicycle" }
```

Filtering

Arrays can be filtered with the `filter` helper. It supports 2 or 3 parameters: `categories:filter#published,1` will return an array of the assigned `Category` objects which are published. You can filter for any data object or system field. The third parameter supports comparison operator, e.g. for a field collection `prices` with a date field `validFrom` you can use `prices:filter#validFrom,now,>=` to get all prices which are valid today or in future (`now` will be evaluated as current date).

After filtering it may be useful to check if the returned array is empty or not. This can be done with PHP's `empty` or with `exists`:

- `categories:filter#published,1:empty` will return 1, if there is no published category assigned, otherwise 0.
- `categories:filter#published,1:exists` will return 1, if there is at least one published category assigned, otherwise 0.
- `categories:filter#published,1:empty:not` is equivalent to `categories:filter#published,1:exists`

Get workflow state

You can get the workflow state of an object by using the expression

- `workflowState#<Workflow-Name>:places:key` for workflows where an object can only be at one state at the same time or
- `workflowState#<Workflow-Name>:places:each:key` for workflows where an object can be at multiple stages at the same time.

Beware if you use `single_state` as marking store for your workflow and name the storage field within your class `workflowState` then you do not need above helper but can access the field content directly via `workflowState`.

Debugging Data Query Selectors

You can append `:debug` to your Data Query selectors to see the single steps what happens behind the scenes. For instance for a Pimcore-object-based dataport you want to debug the data query selector `brand:brandName`, simply change it to `brand:brandName:debug` (or add a new field) and you will get:

```
[debug] Using context object /Products/1234
[debug] Found 1 element
[debug] Using function data_director_564b21289729229e07e4cc840970c641 in
<Pimcore cache directory>/Product_Brand_brandName.php to resolve data query selector
[debug] Result of data query selector part "Brand" is "/Brands/ABC"
[debug] Field or function "brandName" could not be resolved
[debug] Final result:
```

In this case the field `brandName` in the class `Brand` got renamed to `name`, thus the data query selector did not return any data. After it is changed to `brand:name:debug`, you will see that it works:

```
[debug] Using context object /Products/1234
[debug] Found 1 element
[debug] Using function data_director_2ac9c05c54567394d866ea1c99ccbd97 in
<Pimcore cache directory>/Product_Brand_brandName.php to resolve data query selector
[debug] Result of data query selector part "Brand" is "/Brands/ABC"
[debug] Result of data query selector part "name" is "ABC"
[debug] Final result: ABC
```

Finally you can remove the `:debug` and the data query selector will return the brand name `ABC`.

Get all Classification store / object brick field contents incl. field labels

With the data query selector `classificationStore:labels` you can get an associative array of all assigned groups with the translated field names:

```
{
  "Translated name of Group 1": {
    "Translated field title of field 1": "field 1 value",
    "Translated field title of field 2": 123.45
  },
  "Translated name of Group 2": {
    "Translated field title of field 3": "field 3 value"
  }
}
```

If you need the labels but also the field names, you can use `classificationStore:labels#true`. This way you get

```
{
  "groupName1": {
    "groupTitle": "Translated name of Group 1",
    "values": {
      "fieldName 1": {
        "fieldTitle": "Translated field title of field 1",
```

```

        "value": "field value"
    },
    "fieldName 2": {
        "fieldTitle": "Translated field title of field 2",
        "value": 123.45
    }
}
},
"groupName2": {
    "groupTitle": "Translated name of Group 2",
    "values": {
        "fieldName 3": {
            "fieldTitle": "Translated field title of field 3",
            "value": "field 3 value"
        }
    }
}
}
}

```

Field types with variant / placeholder support

There are 3 new field types available in Pimcore's class definition view:

- Input with variables
- Textarea with variables
- Wysiwyg with variables

Those fields are a combination of normal text fields and calculated value fields. With data query selectors you can access data of other fields and other objects, e.g.

This `{{ category:name }}` is `{{ color }}`.

will get resolved to `This t-shirt is black` if there is a many-to-one relation for the product's category which has a field `name` and if the product has an attribute `color`.

You can also implement any logic operations with [Twig syntax](#) with the exception that the variables are defined with data query selectors, e.g.

```

{% if category:name#en == "T-shirts" %}
    This shirt is made of {{ materials:each(name):implode( and , %s) }}.

```

Properties:

```

<ul>
{% for attributeName, attributeValue in attributes:tshirts %}
    <li>{{ attributeName }}: {{ attributeValue }}</li>
{% endfor %}
</ul>
{% endif %}

```

If the assigned category's name is "T-shirts" and there is an object brick container which the object brick `tshirts` got assigned, the result will be:

This shirt is made of cotton and polyester.

Properties:

```
<ul>
  <li>Color: black</li>
  <li>Size: XL</li>
</ul>
```

This allows to define the template for the texts at a high level and then inheriting it down for all descendant products. Thus you have automatic text generation dependent from your template which gets defined directly in Pimcore backend.

This also allows for rapid prototyping of applications: For example if you develop a product detail page and know where the product price shall appear, you can create a field **price** and access it in your template. Later on you can either fill this field and / or also refer to other fields of the data object or other objects. The advantage of this compared to a [calculated value field](#) is that you can implement the logic directly in the data objects and also inherit / override the logic for a certain group of data objects without having to create loads of if conditions in a calculated value calculator.

This way you could for example change the calculation of prices without changing the controllers or frontend templates. This makes you independent of the existence or naming of fields which may be used for price calculation, like date validation range, customer-specific pricing etc.

Different behaviour of getter methods for exports

Normally the data query selectors use the getter methods from the Pimcore classes. These return always the same values independent of they are called from the Pimcore backend, frontend or webservice. But in some use-cases you might want to get a different behaviour depending on the access method. This is possible when you [override the getter method](#). You can then check if you are currently in a data director extraction context with:

```
public function getMyField() {
    $request = \Pimcore::getContainer()->get('request_stack')->getMainRequest();
    $contextResolver = \Pimcore::getContainer()->get(Pimcore\Http\Request\Resolver\PimcoreCont

    $value = parent::getMyField();
    if($contextResolver->matchesPimcoreContext($request, 'webservice')) {
        // implement custom logic here
    }
    return $value;
}
```

That the data director context is named **webservice** has historical reasons. Until Pimcore 6 the data director extended Pimcore's now removed webservice API.

Create / edit multiple objects with one raw dataset

There are cases where you want to create multiple Pimcore objects from one dataset in the import source. For example when you have the following XML as import source:

```
<article>
  <article_number>123</article_number>
  <colors>
    <color>red</color>
    <color>blue</color>
    <color>green</color>
```

```
</colors>
</article>
```

You have two raw item columns (raw item XPath would be `/article`):

- “article number” with XPath `article_number`
- “colors” with XPath `colors/color` with “multiple” checkbox being set

The raw data import generates this raw dataset:

Article number	colors
123	['red', 'blue', 'green']

Now you want to create 3 Pimcore data objects from this raw data item. This can be achieved by **returning an array in a key column**.

For simplification imagine there is a Product class with two input fields `article_number` and `color`. In the import’s attribute mapping you assign **both fields as key columns**. `Article number` gets assigned to the `article_number` field and `colors` to the `color` field. When an array gets returned for a key column this is always interpreted as you want to create / edit multiple objects from this one raw dataset. Internally all possible combinations of all key column values get created. In this case these are:

- Article number: 123, colors: red
- Article number: 123, colors: blue
- Article number: 123, colors: green

The actual import gets run for each of these combinations individually. This is even possible if more key columns contain an array of values (or their callback function returns an array - e.g. splitting a comma-separated list in a CSV file) - so the number of objects being imported per raw dataset is the product of the number of values which the key columns contain.

Finally you will get 3 data objects, all with the same article number, but each with a different color.

Virtual Fields

You can use `{{ VARIABLE NAME }}` as a placeholder in the callback functions. This will add a *virtual field* with the given name to the attribute mapping panel. Virtual fields can be used for several things:

- fetch elements via [data query selectors](#) and use them for additional logic, e.g.

```
$object = {{ OTHER OBJECT }};
return $object->getField1() ?: $object->getField2();
```

This will create a virtual field `OTHER OBJECT` where we could have `return 'Category:name:'.$params['value']` as callback function. This virtual Field will try to find a Category object with the field `name` being equal to the assigned raw data field. In the field with above callback function we then return either the content of `Field1` or if this is empty then the content of `Field2` of the returned Category.

- reusable templates can contain virtual fields. For example the bundle provides a template for uploading export files via FTP. The template defines the virtual fields `FTP Host`, `FTP Port`, `FTP username`, `FTP password`, `FTP Path`. You do not have to edit the code provided by the template but can just change the values for these virtual fields.

Virtual fields support getting values from `app/config/parameters.yml` and `.env` files. For example when you create export dataports for a shop API, you could put your API credentials in one of those files, in this example in `app/config(/parameters.yml`:

```
parameters:
  API_ENDPOINT: http://example.org/api
```

When there is any callback function which uses the placeholder `{{ API_ENDPOINT }}` the value from `parameters.yml` will automatically get used - but of course you can override it by assigning a raw data field or implementing a callback function.

Reuse value of other field

When the name in the double curly braces refers to a really existing field's name (e.g. `{{ Key }}` for the object key) the field does not get treated as a virtual field but it will contain the mapped return value of the referenced field. For example, when you have a callback function for the object key and want to use the same logic also for the `name` field of `Product` objects without duplicating the callback function, then you can use `return {{ key }}` to get the same value which got returned from the callback function for the field `Key`.

Optimize inheritance

Updating the uppermost object

Enabling the *optimize inheritance* checkbox in the dataport settings leads the importer to set the imported data not necessarily directly on the object which is identified by the key fields but at one of its parents / ancestors to take advantage of object inheritance. An example: You have the following master-slave product structure:

- Master
 - Slave 1
 - Slave 2

There are 3 lines in your data source:

ID	Name	Price	Master ID	Category ID
1	Test Product			2
2	Test Product	10	1	2
3	Test Product	20	1	2

When you import with “optimize inheritance” the importer would update the master by setting the name and category. Then it imports the slaves also with name and category. When you later want to change the name or category, you have to change that in all 3 objects.

When you use “optimize inheritance” the importer would update the master as above. But when it comes to importing the slaves, the importer fetches the slave's siblings. If all siblings have the same value in the imported field, the value is only updated at the uppermost object. In this case the name and category fields would stay empty for the slaves because all slaves have the same name and category. The category and name will only be updated at the master. Nevertheless the slaves get the values in these fields via inheritance.

As prices differ between the slaves, they are set on the slave objects.

When you later want to change the name or category, you only have to change it on one single point: the master. This makes editing a lot easier.

This also applies to a lot of other use-cases, e.g. technical data via object bricks: If you structure your object tree hierarchically, it often means that similar objects are under the same branch. For example all products under a certain branch consist of the same material. Then the inheritance optimization would update the object brick field “material” only for the branch root. All objects below would inherit this value.

Creating object hierarchy

To fully take advantage of this feature you should think about grouping objects by certain attributes. For example you could define the desired object tree hierarchy **main category > standard > material > article number**. When you return this structure as the path in attribute mapping for the field **parent element / path**(e.g. **Filters/DIN 123/Stainless Steel**) and the article number in the field **object key** the path’s parts are created as objects (in contrast to folders when you do not use “optimize inheritance”). This allows for optimal use of inheritance.

Import assets

When you choose **Asset** as import class you can import assets without touching any data objects. This is especially useful in combination with the import type **Filesystem** because then you can periodically import assets from a certain directory into Pimcore.

The most important field in field mapping to import assets is the field **Stream**. You can assign it an absolute path to the file or a URL. By returning a string with a trailing slash / you can create folders.

Import documents

Document imports allow creating documents based on a set *master document*. All editables of the master document are available in the import’s attribute mapping. You can either build a [view with editables](#) or create the document structure dynamically via [area blocks](#).

The set *master document* will be set as [content master document](#) in newly created documents. This allows for prefilling content in the master document which does not need to be imported.

Unpublished versions

On the one hand it is important to not overwrite changes of unpublished versions and on the other hand to not publish the changes of unpublished versions. For this reason for elements whose latest version is unpublished, the import gets executed for currently published object version as well as for latest unpublished version - resulting in 2 new versions: one published based on the last published version but including the changes from the import, and one unpublished version based on the latest unpublished version but also including the changes from the import.

For new objects or objects whose latest version is published, of course only one new version will get created (is anything changed during the import).

Logging and error tracking

During a dataport run a lot of messages get logged to know exactly what happens. Those logs can be accessed in the **History & Run** panel of the Data Director. Additionally the same messages get logged to Pimcore's application logger. With the latter you can filter by component `Blackbit/DataDirectorBundle` to only see the logs of this bundle.

By default only logs which are equal or worse than `warning` get logged to the application logger. You can adjust parameter `blackbit_pim.application_logger_log_level` to a [supported log level](#) in your `config/packages/services.yaml` (Pimcore ≥ 10) or `app/config/parameters.yml` (Pimcore < 10).

You can change logging behavior completely by overriding the symfony service `pim.logger`.

For all dataport runs which do not get started manually from Pimcore backend (e.g. cronjob-triggered / automatic dataport runs / REST API calls), the log file gets deleted if no error occurred during the run. This way no disk space gets wasted for completely successful runs.

This bundle also supports [elements-at/ProcessManager](#). With this bundle you can execute import jobs at certain times - like cronjobs but you can config those within the Pimcore GUI. Another advantage of the Process Manager is that you can configure *logging channels*. You can define which error levels shall be logged to which output channels. For example you can create a separate log file only for your imports (so they do not get mixed with the normal Pimcore log) or you can specify to get *notified via email* when critical errors happen during an import.

Notifications

Whenever there is an error during an import which did not get executed with the `--force` flag, all users who are allowed to configure the dataport are notified via email.

For more advanced notifications you can create a custom email in the [result callback function](#) and send it with the included [result document action](#) for sending emails. This can for example be used to:

- get notified of import errors (when not using [elements-at/ProcessManager](#))
- send result documents (e.g. exported CSVs, XMLs etc.)
- get notified about new data in a Pimcore custom report

Permissions

For each dataport you can set which users are allowed to *execute* or to *configure* it. This uses the standard [role-based Pimcore permission system](#).

Additionally there is the special permission **Create dataports** for those users who are allowed to create dataports. The user who creates a dataport automatically gets configuration and execution permission for it.

Other Pimcore permissions and workspaces are respected, so a user who is not allowed to access objects under `/products` can neither execute imports which create / edit objects inside this folder, nor execute exports of objects in this folder.

Deployment

For every dataport there is a definition file in `<Pimcore-Root>/var/bundles/BlackbitDataDirector/` folder. As soon as dataport settings get changed, the respective file gets updated.

You should add this directory to your version control system (e.g. Git) to be able to set up a dataport on one system (e.g. local development) and then deploy it to other servers (e.g. live server).

With the command `bin/console data-director:deployment:dataport-rebuild` you can create / update the dataports based on the files in above mentioned `<Pimcore-Root>/var/bundles/BlackbitDataDirector/` folder. It is also possible to only restore some dataports by providing a comma-separated list of dataport ids or names, e.g. `bin/console data-director:deployment:dataport-rebuild "1,2,Dataport ABC"`.

Additionally, this command can be used to synchronize dataports between different Pimcore systems, e.g. `bin/console data-director:deployment:dataport-rebuild --source=https://other-pimcore.com --api-key=1234` will use the [dataport configuration REST API](#) to retrieve available dataports from the remote system `https://other-pimcore.com`. When working with remote source, only dataports whose latest version has been imported or which do not exist yet, will get imported. For already existing dataports whose latest version has been edited by a real user, you will get asked if you want to overwrite the local dataport configuration (to prevent local changes from being overwritten). As soon as you allow the import, the latest version will be marked as been imported and thus can be synced automatically in future. If you want to always overwrite local changes, use the `--force` flag.

Automatic tests

With the command `bin/console data-director:create-test <Dataport ID>` you can create a PHPUnit test for the given dataport. Based on the currently existing raw items and mapping a test class will get created. You can run those automatic tests via PHPUnit.

Main purposes for automatic tests:

1. Assure that after changes in dataport settings or attribute mapping everything works the same as before.
2. Assure that before and after updates / modifications of underlying libraries like Pimcore or Data Director itself, everything still works as before.

The test classes / files will get created in `<Pimcore root>/tests/BlackbitDataDirectorBundle`.

To run the tests a `phpunit.xml` is needed. If this does not already exist in the Pimcore root folder, it will also get automatically created. If it already exists, a new test suite will get added to it.

The command will display instructions how to run the tests.

Revert imports

Sometimes humans make mistakes but in attribute mapping and especially its callback functions small logic mistakes can have serious consequences: e.g. when the import changed thousands of Pimcore objects the wrong way. In this moment you could restore a complete backup of the Pimcore system. But this has the main disadvantage that in the meantime other objects than the corrupted ones could have been changed - restoring a backup would revert all those changes.

A better way is to use the command `bin/console data-director:revert <Dataport-ID> <Date to revert back to>`. With this command objects of the target class of the dataport get reverted

to a version equal or before the specified date. Furthermore only those fields are reverted which are mapped in the dataport's attribute mapping.

Example: You have a Product class with fields **Article description** (Text) and **Images** (Many to many relation). Until now images got assigned to the field **images** manually. Now you want to add images from an external source via import. The **Article description** is filled manually. You create the dataport for the image import and accidentally activate the checkbox "Clear before import" for the field **images**. You run the import and do not recognize the mistake at first sight. In the meantime the objects' field **Article description** got changed for a lot of products manually.

Eventually you recognize the error with the missing images. It is easy to repair the import but all the previously assigned images are lost. Now you cannot restore a backup as all the manually edited **Article description** data got lost - as well as any data of other objects which got altered meanwhile.

Now the command `bin/console data-director:revert <Dataport-ID> <Date to revert back to>` comes into play: It reverts the product objects to a given point in time (as long as there is a version of the given time or older) and it even reverts only those fields which are mapped in the given import. This means the **images** relation gets reverted while the field **Article description** keeps its current contents.

If you only want to revert objects which are found in the current raw data of the dataport you can use the option `--only-current-rawdata`.

Alternatively you can filter the elements to be reverted with the `--objects` option which you can provide a comma-separated list of object ids.

It is also possible to first do a dry run with the option `--dry-run`. With this you see the same output which objects would get reverted but no data gets actually changed.

Initialization function

In attribute mapping there is a special field named **Initialization function**. This allows you set define a callback function which gets executed once before raw data processing starts. Here you can for example check if another process of this dataport is currently running or reset a certain field of a data object which afterwards gets set again by the imported raw data items.

Import result callback / Result document generation

In attribute mapping there is a special field named **Result callback function**. This function gets executed after a raw data item has been processed. For example you can use this field to:

- generate a response document including all successfully imported items and send that to the source system
- track import errors
- call another import which depends on the current one
- create / export documents (JSON / CSV / XML etc.) which other systems can use as import source
- create response documents for single-page application / PWA frontend requests

This result callback function receives the following parameters in variable `$params`:

- **response** - current response document generated by this callback function from previously processed raw items, initially empty `Symfony\Component\HttpFoundation\Response` object - you can either populate the response via `$params['response']->setContent('my response content')` or return `"my response content";`.
 - response headers can be set via `$params['response']->headers->set('Header Name', 'Header value');`
- **rawItemData** - array with imported raw data
- **request** - object of class `Symfony\Component\HttpFoundation\Request` (even when called via CLI)
 - access current language via `$params['request']->getLocale()`
 - request parameters can be accessed via `$params['request']->get('parameterName')` but are also available via `'{{ parameterName }}` placeholders
- **transfer** - object for data holding between different raw data items, initially empty
- **lastCall** - boolean value to determine if current call is the last one before import / export is finished
- **context** - array with following keys:
 - **dataport**
 - * **id**: dataport ID
 - * **dataportName**: dataport name
 - **resource**: array with set import source, may contain the following keys:
 - * **file**: import resource (e.g. file name for file-based imports or SQL condition for Pimcore-based imports)
 - * **locale**: language to be used - especially for Pimcore-based exports
 - **user**
 - * **id**: id of the user who started the dataport run
 - * **username**: name of the user who started the dataport run
 - **logs**: Logs for whole import in temporal order
- **logs** - array with noteworthy things which happened during import of current raw item, indexed by level of severity, e.g.


```
[
  'warning' => ['First warning', 'Second warning'],
  'error' => ['First error', 'Second error'],
  'alert' => ['Really severe error'],
  'info' => ['First info', 'Second info', 'Third info'],
  'notice' => ['First notice', 'Second notice']
]
```
- **objectIDs** - array with object IDs which got changed by the rawdata item
- **logger** - Logger object which implements `\Psr\Log\Logger`
 - Usage example: `$params['logger']->error('Something unexpected happened');`

Result document actions

Beside `Result` callback function there is another special field in attribute mapping: `Result document action`. With this field you can define what to do with the generated result document. This could be normal output (so the generated document gets sent to the browser), FTP upload, send the document via email etc. For the listed use cases the required functions get already shipped as templates so you do not have to program anything but only select one of the templates (but of course you can adjust the implementation logic if you want to).

This `Result` callback function gets the following parameters in variable `$params`:

- `response` - generated response document, instance of `Symfony\Component\HttpFoundation\Response`
- `transfer` - object which can be populated in [result callback function](#)
- `context` - array with following keys:
 - `dataportId`: dataport ID
 - `resources`: array with set import sources, each dataport resource is an array item with the following keys:
 - * `file`: import resource (e.g. file name for file-based imports or SQL condition for Pimcore-based imports)
 - * `locale`: language to be used - especially for Pimcore-based exports
- `logger` - Logger object which implements `\Psr\Log\Logger`
 - Usage example: `$params['logger']->error('Something 'unexpected happened');`

REST interface

This bundle brings its own REST API interface with the following endpoints:

- `POST http(s)://<YOUR-DOMAIN>/api/rest/import/<Dataport-Name or id>?apikey=<API-KEY> [&async=1] [¶meter1=123] [&anotherParam=1]`
 - In the request body you provide the document to be imported (e.g. CSV / XML / JSON content)
 - `<API-KEY>` is the [API key](#) of the Pimcore user. You can only import data to objects which this user has access to.
 - Example for a CSV import:

```
POST http(s)://[YOUR-DOMAIN]/api/rest/import?apikey=[API-KEY]
&dataportId=[Dataport-ID]

Article number,name
123,My cool product
```
 - via [import callback function](#) you can generate a response document
 - requests with `async` parameter are run in the background. As response you get a URL from which you can fetch the current status and result of the started import (see next bullet point)
 - provided parameters can be accessed in raw data fields and attribute mapping via `{{ parameter1 }}` or `{{ anotherParam }}` - the values come from the URL parameters and can be named however you want, see [parametrized dataports](#)

- GET `http(s)://<YOUR-DOMAIN>/api/rest/export/<Dataport-Name or id>?apikey=<API-KEY>[&locale=<language / locale code>][¶meter1=123][&anotherParam=1]`
 - returns the result of the [result callback function](#)
 - when exporting data from localized fields you can either access the field name itself and handle the target language via the `locale` parameter (e.g. `locale=de`) or you can specify the language to be exported in the raw item fields - e.g. `name#en` will always export english name regardless of `locale` parameter whereas `name` will export the name in the given language. If no valid `locale` is given, the configured language of the Pimcore user, whose API key is being used in the request, will be used.
 - provided parameters can be accessed in raw data fields, SQL condition and attribute mapping via `{{ parameter1 }}` or `{{ anotherParam }}` - the values come from the URL parameters and can be named however you want, see [parametrized dataports](#)
- GET `http(s)://<YOUR-DOMAIN>/api/rest/status?apikey=<API-KEY>&statusKey=<Status key>`
 - get result of asynchronous import
 - `<Status key>` is the key which gets returned by `/api/rest/import` with `async=1`
 - as long as `HTTP 102 Processing` gets returned the import has not finished yet (so the response body contains only data up to the last imported item)

In all above requests the API key can also be transferred via `X-API-Key` header instead of the `apikey` parameter.

API Keys

Below the dataport tree there is a button **Permissions & API Keys**. There you see all users which the current account is allowed to administer (admin accounts will see all users, other users will only see the dataports which they are allowed to configure). Initially the users do not have API keys but as soon as you double-click in the column **API Key** for a certain user an API key will get created. If you want to change it, you can double-click again and edit it. Optionally you can set an expiration date if you want to limit the lifetime of this API key.

Of course the user needs [execution permission](#) for the dataport to be able to execute it via REST API.

Dataport configuration REST API

List dataports

All existing dataports can be retrieved via

GET `http(s)://<YOUR-DOMAIN>/api/dataports?apikey=<API-KEY>`

In the response you will get a list of dataports as JSON:

```
{
  "3": {
    "name": "Import Products",
    "lastModified": 1687849015
  },
  "19": {
    "name": "Export Products",
    "lastModified": 1679390079
  }
}
```



```
},  
...  
}
```

Only dataports get listed which the user, whose API key gets used, has configuration permission.

Dataport configuration

The endpoint to retrieve the full configuration of a dataport is

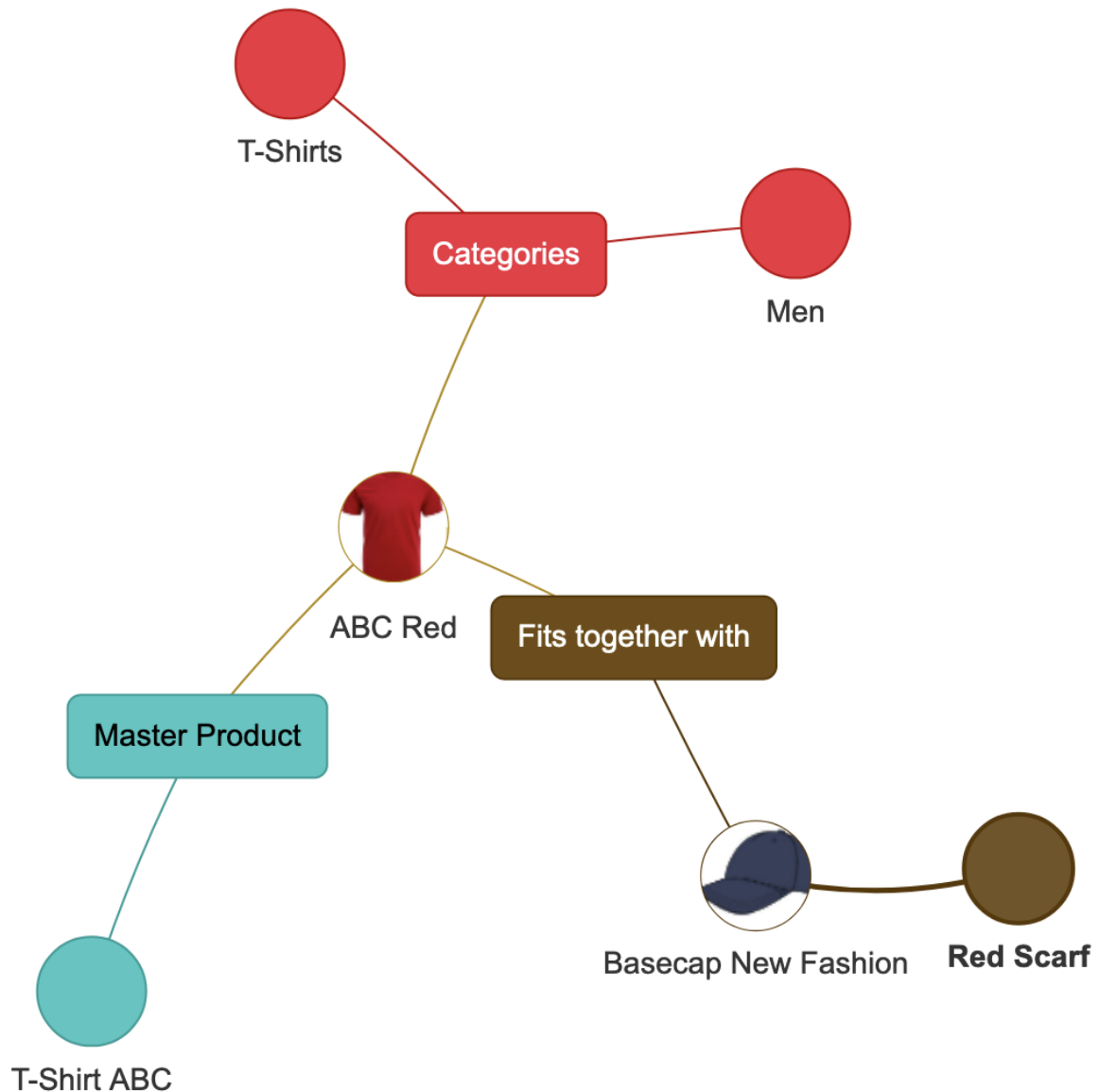
```
GET http(s)://<YOUR-DOMAIN>/api/dataports/<dataport id or name>?apikey=<API-KEY>
```

This will return the full configuration as JSON, similar to the definition files in `<Pimcore-Root>/var/bundles/Black` folder, see [deployment](#).

Object Preview

Pimcore supports a [preview tab](#) in the object editing panel. This feature can be combined with a Data Director export. The export can create an HTML document which can be shown in the preview panel of the currently opened object. To achieve this, you have to enter `@DataDirectorPreview` in the field **General Settings > Preview Generator Class or Service Name** of your data object class. Afterwards when you open a data object of this class you will immediately see the out-of-the-box **Default Preview** available which shows the current object's data as known from the [versions panel](#). But the real cool feature is when you create a [Pimcore-based dataport](#) which generates an HTML document. You can use `o_id='{ { id } }'` as SQL condition and then add the desired raw data fields and generate the desired output in the [result callback function](#).

One use-case which the Data Director already supports out-of-the-box is a dependency graph:



Mouseover information in element tree (tooltip)

By default Pimcore only shows the element ID and type when hovering an element in the tree. Data Director extends this functionality by additionally showing all fields which are configured to be **Visible in Grid View**.

Path formatter

Relational fields support [path formatters](#) to define the display of the related item. Data Director comes with a universal implementation which uses the fields of the related item, which are configured to be **Visible in search Result**, also in the relation field view. To use this, set **Formatter service / class** of the relational field to `@DataDirectorSearchViewPathFormatter`.

This gets done automatically for all relational fields which do not have a **Formatter service / class**. If you prefer to show the full path of the related item (= Pimcore's default display of relations), please use `@DefaultPathFormatter`.

If you prefer to show the fields which are configured to be **Visible in Grid view**, you can use `@DataDirectorGridViewPathFormatter`.

Data option providers

For select and multiselect fields Pimcore supports [data option providers](#) to create a dynamic list of available options. When you set **Options provider class** or **Service Name** in your select field configuration to `@DataDirectorOptionProvider`, you can use a [data query selector](#) to define the allowed options.

For example you can use `Brand:published:true:each(name)` to define the names of all published **Brand** objects as options. The problem with this approach is that as soon as you change the name of an assigned brand, the assignment in the select could not be found anymore. For this reason you can use a more sustainable solution: `Brand:published:true:each(name as label;id as value)`. This way the name will be used for displaying purpose but the actual value on the database will get defined by the **Brand** object's ID.

Additionally, you can access values of the currently opened object to implement dependent select fields: Imagine you have a multiselect field in the **Category** class and use `Product:category:{{ id }}:each(brand:id as value;brand:name as label)`. `{{ id }}` will get resolved to ID of the currently opened object (but also all other fields or chained data query selectors are possible here). In this case you will get a list of all brands of the products which have the currently opened category object assigned in their **category** field.

Perspectives / Dashboards

Pimcore supports [perspectives](#) to customize the backend layout. Data Director provides a basic perspective based on the data object classes' groups. All data object classes which belong to the same **group** get shown in the same perspective panel. The root folder gets automatically optimized to the lowest parent object which contains all objects of the corresponding class.

The Data Director perspective also comes with a dashboard which includes a queue monitor, tagged elements and dataport errors portlets. Of course those portlets are also usable in other dashboards.

Troubleshooting

Import does not update my data objects

If you once imported data to a data object, a property `importhash_<dataport id>` with the hash of all belonging raw data gets added to the object. On the next import it will be checked if the object which got found via the set key attributes already has this hash attribute and it is equal to the hash of the current raw data to be imported. If it is, no data gets updated to not accidentally overwrite manual changes to fields which got initially filled by the import.

To bypass this behaviour you can use the option `--force` (or shortcut `-f`) when doing the "data-director:process" step, e.g. `bin/console data-director:process --force <Dataport-ID>` or `bin/console data-director:complete --force <Dataport-ID>`

Import became slow (was faster before) / Database table `application_logs` is very big

For convenience the importer logs messages to Pimcore's application logger. By default only warnings and worse errors are logged. You can configure this behaviour by setting

```
parameters:
    blackbit_pim.application_logger_log_level: 'debug'
```

in your `app/config/parameters.yml`.

But beware there can be loads of messages, so set the log level only to `debug` when you find data errors or when you configure new imports.

If you want to disable logging to the application logger completely, you can add the following to your `app/config/services.yml`:

```
pim.logger:
    class: Blackbit\DataDirectorBundle\lib\Pim\Logger\Logger
    arguments: ['@monolog.logger.pimcore']
    public: true
```

Import / Export gets aborted automatically

When an import needs a long time to import a single raw data item, the import is assumed to have been terminated via CLI and gets marked as **Aborted**. This for example happens when a single raw item shall update a big number of objects. The time after which an import gets aborted is dependent on the following rules:

- if less than 100 items have been imported: import process gets aborted after 3600 seconds (= 1 hour)
- if more than 100 items have been imported: import gets aborted if current batch of 100 raw items needs more than the average of 100 previously imported raw data items multiplied by factor 6. Or in other words: If importing current batch of 100 raw items needs more time than 6 average batches of previously imported items, the import gets aborted.

You can increase the parameter `blackbit_pim.importstatus.abortion_threshold` in `app/config/parameters.yml` to enlarge the abortion threshold:

```
parameters:
    # in seconds, default is 3600 (1 hour)
    blackbit_pim.importstatus.abortion_threshold: 72600
```

If that does not solve your problem, maybe the process gets killed by the webserver or operating system. For example watch for Apache's `Timeout` or `FcgidIOTimeout` settings.

Also some webhosters kill CLI processes which run longer than a certain threshold.

Queued items shall get processed in a night run

By default queued jobs (e.g. via automatic dataports) get processed immediately. If you prefer to control queue processing on your own (for example with a watchdog process), you can configure

```
blackbit_data_director:
    queue_processing:
        automatic_start: false
```

The queue processing then has to be started via `bin/console data-director:process-queue`. Due to a locking mechanism there can never be more than one queue processor process running.

What if key field and set value shall differ?

In some cases you want to filter the objects to be updated with a different value than you want to set the field content to. For example this is necessary when you have a many-to-many relation and want to additionally refer object A to all objects which currently have object B assigned in the same relation field. In this case you have to apply a little trick: When fetching the key field values `currentValue` of course is not set (as this contains the value of the current processed object but at this stage we want to determine the objects to be fetched).

So we can differentiate between filtering and writing by providing a callback function like:

```
if(!isset($params['currentValue'])) {  
    // find objects which have the following object assigned  
    return 'Reference_Class:field:'. $params['rawItemData']['field_value']['value'];  
}  
  
// value setting  
return [  
    'Reference_Class:field:'. $params['rawItemData']['field_value']['value'],  
    'Reference_Class:field:'. $params['rawItemData']['other_field_value']['value']  
];
```

Demo / Tutorials in Docker

Please install [Docker](#)

To run the demo you need to run `./docker-setup.sh` to create a docker container with the installed data director bundle. You can then access the Pimcore demo admin under <http://localhost:2000/admin>.

- User: admin
- Password: admin

The installation will need some time. When you installed it once and only want to restart the container, just call `docker-compose up -d`. You can stop it via `docker-composer stop`.

Tutorials

When you want to try yourself what is shown in the [tutorial videos](#) you will find all the examples including instructions, import data and finished solutions (dataport JSON files you can simply import if you do not know how to proceed) in the `examples/` folder.