Netgen's Site API for eZ Platform

Mar 08, 2019

Contents

CHAPTER 1

Introduction

The intention of this page is to give you a short overview of what Site API is. For that purpose we can break the whole package into three main parts:

1. *Dedicated API layer*
2. *Integration with eZ Platform*
3. *Query Types*

1.1 Dedicated API layer

As Repository API was designed to be usable for general purpose, it can come as awkward and too verbose when used for building websites. Site API fixes this by implementing a dedicated API layer on top of eZ Platform Repository API which is designed for developing websites.

Having a dedicated layer enables us to take an extra step and do things you would not typically want to do in Repository API. With Site API we can we can implement lazy loaded properties and methods that enable content model traversal directly from the entities because:

1. it's a dedicated layer for building websites
2. it's not intended to be layered (meaning no different API implementations like Cache, Permission etc)

1.1.1 Handling multiple languages

The way Site API handles multiple languages was the initial motive for implementing it and deserves to be mentioned separately.

Language configuration for a siteaccess consists of a prioritized list of languages. For example, you could have a siteaccess with two languages, Croatian language as the most prioritized one and English language as a fallback when Croatian translation does not exist:

```
ezpublish:
    system:
        cro:
            languages:
                - 'cro-HR'
                - 'eng-GB'
```

The intention here is that the siteaccess should first show content in Croatian language if it's available, fallback to English translation when Croatian is not available and ignore any other language. However, this is quite hard to implement correctly with vanilla Repository API, even with the newest addition of siteaccess-aware Repository layer introduced in eZ Platform 7.2.

With Site API this comes out of the box and you don't have to pay special attention to it. All possible ways to get a Content or a Location, whether through loading by ID, as a related Content, accessing the field on the parent Location's Content, searching or using methods and properties on the Site API objects – it already respects this configuration. You can depend that you will always get back only what can and should be rendered on the current siteaccess and then simply stop caring about it, because it just works.

That feature alone significantly reduces cognitive load for developers, frees them from writing tedious boilerplate code just to respect the language configuration, avoids ridiculous sanity checks and mistakes and improves the overall developer experience.

1.1.2 Objects

Site API entities and values are similar to their counterparts in eZ Platform's Repository API:

- `Content`

  The first difference from Repository Content is that it exist in a single translation, meaning it contains the fields for only one translation. That translation will always be the correct one to be rendered, resolved from the language configuration of the siteaccess. You won't need to choose the field in the correct translation, manually or through some kind of helper service. The Content's single translation is always the correct one.

  Content fields are lazy-loaded, which means they are loaded only if accessed. This voids the need to have a separate, light version of Content (ContentInfo in Repository API). Content object also provides properties and methods to enable access to Content's Locations and relations. Example usage from Twig:

  ```
  <h1>{{ content.name }}</h1>
  <h2>Parent name: {{ content.mainLocation.parent.content.name }}</h2>
  <h3>Number of Locations: {{ content.locations|length }}</h3>

  {{ ng_render_field(content.fields.title) }}

  <ul>
      {% for relation in content.fieldRelations('articles') %}
          <li>{{ relation.title }}</li>
      {% endfor %}
  </ul>
  ```

- `ContentInfo`

  The purpose of ContentInfo object in Repository API is to provide a lightweight version of Content object, containing only metadata (and omitting the fields). Since in Site API Content's fields are lazy-loaded, there is no real need for ContentInfo. Still, Site API provides it to keep the usage in templates similar to standard eZ Platform templates and through that make the migration and comparison easier.

  Site ContentInfo also provides access to data that is in Repository API available only through loading other objects, like ContentType identifier. Example usage from Twig:

```
<h2>Section ID: {{ content.contentInfo.sectionId }}</h2>
<h2>ContentType identifier: {{ content.contentInfo.contentTypeIdentifier }}</h2>
```

Note:

In Site API it is not possible to load `ContentInfo` directly.
It is only available through properties on `Content` and `Location` objects.

- `Location`

  Site `Location` is similar to Repository Location. It provides properties and methods to enable simple Location tree traversal (siblings, children, parents, ancestors etc). Example usage from Twig:

```
<h1>{{ location.content.name }} - Articles</h1>
<h2>Parent: {{ location.parent.content.name }}</h2>
<h3>Grandparent: {{ location.parent.parent.content.name }}</h3>

{% set children = location.filterChildren(['article']) %}

<ul>
{% for child in children %}
    <li>{{ child.content.name }}</li>
{% endfor %}
</ul>

{{ pagerfanta( children, 'twitter_bootstrap' ) }}
```

- `Field`

  `Field` object aggregates some properties from it's FieldDefinition, like FieldType identifier, name and description. It also implements `isEmpty()` method, which makes simple to check if the field value is empty, without requiring external helpers. Example usage from Twig:

```
<h1>{{ content.fields.title.name }}</h1>
<p>You can access the value directly: {{ content.fields.title.value.text }}</p>

{% if not content.fields.title.empty %}
    <p>{{ ng_render_field( content.fields.title ) }}</p>
{% endif %}

{% set image = content.fields.image %}
{% if not image.empty %}
    <img src="{{ ng_image_alias( image, 'i1140' ).uri }}"
         alt="{{ image.value.alternativeText }}" />
{% endif %}
```

For your convenience all objects contain their corresponding Repository objects in properties prefixed with `inner`. Example usage from Twig:

```
<h1>Content ID: {{ content.innerContent.id }}</h1>
<h2>Location ID: {{ location.innerLocation.id }}</h2>
<h3>Field ID: {{ field.innerField.id }}</h3>
```

For more details see *Templating* and *Objects* reference pages.

1.1.3 Services

The API provides you with a set of **read-only** services:

1. `LoadService`

   Provides methods to load Content and Locations by ID (and remote ID):

2. `FindService`

   Provides methods to find Content and Locations using eZ Platform Repository Search API.

3. `FilterService`

   This is quite similar to the `FindService`, but only works with Legacy search engine, even if that is not the configured engine for the repository.

   Why? While Solr search engine provides more features and more performance than Legacy search engine, it's a separate system needs to be synchronized with changes in the database. This synchronization comes with a delay, which can be a problem in some cases.

   FilterService gives you access to search that is always up to date, because it uses Legacy search engine that works directly with database. At the same time, search on top of Solr, with all the advanced features (like fulltext search or facets) is still available through FindService.

4. `RelationService`

   Provides methods for loading relations.

All services return only published Content and handle translations in a completely transparent way. Language fallback configuration for the current siteaccess is automatically taken into account and you will always get back only what should be rendered on the siteaccess. If the available translation is not configured for a siteaccess, you won't be able to find or load Content or Location. The services will behave as if it does not exist.

Note: All of the Site API services are read-only. If you need to write to the eZ Platform's content repository, use it's existing Repository API.

For more details see *Services reference* page.

1.2 Integration with eZ Platform

You can use the Site API services described above as you would normally do it a Symfony application. But these are also integrated into eZ Platform's view layer. There is a Site API version of the view configuration, available under `ngcontent_view` key:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                line:
                    article:
                        template: "NetgenSiteBundle:content/line:article.html.twig"
                        match:
                            Identifier\ContentType: article
```

Aside from Query Type configuration described below, the format is exactly the same as eZ Platform's view configuration under `content_view` key. Separate view configuration is also needed because we need to handle it with code that will inject Site API objects to the template, instead of standard eZ Platform objects. Together with this we

provide Site API version of the Content View object, which is used by the default Content view controller and *custom controllers*.

With the configuration from above you you will be able to render a line view for an article by executing a request to `ng_content:viewAction`. However, that does not mean URL aliases will be handled by the Site API view configuration as well. This needs to be explicitly enabled, per siteaccess:

```
netgen_ez_platform_site_api:
    system:
        frontend_group:
            override_url_alias_view_action: true
```

Note: You can use the Site API's view configuration and eZ Platform's view configuration at the same time. However, URL aliases can be handled exclusively by the one or the other.

For more details see *Configuration reference* page.

1.3 Query Types

Query Types provide a set of predefined queries that can be configured for a specific view, as part of the view configuration under `ngcontent_view` key. It also provides a system for developing new queries inheriting common functionality.

While they can be used from PHP, main intention is to use them from the view configuration. This is best explained with an example:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    folder:
                        template: '@ezdesign/content/full/folder.html.twig'
                        match:
                            Identifier\ContentType: folder
                        queries:
                            children_documents:
                                query_type: SiteAPI:Content/Location/Children
                                max_per_page: 10
                                page: '@=queryParam("page", 1)'
                                parameters:
                                    content_type: document
                                    section: restricted
                                    sort: priority desc
```

Other side of the configuration from the example above is full view `folder` template:

```
{% set documents = ng_query( 'children_documents' ) %}

<h3>Documents in this folder</h3>

<ul>
{% for document in documents %}
    <li>{{ document.name }}</li>
```

(continues on next page)

(continued from previous page)

```
{% endfor %}
</ul>

{{ pagerfanta( documents, 'twitter_bootstrap' ) }}
```

If you used Legacy eZ Publish, this is similar to template fetch function. Important difference is that in Legacy you used template fetch functions to pull the data into the template. Instead, with Site API Query Types you push the data to the template. This keeps the logic out of the templates and gives you better control and overview.

For more details see *Query Types reference* page.

Site API is a lightweight layer built on top of eZ Platform's **Repository API**. It's purpose is to solve common use cases, remove boilerplate code and provide better **developer experience** for building websites. While it will make **PHP developers** more productive, it will also lower the entry barrier for newcomers and open most of the development process to **roles other than PHP developer**.

With it, frontend developers, content builders and others will be able to do most of the work related to eZ Platform content modelling, independently of PHP developers and without the need to work with PHP. They will require only knowledge of eZ Platform's content model, Twig templates and view configuration in YAML.

If you are new to Site API, read the *Introduction* first.

CHAPTER 2

Reference

2.1 Reference

2.1.1 Installation

To install Site API first add it as a dependency to your project:

```
$ composer require netgen/ezplatform-site-api:^2.5
```

Once Site API is installed, activate the bundle in `app/AppKernel.php` file by adding it to the `$bundles` array in `registerBundles()` method, together with other required bundles:

```
public function registerBundles()
{
    //...

    $bundles[] = new␣
↪Netgen\Bundle\EzPlatformSiteApiBundle\NetgenEzPlatformSiteApiBundle();
    $bundles[] = new␣
↪Netgen\Bundle\EzPlatformSearchExtraBundle\NetgenEzPlatformSearchExtraBundle();

    return $bundles;
}
```

And that's it. Once you finish the installation you will be able to use Site API services as you would normally do in a Symfony application. However, at this point Site API is not yet fully enabled. That is done per siteaccess, see *Configuration* page to learn more.

2.1.2 Configuration

Site API has it's own view configuration, available under `ngcontent_view` key. Aside from *Query Type* options documented separately, this is exactly the same as eZ Platform's default view configuration under `content_view`

key. You can use this configuration right after the installation, but note that it won't be used for full views rendered for eZ Platform URL aliases right away. Until you configure that, it will be used only when calling its controller explicitly with `ng_content:viewAction`.

To use Site API view rules for pages rendered from eZ Platform URL aliases, you have to enable it for a specific siteaccess with the following semantic configuration:

```
netgen_ez_platform_site_api:
    system:
        frontend_group:
            override_url_alias_view_action: true
```

Here `frontend_group` is the siteaccess group (or a siteaccess) for which you want to activate the Site API. This switch is useful if you have some siteaccesses which can't use the it, like custom admin or intranet interfaces.

Note: To use Site API view configuration automatically on pages rendered from eZ Platform URL aliases, you need to enable it manually per siteaccess.

One you do this, all your **full view** templates and controllers will need to use Site API to keep working. They will be resolved from Site API view configuration, available under `ngcontent_view` key. That means Content and Location variables inside Twig templates will be instances of Site API Content and Location value objects, `$view` variable passed to your custom controllers will be an instance of Site API ContentView variable, and so on.

If needed you can still use `content_view` rules. This will allow you to have both Site API template override rules as well as original eZ Platform template override rules, so you can rewrite your templates bit by bit. You can decide which one to use by calling either `ng_content:viewAction` or `ez_content:viewAction` controller.

Tip:

View configuration is the only eZ Platform configuration regularly edited by frontend developers.

For example, if using the following configuration:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                line:
                    article:
                        template: 'Bundle:content/line:article.html.twig'
                        match:
                            Identifier\ContentType: article
            content_view:
                line:
                    article:
                        template: 'Bundle:content/line:ez_article.html.twig'
                        match:
                            Identifier\ContentType: article
```

Rendering a line view for an article with `ng_content:viewAction` would use `Bundle:content/line:article.html.twig` template, while rendering a line view for an article with `ez_content:viewAction` would use `Bundle:content/line:ez_article.html.twig` template.

It is also possible to use custom controllers, this is documented on *Custom controllers reference* documentation page.

2.1.3 Templating

Site API objects are used directly in the templates. Below you will find examples for the most common use cases. Objects are documented in more detail on *Objects reference* documentation page.

Site API provides two Twig functions for content rendering:

- `ng_render_field`

  Similar to `ez_render_field` from eZ Platform, this function is used to render the Content's field using the configured template:

```
<p>{{ ng_render_field( content.field.body ) }}</p>
```

- `ng_image_alias`

  Similar to `ez_image_alias` from eZ Platform, this function provides access to the image variation of a `ezimage` type field:

```
<img src="{{ ng_image_alias( content.fields.image, 'large' ).uri }}" />
```

Both are shown in more detail in the examples below. There are two other Twig functions, `ng_query` and `ng_raw_query`. These are used with Query Types and are documented separately on *Query Types reference* documentation page.

Basic usage

- **Accessing Location's Content object**

  Content is available in the Location's property `content`:

```
{{ set content = location.content }}
```

- **Displaying the name of a Content**

  Content's name is available in the `name` property:

```
<h1>Content's name: {{ content.name }}</h1>
```

- **Linking to a Location**

  Linking is done using the `path()` Twig function, same as before.

```
<a href="{{ path(location) }}">{{ location.content.name }}</a>
```

- **Linking to a Content**

  Linking to Content will create a link to Content's main Location.

```
<a href="{{ path(content) }}">{{ content.name }}</a>
```

Working with Content fields

- **Accessing a Content Field**

Note: Content's fields are lazy-loaded, which means they will be transparently loaded only at the point you access them.

The most convenient way to access a Content field in Twig is using the dot notation:

```
{% set title_field = content.fields.title %}
```

Alternatively, you can do the same using the array notation:

```
{% set title_field = content.fields['title'] %}
```

Or by calling `getField()` method on the Content object, also available as `field()` in Twig, which requires Field identifier as argument:

```
{% set title_field = content.field('title') %}
```

- **Checking if the Field exists**

  Checking if the field exists can be done with `hasField()` method on the Content object:

```
{% if content.hasField('title') %}
    <p>Content has a 'title' field</p>
{% endif %}
```

- **Displaying Field's metadata**

  Field object aggregates some data from the FieldDefinition:

```
{% set title_field = content.fields.title %}

<p>Field name: {{ title_field.name }}</p>
<p>Field description: {{ title_field.description }}</p>
<p>FieldType identifier: {{ title_field.fieldTypeIdentifier }}</p>
```

- **Rendering the field using the configured template**

  To render a field in vanilla eZ Platform you would use ez_render_field function, which does that using the configured template block. For the same purpose and using the same templates, Site API provides it's own function `ng_render_field`. It has two parameters:

  1. **required** Field object
  2. **optional** hash of parameters, by default an empty array `[]`

     This parameter is exactly the same as you would use with `ez_render_field`. The only exception is the `lang` parameter, used to override the language of the rendered field, which is not used by the `ng_render_field`.

  Basic usage:

```
{{ ng_render_field( content.fields.title ) }}
```

  Using the second parameter to override the default template block:

```
{{
    ng_render_field(
        content.fields.title,
        { 'template': 'AcmeTestBundle:fields:my_field_template.html.twig' }
    )
}}
```

- **Checking if the Field's value is empty**

This is done by calling `isEmpty()` method on the Field object, also available as `empty()` or just `empty` in Twig:

```
{% if content.fields.title.empty %}
    <p>Title is empty</p>
{% else %}
    {{ ng_render_field( content.fields.title ) }}
{% endif %}
```

- **Accessing the Field's value**

  Typically you would render the field using `ng_render_field` Twig function, but if needed you can also access field's value directly. Value format varies by the FieldType, so you'll need to know about the type of the Field whose value you're accessing. You can find out more about that on the official FieldType reference page or even looking at the value's code.

  Here we'll assume `title` field is of the FieldType `ezstring`. Latest code for that FieldType's value can be found here.

```
<h1>Value of the title field is: '{{ content.field.title.value.text }}'</h1>
```

- **Rendering the image field**

  Typically for this you would use the built-in template through `ng_render_field` function, but you can also do it manually if needed:

```
{% set image = content.fields.image %}

{% if not image.empty %}
    <img src="{{ ng_image_alias( image, 'i1140' ).uri }}"
         alt="{{ image.value.alternativeText }}" />
{% endif %}
```

Traversing the Content model

Content Locations

- **Accessing the main Location of a Content**

```
{% set main_location = content.mainLocation %}
```

- **Listing Content's Locations**

  This is done by calling the method `getLocations()`, also available as `locations()` in Twig. It returns an array of Locations sorted by the path string (e.g. `/1/2/191/300/`) and optionally accepts maximum number of items returned (by default `25`).

```
{% set locations = content.locations(10) %}

<p>First 10 Content's Locations:</p>

<ul>
{% for location in locations %}
    <li>
        <a href="{{ path(location) }}">Location #{{ location.id }}</a>
    </li>
```

(continues on next page)

(continued from previous page)

```
{% endif %}
</ul>
```

- **Paginating through Content's Locations**

  This is done by calling the method `filterLocations()`, which returns a `Pagerfanta` instance with Locations sorted by the path string (e.g. `/1/2/191/300/`) and accepts two optional parameters:

  1. **optional** maximum number of items per page, by default `25`
  2. **optional** current page, by default `1`

```
{% set locations = content.filterLocations(10, 2) %}

<h3>Content's Location, page {{ locations.currentPage }}</h3>
<p>Total: {{ locations.nbResults }} items</p>

<ul>
{% for location in locations %}
    <li>
        <a href="{{ path(location) }}">Location #{{ location.id }}</a>
    </li>
{% endfor %}
</ul>

{{ pagerfanta( locations, 'twitter_bootstrap' ) }}
```

Content Field relations

- **Accessing a single field relation**

  This is done by calling the method `getFieldRelation()`, also available as `fieldRelation()` in Twig. It has one required parameter, which is the identifier of the relation field. In our example, the relation field's identifier is `related_article`.

```
{% set related_content = content.fieldRelation('related_article') %}

{% if related_content is defined %}
    <a href="{{ path(related_content) }}">{{ related_content.name }}</a>
{% else %}
    <p>There are two possibilities:</p>
    <ol>
        <li>Relation field 'related_article' is empty</p>
        <li>You don't have a permission to read the related Content</li>
    </ol>
    <p>In any case, you can't render the related Content!</p>
{% endif %}
```

Note: If relation field contains multiple relations, the first one will be returned. If it doesn't contain relations or you don't have the access to read the related Content, the method will return `null`. Make sure to check if that's the case.

- **Accessing all field relations**

This is done by calling the method `getFieldRelations()`, also available as `fieldRelations()` in Twig. It returns an array of Content items and has two parameters:

1. **required** identifier of the relation field
2. **optional** maximum number of items returned, by default `25`

```
{% set related_articles = content.fieldRelations('related_articles', 10) %}

<ul>
{% for article in related_articles %}
    <a href="{{ path(article) }}">{{ article.name }}</a>
{% endfor %}
</ul>
```

- **Filtering through field relations**

This is done by calling the method `filterFieldRelations()`, which returns a Pagerfanta instance and has four parameters:

1. **required** identifier of the relation field
2. **optional** array of ContentType identifiers that will be used to filter the result, by default an empty array `[]`
3. **optional** maximum number of items per page, by default `25`
4. **optional** current page, by default `1`

```
{% set articles = content.filterFieldRelations('related_items', ['article'], 10,␣
↪1) %}

<ul>
{% for article in articles %}
    <a href="{{ path(article) }}">{{ article.name }}</a>
{% endfor %}
</ul>

{{ pagerfanta( events, 'twitter_bootstrap' ) }}
```

Location children

- **Listing Location's children**

This is done by calling the method `getChildren()`, also available as `children()` in Twig. It returns an array of children Locations and optionally accepts maximum number of items returned (by default `25`).

```
{% set children = location.children(10) %}

<h3>List of 10 Location's children, sorted as is defined on the Location</h3>

<ul>
{% for child in children %}
    <a href="{{ path(child) }}">{{ child.name }}</a>
{% endfor %}
</ul>
```

- **Filtering through Location's children**

This is done by calling the method `filterChildren()`, which returns a Pagerfanta instance and has three parameters:

1. **optional** array of ContentType identifiers that will be used to filter the result, by default an empty array `[]`
2. **optional** maximum number of items per page, by default `25`
3. **optional** current page, by default `1`

```
{% set documents = location.filterChildren(['document'], 10, 1) %}

<h3>Children documents, page {{ documents.currentPage }}</h3>
<p>Total: {{ documents.nbResults }} items</p>

<ul>
{% for document in documents %}
    <a href="{{ path(document) }}">{{ document.name }}</a>
{% endfor %}
</ul>

{{ pagerfanta( documents, 'twitter_bootstrap' ) }}
```

Location siblings

- **Listing Location's siblings**

  This is done by calling the method `getSiblings()`, also available as `siblings()` in Twig. It returns an array of children Locations and optionally accepts maximum number of items returned (by default `25`).

```
{% set children = location.siblings(10) %}

<h3>List of 10 Location's siblings, sorted as is defined on the parent Location</
↪h3>

<ul>
{% for sibling in siblings %}
    <a href="{{ path(sibling) }}">{{ sibling.name }}</a>
{% endfor %}
</ul>
```

- **Filtering through Location's siblings**

  This is done by calling the method `filterSiblings()`, which returns a Pagerfanta instance and has three parameters:

1. **optional** array of ContentType identifiers that will be used to filter the result, by default an empty array `[]`
2. **optional** maximum number of items per page, by default `25`
3. **optional** current page, by default `1`

```
{% set articles = location.filterSiblings(['article'], 10, 1) %}

<h3>Sibling articles, page {{ articles.currentPage }}</h3>
<p>Total: {{ articles.nbResults }} items</p>

<ul>
{% for article in articles %}
    <a href="{{ path(articles) }}">{{ articles.name }}</a>
{% endfor %}
</ul>
```

(continues on next page)

(continued from previous page)

```
{{ pagerfanta( articles, 'twitter_bootstrap' ) }}
```

2.1.4 Query Types

Site API Query Types expand upon Query Type feature from eZ Publish Kernel, using the same basic interfaces. That will enable using your existing Query Types, but how Site API integrates them with the rest of the system differs from eZ Publish Kernel.

Built-in Site API Query Types

A number of generic Query Types is provided out of the box. We can separate these into three groups:

General purpose

General purpose Content fetch

This Query Type is used to build general purpose Location queries.

Identifier	`SiteAPI:Location/Fetch`
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

Common Content conditions

`content_type`

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`

- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`
- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

limit

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`

- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

offset

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

- *Location depth*
- *Content Field*
- *Content modification date*

- *Content name*
- *Location priority*
- *Content publication date*

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

General purpose Location fetch

This Query Type is used to build general purpose Location queries.

Identifier	`SiteAPI:Location/Fetch`
Inherited Location conditions	• *depth* • *main* • *parent_location_id* • *priority* • *subtree* • *visible*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

Inherited Location conditions

depth

Defines absolute depth of the Location in the tree.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
depth: 3
```

```
depth:
    eq: 3
```

```
# identical to the example below
depth: [3, 4, 8]
```

```
depth:
    in: [3, 4, 8]
```

```
# multiple operators are combined with logical AND
depth:
    in: [3, 4, 5]
    gt: 4
    lte: 8
```

```
depth:
    between: [4, 7]
```

main

Defines whether returned Locations are main Locations or not. Use `true` to get main Locations, `false` to get non-main Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
main: true
```

```
main:
    eq: true
```

```
# get both main and non-main Locations, which is also the default behaviour
main: ~
```

parent_location_id

Defines Location's parent Location ID.

- **value type**: `integer`, `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
parent_location_id: 42
```

```
parent_location_id:
    eq: 42
```

```
# identical to the example below
parent_location_id: [11, 24, 42]
```

```
parent_location_id:
    in: [11, 24, 42]
```

priority

Defines the priority of the Location.

- **value type**: `integer`
- **value format**: `single`
- **operators**: `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# multiple operators are combined with logical AND
depth:
    gt: 4
    lte: 8
```

```
depth:
    between: [4, 7]
```

subtree

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

bla bla

visible

Defines whether returned Locations are visible or not. Use `true` to get visible Locations, `false` to get hidden Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
visible: false
```

```
visible:
    eq: false
```

```
# get both visible and hidden Locations, which also the default behaviour
visible: ~
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`
- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

limit

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

offset

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a

string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Content relations

All tag fields Content relations Query Type

This Query Type is used to build queries that fetch Content tag field relations from all tag fields of a given Content.

Hint: Tag field Content relations are Content items tagged with a tag contained in the tag fields of a given Content.

Hint: This query type assumes Netgen's TagsBundle is used for tagging functionality.

Identifier	`SiteAPI:Content/Relations/AllTagFields`
Own conditions	• *content* • *exclude_self*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

On full view for `product` type Content fetch all Content of type `article` that is tagged with any of the tags from the given product. Sort them by name and paginate them by 10 per page using URL query parameter `page`:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    product:
                        template: '@ezdesign/content/full/product.html.twig'
                        match:
                            Identifier\ContentType: product
                        queries:
                            related_articles:
                                query_type: SiteAPI:Content/Relations/AllTagFields
                                max_per_page: 10
                                page: '@=queryParam("page", 1)'
                                parameters:
                                    content_type: article
                                    sort: name
```

```
{% set articles = ng_query( 'related_articles' ) %}

<h3>Related articles</h3>

<ul>
{% for article in articles %}
    <li>{{ article.name }}</li>
{% endfor %}
</ul>

{{ pagerfanta( articles, 'twitter_bootstrap' ) }}
```

Own conditions

content

Defines the source (from) relation Content, which is the one containing tag fields.

Note: This condition is required. It's also automatically set to the `Content` instance resolved by the view builder if the query is defined in the view builder configuration.

- **value type**: `Content`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
# this is also automatically set when using from view builder configuration
location: '@=content'
```

```
# fetch relations from Content's main Location parent Location's Content
location: '@=content.mainLocation.parent.content'
```

```
# fetch relations from Content's main Location parent Location's parent Location's␣
↪Content
location: '@=content.mainLocation.parent.parent.content'
```

exclude_self

Defines whether to include Content defined by the `content` condition in the result set.

- **value type**: `boolean`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `false`
- **default**: `true`

Examples:

```
# do not include the source relation Content, this is also the default behaviour
exclude_self: true
```

```
# include the source relation Content
exclude_self: false
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`
- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

limit

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

offset

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a

string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Forward field Content relations Query Type

This Query Type is used to build fetch Content that is related to from relation type fields of the given Content.

Identifier	`SiteAPI:Content/Relations/ForwardFields`
Own conditions	• *content* • *relation_field*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

Content of type `blog_post` has relation field `images` which is used to define relations to `image` type Content. On full view for `blog_post` fetch 10 related images sorted by name and paginate them by 10 per page using URL query parameter `page`.

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    blog_post:
                        template: '@ezdesign/content/full/blog_post.html.twig'
                        match:
                            Identifier\ContentType: blog_post
                        queries:
                            related_images:
                                query_type: SiteAPI:Content/Relations/ForwardFields
                                max_per_page: 10
                                page: 1
                                parameters:
                                    relation_field: images
                                    content_type: image
                                    sort: name
```

```
<h3>Related images</h3>

<ul>
{% for image in ng_query( 'related_images' ) %}
    <li>
        {{ ng_image_alias( image.fields.image, 'gallery' ) }}
    </li>
{% endfor %}
</ul>

{{ pagerfanta( documents, 'twitter_bootstrap' ) }}
```

Own conditions

content

Defines the source (from) relation Content, which is the one containing relation type fields.

Note: This condition is required. It's also automatically set to the `Content` instance resolved by the view builder if the query is defined in the view builder configuration.

- **value type**: `Content`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
# this is also automatically set when using from view builder configuration
location: '@=content'
```

```
# fetch relations from Content's main Location parent Location's Content
location: '@=content.mainLocation.parent.content'
```

```
# fetch relations from Content's main Location parent Location's parent Location's␣
↪Content
location: '@=content.mainLocation.parent.parent.content'
```

relation_field

Defines Content fields to take into account for determining relations.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
relation_field: appellation
```

```
relation_field: [head, heart, base]
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`
- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

limit

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

offset

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a

string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Reverse field Content relations Query Type

This Query Type is used to build fetch Content that relates to the given Content from its relation type fields.

Identifier	`SiteAPI:Content/Relations/ReverseFields`
Own conditions	• *content* • *relation_field*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

Content of type `article` has relation field `authors` which is used to define relations to `author` type Content. On full view for `author` fetch all articles authored by that author, sort them by title and paginate them by 10 per page using URL query parameter `page`:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    author:
                        template: '@ezdesign/content/full/author.html.twig'
                        match:
                            Identifier\ContentType: author
                        queries:
                            authored_articles:
                                query_type: SiteAPI:Content/Relations/ReverseFields
                                max_per_page: 10
                                page: '@=queryParam("page", 1)'
                                parameters:
                                    relation_field: authors
                                    content_type: article
                                    sort: field/article/title asc
```

```
<h3>Author's articles</h3>

<ul>
{% for article in ng_query( 'authored_articles' ) %}
    <li>{{ article.name }}</li>
{% endfor %}
</ul>

{{ pagerfanta( children, 'twitter_bootstrap' ) }}
```

Own parameters

content

Defines the destination (to) relation Content.

Note: This condition is required. It's also automatically set to the `Content` instance resolved by the view builder if the query is defined in the view builder configuration.

Note: Since this is about **reverse** relations, Content defined by this condition is **not** the one containing relation type fields referenced by `relation_field`. It's the one receiving relations from Content containing those fields.

- **value type**: `Content`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
# this is also automatically set when using from view builder configuration
location: '@=content'
```

```
# fetch relations to Content's main Location parent Location's Content
location: '@=content.mainLocation.parent.content'
```

```
# fetch relations to Content's main Location parent Location's parent Location's␣
↪Content
location: '@=content.mainLocation.parent.parent.content'
```

relation_field

Defines Content fields to take into account for determining relations.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
relation_field: authors
```

```
relation_field: [color, size]
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
```

(continues on next page)

(continued from previous page)

```
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none

- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`
- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
```

(continues on next page)

(continued from previous page)

```
    approval:
        eq: rejected
```

Common query parameters

limit

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

offset

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`

- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Tag field Content relations Query Type

This Query Type is used to build queries that fetch Content tag field relations from selected tag fields of a given Content.

Hint: Tag field Content relations are Content items tagged with a tag contained in a tag field of a given Content.

Hint: This query type assumes Netgen's TagsBundle is used for tagging functionality.

Identifier	`SiteAPI:Content/Relations/TagFields`
Own conditions	• *content* • *exclude_self* • *relation_field*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

Your project is a web shop, where Content of type `product` is tagged with tags that define product's market. Specific tag field named `market` is used for that. For example, you could have a wireless keyboard product tagged with market tag `components`. Various other Content is also tagged with that tag, for example we could have files and articles using that same tag.

On the full view for Content of type `product`, fetch articles from the same market, sort them by their publication date and paginate them by 10 per page using URL query parameter `page`:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    product:
                        template: '@ezdesign/content/full/product.html.twig'
                        match:
                            Identifier\ContentType: product
                        queries:
                            market_articles:
                                query_type: SiteAPI:Content/Relations/TagFields
                                max_per_page: 10
                                page: '@=queryParam("page", 1)'
                                parameters:
                                    relation_field: market
                                    content_type: article
                                    sort: published desc
```

```
{% set articles = ng_query( 'market_articles' ) %}

<h3>Related market articles</h3>

<ul>
{% for article in articles %}
    <li>{{ article.name }}</li>
```

(continues on next page)

(continued from previous page)

```
{% endfor %}
</ul>

{{ pagerfanta( articles, 'twitter_bootstrap' ) }}
```

Own conditions

content

Defines the source (from) relation Content, which is the one containing tag fields.

Note: This condition is required. It's also automatically set to the `Content` instance resolved by the view builder if the query is defined in the view builder configuration.

- **value type**: `Content`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
# this is also automatically set when using from view builder configuration
location: '@=content'
```

```
# fetch relations from Content's main Location parent Location's Content
location: '@=content.mainLocation.parent.content'
```

```
# fetch relations from Content's main Location parent Location's parent Location's␣
↪Content
location: '@=content.mainLocation.parent.parent.content'
```

exclude_self

Defines whether to include Content defined by the `content` condition in the result set.

- **value type**: `boolean`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `false`
- **default**: `true`

Examples:

```
# do not include the source relation Content, this is also the default behaviour
exclude_self: true
```

```
# include the source relation Content
exclude_self: false
```

relation_field

Defines Content fields to take into account for determining relations.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
relation_field: appellation
```

```
relation_field: [head, heart, base]
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`
- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier

- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

limit

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

offset

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

- *Location depth*
- *Content Field*
- *Content modification date*
- *Content name*
- *Location priority*
- *Content publication date*

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Location hierarchy

Location children Query Type

This Query Type is used to build queries that fetch children Locations.

Identifier	`SiteAPI:Location/Children`
Own conditions	• *location*
Inherited Location conditions	• *main* • *priority* • *visible*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

On full view for `folder` type Location fetch folder's children Locations of the type `document` that are in `restricted` Section, sort them by priority descending and paginate them by 10 per page using URL query parameter `page`:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
```

(continues on next page)

(continued from previous page)

```
                full:
                    folder:
                        template: '@ezdesign/content/full/folder.html.twig'
                        match:
                            Identifier\ContentType: folder
                        queries:
                            children_documents:
                                query_type: SiteAPI:Content/Location/Children
                                max_per_page: 10
                                page: '@=queryParam("page", 1)'
                                parameters:
                                    content_type: document
                                    section: restricted
                                    sort: priority desc
```

```
{% set documents = ng_query( 'children_documents' ) %}

<h3>Documents in this folder</h3>

<ul>
{% for document in documents %}
    <li>{{ document.name }}</li>
{% endfor %}
</ul>

{{ pagerfanta( documents, 'twitter_bootstrap' ) }}
```

Own conditions

location

Defines the parent Location for children Locations.

Note: This condition is required. It's also automatically set to the `Location` instance resolved by the view builder if the query is defined in the view builder configuration.

- **value type**: `Location`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
# this is also automatically set when using from view builder configuration
location: '@=location'
```

```
# fetch children of the parent Location
location: '@=location.parent'
```

```
# fetch children of the parent Location's parent Location
location: '@=location.parent.parent'
```

Inherited Location conditions

main

Defines whether returned Locations are main Locations or not. Use `true` to get main Locations, `false` to get non-main Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
main: true
```

```
main:
    eq: true
```

```
# get both main and non-main Locations, which is also the default behaviour
main: ~
```

priority

Defines the priority of the Location.

- **value type**: `integer`
- **value format**: `single`
- **operators**: `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# multiple operators are combined with logical AND
depth:
    gt: 4
    lte: 8
```

```
depth:
    between: [4, 7]
```

visible

Defines whether returned Locations are visible or not. Use `true` to get visible Locations, `false` to get hidden Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
visible: false
```

```
visible:
    eq: false
```

```
# get both visible and hidden Locations, which also the default behaviour
visible: ~
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`

- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

`limit`

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

`offset`

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

- *Location depth*
- *Content Field*
- *Content modification date*
- *Content name*
- *Location priority*
- *Content publication date*

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Location siblings Query Type

This Query Type is used to build queries that fetch Location siblings.

Identifier	`SiteAPI:Location/Siblings`
Own conditions	• *location*
Inherited Location conditions	• *main* • *priority* • *visible*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

On the full view for `article` type Content fetch all siblings of type `news` that are in ObjectState `review/approved`, sort them by name and paginate them by 10 per page using URL query parameter `page`:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    article:
                        template: '@ezdesign/content/full/article.html.twig'
                        match:
```

(continues on next page)

(continued from previous page)

```
                            Identifier\ContentType: article
                    queries:
                        news_siblings:
                            query_type: SiteAPI:Content/Location/Siblings
                            max_per_page: 10
                            page: '@=queryParam("page", 1)'
                            parameters:
                                content_type: news
                                state:
                                    review: approved
                                sort: name
```

```
{% set news_list = ng_query( 'news_siblings' ) %}

<h3>Article's news siblings</h3>

<ul>
{% for news in news_list %}
    <li>{{ news.name }}</li>
{% endfor %}
</ul>

{{ pagerfanta( news_list, 'twitter_bootstrap' ) }}
```

Own conditions

`location`

Defines sibling Location reference for fetching other siblings Locations.

Note: This condition is required. It's also automatically set to the `Location` instance resolved by the view builder if the query is defined in the view builder configuration.

- **value type**: `Location`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
# this is also automatically set when using from view builder configuration
location: '@=location'
```

```
# fetch siblings of the parent Location
location: '@=location.parent'
```

```
# fetch siblings of the parent Location's parent Location
location: '@=location.parent.parent'
```

Inherited Location conditions

main

Defines whether returned Locations are main Locations or not. Use `true` to get main Locations, `false` to get non-main Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
main: true
```

```
main:
    eq: true
```

```
# get both main and non-main Locations, which is also the default behaviour
main: ~
```

priority

Defines the priority of the Location.

- **value type**: `integer`
- **value format**: `single`
- **operators**: `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# multiple operators are combined with logical AND
depth:
    gt: 4
    lte: 8
```

```
depth:
    between: [4, 7]
```

visible

Defines whether returned Locations are visible or not. Use `true` to get visible Locations, `false` to get hidden Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
visible: false
```

```
visible:
    eq: false
```

```
# get both visible and hidden Locations, which also the default behaviour
visible: ~
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`

- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

`limit`

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

`offset`

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

- *Location depth*
- *Content Field*
- *Content modification date*
- *Content name*
- *Location priority*
- *Content publication date*

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Location subtree Query Type

This Query Type is used to build queries that fetch from the Location subtree.

Identifier	`SiteAPI:Location/Subtree`
Own conditions	• *exclude_self* • *location* • *relative_depth*
Inherited Location conditions	• *depth* • *main* • *priority* • *visible*
Common Content conditions	• *content_type* • *field* • *publication_date* • *section* • *state*
Common query parameters	• *limit* • *offset* • *sort*

Examples

Subtree of the `calendar` type Location contains `event` type Locations. On the full view for `calendar` fetch all pending events from its subtree up to depth of 3, sort them by their start date and paginate them by 10 per page using URL query parameter `page`:

```
ezpublish:
    system:
        frontend_group:
```

(continues on next page)

(continued from previous page)

```
            ngcontent_view:
                full:
                    calendar:
                        template: '@ezdesign/content/full/calendar.html.twig'
                        match:
                            Identifier\ContentType: calendar
                        queries:
                            pending_events:
                                query_type: SiteAPI:Content/Location/Subtree
                                max_per_page: 10
                                page: '@=queryParam("page", 1)'
                                parameters:
                                    content_type: event
                                    relative_depth:
                                        lte: 3
                                    field:
                                        start_date:
                                            gt: '@=timestamp("today")'
                                    sort: field/event/start_date asc
```

```
{% set events = ng_query( 'pending_events' ) %}

<h3>Pending events</h3>

<ul>
{% for event in events %}
    <li>{{ event.name }}</li>
{% endfor %}
</ul>

{{ pagerfanta( events, 'twitter_bootstrap' ) }}
```

Own conditions

exclude_self

Defines whether to include Location defined by the `location` condition in the result set.

- **value type**: `boolean`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `false`
- **default**: `true`

Examples:

```
# do not include the subtree root Location, this is also default behaviour
exclude_self: true
```

```
# include the subtree root Location
exclude_self: false
```

location

Defines the root Location of the Location subtree.

Note: This condition is required. It's also automatically set to the `Location` instance resolved by the view builder if the query is defined in the view builder configuration.

- **value type**: `Location`
- **value format**: `single`
- **operators**: none
- **target**: none
- **required**: `true`
- **default**: not defined

Examples:

```
# this is also automatically set when using from view builder configuration
location: '@=location'
```

```
# fetch from subtree of the parent Location
location: '@=location.parent'
```

```
# fetch from subtree of the parent Location's parent Location
location: '@=location.parent.parent'
```

relative_depth

Defines depth of the Location in the tree relative to the Location defined by `location` condition.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
depth: 2
```

```
depth:
    eq: 2
```

```
# identical to the example below
depth: [2, 3]
```

```
depth:
    in: [2, 3]
```

```
# multiple operators are combined with logical AND
depth:
    in: [2, 3]
    gt: 1
    lte: 3
```

```
depth:
    between: [2, 4]
```

Inherited Location conditions

depth

Defines absolute depth of the Location in the tree.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
depth: 3
```

```
depth:
    eq: 3
```

```
# identical to the example below
depth: [3, 4, 8]
```

```
depth:
    in: [3, 4, 8]
```

```
# multiple operators are combined with logical AND
depth:
    in: [3, 4, 5]
    gt: 4
    lte: 8
```

```
depth:
    between: [4, 7]
```

main

Defines whether returned Locations are main Locations or not. Use `true` to get main Locations, `false` to get non-main Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
main: true
```

```
main:
    eq: true
```

```
# get both main and non-main Locations, which is also the default behaviour
main: ~
```

priority

Defines the priority of the Location.

- **value type**: `integer`
- **value format**: `single`
- **operators**: `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# multiple operators are combined with logical AND
depth:
    gt: 4
    lte: 8
```

```
depth:
    between: [4, 7]
```

visible

Defines whether returned Locations are visible or not. Use `true` to get visible Locations, `false` to get hidden Locations and `null` to get both (which is also the default behaviour).

- **value type**: `boolean`, `null`
- **value format**: `single`
- **operators**: `eq`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
visible: false
```

```
visible:
    eq: false
```

```
# get both visible and hidden Locations, which also the default behaviour
visible: ~
```

Common Content conditions

content_type

Defines ContentType of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: `string` ContentType identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
content_type: article
```

```
content_type:
    eq: article
```

```
# identical to the example below
content_type: [image, video]
```

```
content_type:
    in: [image, video]
```

field

Defines conditions on Content fields.

- **value type**: `integer`, `string`, `boolean`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`, `like`, `contains`
- **target**: `string` Field identifier
- **required**: `false`
- **default**: not defined

Examples:

```
field:
    date_field:
        not:
            gt: 'today +5 days'
    price:
        between: [100, 200]
        not: 155
```

publication_date

Defines the publication date of the Content as a timestamp.

- **value type**: `integer`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`, `gt`, `gte`, `lt`, `lte`, `between`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
publication_date: 1535117737
```

```
depth:
    eq: 1535117737
```

```
# identical to the example below
publication_date: [1435117737, 1535117737]
```

```
publication_date:
    in: [1435117737, 1535117737]
```

```
# multiple operators are combined with logical AND
publication_date:
    gt: '29 June 1991'
    lte: '5 August 1995'
```

```
publication_date:
    gt: 'today'
```

```
publication_date:
    between: ['today', '+1 week 2 days 4 hours 2 seconds']
```

section

Defines Section of the Content by the identifier.

- **value type**: `string`
- **value format**: `single`, `array`
- **operators**: `eq`, `in`
- **target**: none
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
section: standard
```

```
section:
    eq: standard
```

```
# identical to the example below
section: [standard, restricted]
```

```
section:
    in: [standard, restricted]
```

state

Defines ObjectState of the Content by the ObjectStateGroup and ObjectState identifiers.

Note: Content can only exist in single ObjectState from the same ObjectStateGroup.

- **value type**: `string` ObjectState identifier
- **value format**: `single`
- **operators**: `eq`
- **target**: `string` ObjectStateGroup identifier
- **required**: `false`
- **default**: not defined

Examples:

```
# identical to the example below
state:
    ez_lock: not_locked
```

```
state:
    ez_lock:
        eq: not_locked
```

```
# multiple states are combined with logical AND
# identical to the example below
state:
    ez_lock: locked
    approval: rejected
```

```
state:
    ez_lock:
        eq: locked
    approval:
        eq: rejected
```

Common query parameters

limit

Defines the maximum number of items to return.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`
- **value format**: `single`
- **required**: `false`
- **default**: `25`

Examples:

```
limit: 10
```

offset

Defines the offset for search hits, used for paging the results.

Note: This parameter will not be used if you execute the query from Twig using `ng_query` function. In that case `Pargerfanta` pager is used with semantic parameters `page` and `max_per_page`. To execute the query directly use `ng_raw_query` Twig function instead.

- **value type**: `integer`

- **value format**: `single`
- **required**: `false`
- **default**: `0`

Examples:

```
offset: 20
```

sort

- **value type**: `string`, `SortClause`
- **value format**: `single`, `array`
- **required**: `false`
- **default**: not defined

For this parameter you can use any SortClause implementation. But if you define the query in the view configuration, you won't be able to instantiate the SortClause there. For that reason we provide a way to define the sort clause as a string instead. We this format a subset of commonly used SortClauses is supported. Sort direction is defined as `asc` for ascending and `desc` for descending. In can be omitted, in which case it will default to `asc`.

Strings can be used to define multiple sort clauses through an array of definitions:

```
sort:
    - depth asc
    - modified desc
```

Following sort clauses are available through string definition:

- *Location depth*
- *Content Field*
- *Content modification date*
- *Content name*
- *Location priority*
- *Content publication date*

Location depth

String `depth` enables sorting by Location's depth:

```
sort: depth
```

```
sort: depth asc
```

```
sort: depth desc
```

Content Field

String in form of of `field/[content_type]/[field]` enables sorting by any Content Field. For example by Field with identifier `title` in ContentType with identifier `article`:

```
sort: field/article/title
```

```
sort: field/article/title asc
```

```
sort: field/article/title desc
```

Content modification date

String `modified` enables sorting by the Content modification date:

```
sort: modified
```

```
sort: modified asc
```

```
sort: modified desc
```

Content name

String `name` enables sorting by the Content name:

```
sort: name
```

```
sort: name asc
```

```
sort: name desc
```

Location priority

String `priority` enables sorting by the Location priority:

```
sort: priority
```

```
sort: priority asc
```

```
sort: priority desc
```

Content publication date

String `published` enables sorting by the Content publication/creation date:

```
sort: published
```

```
sort: published asc
```

```
sort: published desc
```

Query configuration

Query Types have their own semantic configuration under `queries` key in configuration for a particular Content view. Under this key separate queries are defined under their own identifier keys, which are later used to reference the configured query from the Twig templates.

Available parameters and their default values are:

- `query_type` - identifies the Query Type to be used
- `named_query` - identifies named query to be used
- `max_per_page: 25` - pagination parameter for maximum number of items per page
- `page: 1` - pagination parameter for current page
- `use_filter: true` - whether to use `FilterService` or `FindService` for executing the query
- `parameters: []` - contains the actual Query Type parameters

Parameters `query_type` and `named_query` are mutually exclusive, you are allowed to set only one or the other. But they are also mandatory - you will have to set one of them.

Example below shows how described configuration looks in practice:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    category:
                        template: '@ezdesign/content/full/category.html.twig'
                        match:
                            Identifier\ContentType: 'category'
                        queries:
                            children:
                                query_type: 'SiteAPI:Location/Children'
                                max_per_page: 10
                                page: 1
                                parameters:
                                    content_type: 'article'
                                    sort: 'published desc'
                            related_images:
                                query_type: 'SiteAPI:Content/Relations/ForwardFields'
                                max_per_page: 10
                                page: 1
                                parameters:
                                    content_type: 'image'
                                    sort: 'published desc'
                        params:
                            ...
```

Note: You can define unlimited number of queries on any controller.

Named query configuration

As hinted above with `named_query` parameter, it is possible to define "named queries", which can be referenced in query configuration for a particular content view. They are configured under `ng_named_query`, which is a top section of a siteaccess configuration, on the same level as `ng_content_view`:

```
ezpublish:
    system:
        frontend_group:
            ng_named_query:
                children_named_query:
                    query_type: 'SiteAPI:Location/Children'
                    max_per_page: 10
                    page: 1
                    parameters:
                        content_type: 'article'
                        sort: 'published desc'
            ngcontent_view:
                full:
                    category:
                        template: '@ezdesign/content/full/category.html.twig'
                        match:
                            Identifier\ContentType: 'category'
                        queries:
                            children: 'children_named_query'
                            children_5_per_page:
                                named_query: 'children_named_query'
                                max_per_page: 5
                            images:
                                named_query: 'children_named_query'
                                parameters:
                                    content_type: 'image'
                        params:
                            ...
```

Note: You can override some of the parameters from the referenced named query.

You can notice that there are two ways of referencing a named query. In case when there are no other parameters, you can do it directly like this:.

```
queries:
    children: 'children_named_query'
```

The example above is really just a shortcut to the example below:

```
queries:
    children:
        named_query: 'children_named_query'
```

You can also notice that it's possible to override parameters from the referenced named query. This is limited to first level keys from the main configuration and also first level keys under the `parameters` key.

Parameters with expressions

When defining parameters it's possible to use expressions. These are evaluated by Symfony's Expression Language component, whose syntax is based on Twig and documented here.

Expression strings are recognized by `@=` prefix. Following values resolved from the current view will be available in expression:

- Site API view object as `view`

  You can access view object and any parameters injected into it, for example current page in children query:

```
...
    queries:
        children:
            query_type: 'SiteAPI:Location/Children'
            max_per_page: 10
            page: '@=view.getParameter("page")'
            parameters:
                content_type: 'article'
                sort: 'published desc'
```

- Symfony's Request object as `request`

  Similar to the above, you could access current page directly from the parameter in the Request object:

```
...
    queries:
        children:
            query_type: 'SiteAPI:Location/Children'
            max_per_page: 10
            page: '@=request.query.get("page", 1)'
            parameters:
                content_type: 'article'
                sort: 'published desc'
```

- *Site API Content object* as `content`

  Full Content object is available, for example you could store ContentType identifier for the children in a TextLine field `content_type` and access it like this:

```
...
    queries:
        children:
            query_type: 'SiteAPI:Location/Children'
            max_per_page: 10
            page: 1
            parameters:
                content_type: '@=content.fields.content_type.value.text'
                sort: 'published desc'
```

- *Site API Location object* as `location`

  Full Location object is also available, in the following example we use it to find only children of the same ContentType as the parent:

```
...
    queries:
```

(continues on next page)

(continued from previous page)

```
        children:
            query_type: 'SiteAPI:Location/Children'
            max_per_page: 10
            page: 1
            parameters:
                content_type: '@=location.contentInfo.
↪contentTypeIdentifier'
                sort: 'published desc'
```

Several functions are also available for use in expressions. Most of these are provided to access the values described above in a more convenient way:

- `viewParam(name, default)`

  Method `getParameter()` on the View object does not support default value fallback and if the requested parameter is not there an exception will be thrown. Function `viewParam()` is just a wrapper around it that provides default value fallback:

```
...
    queries:
        children:
            query_type: 'SiteAPI:Location/Children'
            max_per_page: 10
            page: '@=viewParam("page", 10)'
            parameters:
                content_type: 'article'
                sort: 'published desc'
```

- `queryParam(name, default)`

  This function is just a shortcut to `GET` parameters on the Request object:

```
...
    queries:
        children:
            query_type: 'SiteAPI:Location/Children'
            max_per_page: 10
            page: '@=queryParam("page", 1)'
            parameters:
                content_type: 'article'
                sort: 'published desc'
```

- `timestamp(value)`

  This function is used to get a timestamp value, typically used to define time conditions on the query. For example you could use it to fetch only events that have not yet started:

```
...
    queries:
        pending_events:
            query_type: SiteAPI:Content/Location/Subtree
            max_per_page: 10
            page: 1
            parameters:
                content_type: event
                field:
                    start_date:
                        gt: '@=timestamp("today")'
```

Note: Function `timestamp()` maps directly to the PHP's function strtotime. That means you can pass it any supported date and time format.

Templating

Configured queries will be available in Twig templates, through `ng_query` or `ng_raw_query`. The difference it that the former will return a `Pagerfanta` instance, while the latter will return an instance of `SerachResult`. That also means `ng_query` will use `max_per_page` and `page` parameters to configure the pager, while `ng_raw_query` ignores them and executes the configured query directly.

Note: Queries are only executed as you access them through `ng_query` or `ng_raw_query`. If you don't call those functions on any of the configured queries, none of them will be executed.

Both `ng_query` and `ng_raw_query` accept a single argument. This is the identifier of the query, which is the key under the `queries` section, under which the query is configured.

Example usage of `ng_query`:

```
{% set images = ng_query( 'images' ) %}

<p>Total images: {{ images.nbResults }}</p>

{% for image in images %}
    <p>{{ image.content.name }}</p>
{% endfor %}

{{ pagerfanta( images, 'twitter_bootstrap' ) }}
```

Example usage of `ng_raw_query`:

```
{% set searchResult = ng_raw_query( 'categories' ) %}

{% for categoryHit in searchResult.searchHits %}
    <p>{{ categoryHit.valueObject.content.name }}: {{ categoryHit.valueObject.score }}
↪</p>
{% endfor %}
```

Note: You can't execute named queries. They are only available for referencing in concrete query configuration for a particular view.

Hint: Execution of queries is **not cached**. If you call `ng_query` or `ng_raw_query` on the same query multiple times, the same query will be executed multiple times. If you need to access the query result multiple times, store it in a variable and access the variable instead.

2.1.5 Objects

Site API comes with it's own set of entities and values. These are similar, but still different from their counterparts in eZ Platform's Repository API. Main benefits they provide over them are:

- Content is available in a single translation, this voids the need for various helper services
- Additional properties otherwise available only through separate entities (like ContentType identifier, FieldType identifier and others)
- Additional properties and methods that enable simple traversal and filtering of the content model (relations, parent, siblings, children)

Note: Note that content traversal that is achievable through the objects is not complete. It aims to cover only the most common use cases. For more complex use cases *Query Types* should be used.

Note: In Twig templates methods beginning with `get` and `is` are also available with that prefix removed. Also, parentheses can be omitted if there are no required arguments.

For example, method `field.isEmpty()` is also available as `field.empty()` or just `field.empty`, and method `content.getLocations()` is available as `content.locations()` or just `content.locations`.

Content on this page:

        * *getChildren*
        * *filterChildren*
        * *getSiblings*
        * *filterSiblings*
    - *Properties*

Content

The first difference from Repository Content is that it exist it a single translation only, meaning it contains the fields for only one translation. That will always be the translation to be rendered on the siteaccess. You won't need to choose the field in the correct translation, manually or through some kind of helper service. The Content's single translation is always the correct one.

Content fields are lazy-loaded, which means they are initially not loaded, but will be transparently loaded at the point you access them. This voids the need to have separate, lightweight version of Content (ContentInfo plays this role in Repository API). It also provides you with some additional properties and methods.

Example usage from Twig:

```
<h1>{{ content.name }}</h1>
<h2>Parent name: {{ content.mainLocation.parent.content.name }}</h2>
<h3>Number of Locations: {{ content.locations|length }}</h3>

{% for field in content.fields %}
    {% if not field.empty %}
        {{ ng_render_field(field) }}
    {% endif %}
{% endfor %}
```

Methods

- *hasField*
- *getField*
- *hasFieldById*
- *getFieldById*
- *getFieldValue*
- *getFieldValueById*
- *getLocations*
- *filterLocations*
- *getFieldRelation*
- *getFieldRelations*
- *filterFieldRelations*

hasField

Check if Content has a *Field* with the given `$identifier`.

Parameters	`string $identifier`
Returns	`bool`
Example in PHP	`if ($content->hasField('title')) { // ... }`
Example in Twig	`{% if content.hasField('title') %} ... {% endif %}`

getField

Get the *Field* with the given `$identifier`.

Note: This method can return `null` if Field with the given `$identifier` doesn't exist.

Parameters	`string $identifier`
Returns	*Field* instance or `null`
Example in PHP	`$field = $content->getField('title');`
Example in Twig	`{{ set field = content.field('title') }}`

hasFieldById

Check if Content has a *Field* with the given `$id`.

Parameters	`int|string $id`
Returns	`bool`
Example in PHP	`$content->hasFieldById(42);`
Example in Twig	`{{ content.hasFieldById(42) }}`

getFieldById

Get the *Field* with the given `$id`.

Note: This method can return `null` if Field with the given `$id` doesn't exist.

Parameters	`string $id`
Returns	*Field* instance or `null`
Example in PHP	`$field = $content->getFieldById(42));`
Example in Twig	`{% set field = content.fieldById(42) %}`

getFieldValue

Get the value of the *Field* with the given `$identifier`.

Note: This method can return `null` if Field with the given `$identifier` doesn't exist.

Note: Returned value object depends of the FieldType. Best way to learn about the specific value format is reading the official FieldType reference documentation, or looking directly at code (for example the code of TextLine Value).

Parameters	`string $identifier`
Returns	Value instance of the *Field* or `null`
Example in PHP	`$value = $content->getFieldValue('title↪'));`
Example in Twig	`{% set value = content.fieldValue('title↪') %}`

getFieldValueById

Get the value of the *Field* with the given `$id`.

Note: This method can return `null` if Field with the given `$id` doesn't exist.

Parameters	`string $id`
Returns	Value instance of the *Field* or `null`
Example in PHP	`$value = $content->getFieldValueById(42));`
Example in Twig	`{% set value = content.fieldValueById(42) %}`

getLocations

Used to get Content's Locations, limited by the `$limit`. Locations will be sorted their path string (a string with materialized IDs, e.g. `/1/2/45/67/`).

Parameters	`int $limit = 25`
Returns	An array of Content's *Locations*
Sorting method	Location's path string (e.g. `/1/2/45/67/`)
Example in PHP	`$locations = $content->locations(10));`
Example in Twig	`{% set locations = content.locations %}`

filterLocations

List a slice of Content's Locations, by the `$maxPerPage` and `$currentPage`. Locations will be sorted their path string (a string with materialized IDs, e.g. `/1/2/45/67/`).

Parameters	1. `int $maxPerPage = 25` 2. `int $currentPage = 1`
Returns	Pagerfanta instance with a slice of Content's *Locations*
Sorting method	Location's path string (e.g. `/1/2/45/67/`)
Example in PHP	`$locations = $content->filterLocations(10, 2));`
Example in Twig	`{% set locations = content.filterLocations(10, 2) %}`

getFieldRelation

Used to get a single field relation from the *Field* with the given `$identifier`.

Parameters	`string $identifier`
Returns	Related *Content* or `null` if the relation does not exist
Example in PHP	`$relation = $content->getFieldRelation('author'));`
Example in Twig	`{% set relation = content.fieldRelation('author') %}`

getFieldRelations

Used to get `$limit` field relations from the *Field* with the given `$identifier`. Relations will be sorted as is defined by the relation field.

Parameters	1. `string $identifier` 2. `int $limit = 25`
Returns	An array of related *Content* items
Sorting method	Sorted as is defined by the relation *Field*
Example in PHP	`$relations = $content->getFieldRelations('images', 10));`
Example in Twig	`{% set relations = content.fieldRelation('images') %}`

filterFieldRelations

Used to filter field relations from the *Field* with the given `$identifier`.

Parameters	1. `string $identifier` 2. `array $contentTypeIdentifiers = []` 3. `int $maxPerPage = 25` 4. `int $currentPage = 1`
Returns	Pagerfanta instance with related Content items
Example in PHP	`$relations = $content->filterFieldRelations('related_items', ['images', 'videos'], 10, 2);`
Example in Twig	`{% set relations = content.fieldRelation('related_items' ['images', 'videos'] 10, 2) %}`

Properties

Name	Type	Description
`$id`	`string|int`	ID
`$mainLocationId`	`string|int|null`	Optional main *Location* ID
`$name`	`string`	Name
`$languageCode`	`string`	Translation language code
`$contentInfo`	*ContentInfo*	ContentInfo object
`$fields`	`Field[]`	An array of *Field* instances, which can be accessed in two different ways: `{{ set field = content.fields.title }}` `{{ set field = content.fields['title'] }}`
`$mainLocation`	*Location*	Optional Location object
`$owner`	*Content*	Optional owner user's Content object

`ContentInfo`

Site `ContentInfo` object is similar to the Repository ContentInfo, additionally providing access to

Properties

Name	Type	Description
`$id`	`string|int`	ID of the Content
`$contentTypeId`	`string|int`	ID of the ContentType
`$sectionId`	`string|int`	ID of the Section
`$currentVersionNo`	`int`	Current version number
`$published`	`bool`	Indicates that the Content is published
`$ownerId`	`string|int`	ID of the owner user Content
`$modificationDate`	`\DateTime`	Modification date
`$publishedDate`	`\DateTime`	Publication date
`$alwaysAvailable`	`bool`	Indicates that the Content is always available in it's main translation
`$remoteId`	`string`	Remote ID of the Content
`$mainLanguageCode`	`string`	Main translation language code
`$mainLocationId`	`string|int`	ID of the main Location
`$name`	`string`	Content's name
`$languageCode`	`string`	Language code of Content's translation
`$contentTypeIdentifier`	`string`	Identifier of the Content Type
`$contentTypeName`	`string`	Name of the Content Type
`$contentTypeDescription`	`string`	Description of the Content Type
`$mainLocation`	*Location*	Content's main Location object
`$content`	*Content*	Content object

`Field`

Site `Field` object is similar to the Repository Field, additionally providing access to the field's *Content* and properties that are otherwise available only through the corresponding FieldDefinition object: name, description and FieldType identifier.

Methods

`isEmpty`

Checks if the field's value is empty.

Parameters	None
Returns	`bool`
Example in PHP	`if ($content->getField('title')->isEmpty()) { // ... }`
Example in Twig	`{% if content.fields.title.empty %} ... {% endif %}`

Properties

Name	Type	Description
`$id`	`string|int`	ID of the Field
`$fieldDefIdentifier`	`string`	Identifier (FieldDefinition identifier, e.g. `title`)
`$value`	Value object	Value object
`$languageCode`	`string`	Translation language code
`$fieldTypeIdentifier`	`string`	FieldType identifier (e.g. `ezstring`)
`$name`	`string`	ID of the Content
`$description`	`string`	ID of the Content
`$content`	*Content*	ID of the Content

`Location`

Site `Location` object is similar to the Repository Location, additionally providing methods and properties that enable simple traversal and filtering of the Location tree (siblings, children, parent, ancestors etc).

Methods

`getChildren`

List children Locations.

Children will be sorted as is defined by their parent Location, which is the Location the method is called on. The single optional parameter of this method is `$limit`, which limits the number of children returned and defaults to `25`.

Parameters	`string $limit = 25`
Returns	An array of first `$limit` children Locations
Sorting method	As is defined by the Location
Example in PHP	`$children = $location->getChildren(10));`
Example in Twig	`{% set children = location.children(10) %}`

filterChildren

Filter and paginate children Locations.

This enables filtering of the children by their ContentType with `$contentTypeIdentifiers` parameter and pagination using `$maxPerPage` and `$currentPage` parameters. The method returns a Pagerfanta instance.

Parameters	1. `array $contentTypeIdentifiers = []` 2. `int $maxPerPage = 25` 3. `int $currentPage = 1`
Returns	Pagerfanta instance with a slice of children Locations
Sorting method	As is defined by the Location
Example in PHP	`$children = $content->filterChildren(['articles'], 10, 2);`
Example in Twig	`{% set relation = content.filterChildren( ['articles'], 10, 2 ) %}`

getSiblings

List sibling Locations.

Siblings will be sorted as is defined by their parent Location, which is the parent Location of the Location the method is called on. The single optional parameter of this method is `$limit`, which limits the number of siblings returned and defaults to `25`.

Parameters	`string $limit = 25`
Returns	An array of first `$limit` sibling Locations
Sorting method	As is defined by the parent Location
Example in PHP	`$siblings = $location->getSiblings(10);`
Example in Twig	`{% set siblings = location.siblings(10) %}`

`filterSiblings`

Filter and paginate sibling Locations.

This enables filtering of the siblings by their ContentType with `$contentTypeIdentifiers` parameter and pagination using `$maxPerPage` and `$currentPage` parameters. The method returns a Pagerfanta instance.

Parameters	1. `array $contentTypeIdentifiers = []` 2. `int $maxPerPage = 25` 3. `int $currentPage = 1`
Returns	Pagerfanta instance with a slice of filtered sibling Locations
Sorting method	As is defined by the parent Location
Example in PHP	`$siblings = $location->filterSiblings(['articles'], 10, 2);`
Example in Twig	`{% set siblings = location.filterSiblings( ['articles'], 10, 2 ) %}`

Properties

Name	Type	Description
`$id`	`string|int`	ID of the Location
`$status`	`int`	Constant defining status (published or draft)
`$priority`	`int`	Priority
`$hidden`	`bool`	Own hidden state
`$invisible`	`bool`	Invisibility state (hidden by itself or by an ancestor)
`$remoteId`	`string`	Remote ID
`$parentLocationId`	`string|int`	Parent Location ID
`$pathString`	`string`	Path with materialized IDs (`/1/2/42/56/`)
`$path`	`int[]`	An array with materialized IDs (`[1, 2, 42, 56]`)
`$depth`	`int`	Depth in the Location tree
`$sortField`	`int`	Constant defining field for sorting children Locations
`$sortOrder`	`int`	Constant defining sort order for children Locations
`$contentId`	`string|int`	ID of the Content
`$contentInfo`	*ContentInfo*	ContentInfo object
`$parent`	*Location*	Parent Location object (lazy loaded)
`$content`	*Content*	Content object (lazy loaded)

2.1.6 Services

First thing to know about the Site API services is that all of them handle language configuration in a completely transparent way. You can be sure that all objects you work with:

1. can be rendered on the current siteaccess
2. are loaded in the single correct translation to be rendered on the current siteaccess

This works for both Content and Locations, whether they are obtained through search, loading by the ID, as relations or otherwise. If the object doesn't have a translation that can be rendered on a siteaccess, you won't be able to load it in the first place. That means you can put the whole language logic off your mind and solve real problems instead.

Following services are available:

- *LoadService*
  - *Methods*
    - *`loadContent()`*
    - *`loadContentByRemoteId()`*
    - *`loadLocation()`*
    - *`loadLocationByRemoteId()`*
- *FindService*
  - *Methods*
    - *`findContent()`*
    - *`findLocations()`*
- *FilterService*

LoadService

Instance of	`Netgen\EzPlatformSiteApi\API\LoadService`
Container service ID	`netgen.ezplatform_site.load_service`

The purpose of `LoadService` is to load Site Content and Locations by their ID.

Methods

`loadContent()`

Load Content object by it's ID.

Parameters	`string|int $id`
Returns	Content object
Example	`$content = $loadService->loadContent(42);`

`loadContentByRemoteId()`

Load Content object by it's remote ID.

Parameters	`string $remoteId`
Returns	Content object
Example	`$content = $loadService->loadContentByRemoteId('f2bfc25');`

`loadLocation()`

Load Location object by it's ID.

Parameters	`string|int $id`
Returns	Location object
Example	`$content = $loadService->loadLocation(42);`

`loadLocationByRemoteId()`

Load Location object by it's remote ID.

Parameters	`string $remoteId`
Returns	Location object
Example	`$content = $loadService->loadLocationByRemoteId('a44fd4e');`

FindService

Instance of	`Netgen\EzPlatformSiteApi\API\FindService`
Container service ID	`netgen.ezplatform_site.find_service`

The purpose of the `FindService` is to find Content and Locations by using eZ Platform's Repository Search API. This service will use the search engine that is configured for the Repository. That can be Legacy search engine or Solr search engine.

The service will return `SearchResult` object from the Repository API containing Site API objects.

Methods

findContent()

Find Content by the Content Query.

Parameters	`string|int $id`
Returns	Location object
Example	`$content = $findService->findContent(` `↪$query);`

findLocations()

Find Locations by the LocationQuery.

Parameters	`eZ\Publish\API\Repository\Values\Content\LocationQ` `$query`
Returns	`eZ\Publish\API\Repository\Values\Content\Search\Se`
Example	`$content = $findService->findLocations(` `↪$locationQuery);`

FilterService

Instance of	`Netgen\EzPlatformSiteApi\API\FilterService`
Container service ID	`netgen.ezplatform_site.load_service`

The purpose of the `FindService` is to find Content and Locations by using eZ Platform's Repository Search API. That is the same as `FindService`, but with the difference that it will always use Legacy search engine.

While Solr search engine provides more features and more performance than Legacy search engine, it's a separate system needs to be synchronized with changes in the database. This synchronization comes with a delay, which can be a problem in some cases.

FilterService gives you access to search that is always up to date, because it uses Legacy search engine that works directly with database. At the same time, search on top of Solr, with all the advanced features (like fulltext search or facets) is still available through FindService.

The service will return `SearchResult` object from the Repository API containing Site API objects.

Methods

filterContent()

Filter Content by the Content Query.

Parameters	`string|int $id`
Returns	Location object
Example	`$content = $filterService->filterContent(↪$query);`

filterLocations()

Filter Locations by the LocationQuery.

Parameters	`eZ\Publish\API\Repository\Values\Content\LocationQ $query`
Returns	`eZ\Publish\API\Repository\Values\Content\Search\Se`
Example	`$content = $filterService->↪filterLocations($locationQuery);`

RelationService

Instance of	`Netgen\EzPlatformSiteApi\API\RelationService`
Container service ID	`netgen.ezplatform_site.relation_service`

The purpose of `RelationService` is to provide a way to load field relations. This needs to be done respecting permissions and sort order and actually requires surprising amount of code when using Repository API.

Methods

loadFieldRelation()

Load single field relation from a specific field of a specific Content.

The method will return `null` if the field does not contain relations that can be loaded by the current user. If the field contains multiple relations, the first one will be returned. The method supports optional filtering by ContentType.

Parameters	1. `string|int $contentId` 2. `string $fieldDefinitionIdentifier` 3. `array $contentTypeIdentifiers = []`
Returns	`Content` or `null`
Example	`$content = $relationService->loadFieldRelation(42, 'relations', ['articles']);`

`loadFieldRelations()`

Load all field relations from a specific field of a specific Content. The method supports optional filtering by Content-Type.

Parameters	1. `string|int $contentId` 2. `string $fieldDefinitionIdentifier` 3. `array $contentTypeIdentifiers = []`
Returns	`Content` or `null`
Example	`$content = $relationService->loadFieldRelations(42, 'relations', ['articles']);`

Settings

The purpose of `Settings` object is to provide read access to current configuration.

Instance of	`Netgen\EzPlatformSiteApi\API\Settings`
Container service ID	`netgen.ezplatform_site.settings`

Properties

Property	Type	Description
`$prioritizedLanguages`	`string[]`	An array of prioritized languages of the current siteaccess
`$useAlwaysAvailable`	`bool`	Whether always available Content is taken into account when resolving translations
`$rootLocationId`	`string|int`	Root Location of the current siteaccess

Site

The purpose of `Site` service is to aggregate all other Site API services in one place. It implements a getter method for each of the services described above.

Instance of	`Netgen\EzPlatformSiteApi\API\Site`
Container service ID	`netgen.ezplatform_site.site`

Methods

Method	Returns
`getLoadService()`	*LoadService*
`getFindService()`	*FindService*
`getFilterService()`	*FilterService*
`getRelationService()`	*RelationService*
`getSettings()`	*Settings*

2.1.7 Custom controllers

Implementing a custom controller is similar to the vanilla eZ Platform. First, you have to implement it with extending the Site API base controller:

```
namespace AppBundle\Controller;

use Netgen\Bundle\EzPlatformSiteApiBundle\Controller\Controller;
use Netgen\Bundle\EzPlatformSiteApiBundle\View\ContentView;

class DemoController extends Controller
{
    /**
     * @param \Netgen\Bundle\EzPlatformSiteApiBundle\View\ContentView $view
     *
     * @return \Netgen\Bundle\EzPlatformSiteApiBundle\View\ContentView
     */
    public function viewArticleAction(ContentView $view)
```

(continues on next page)

(continued from previous page)

```
    {
        $content = $view->getSiteContent();
        $location = $view->getSiteLocation();

        $filterService = $this-getSite()->getFilterService();

        $hasRelatedItems = false;

        if (!$content->getField('related')->isEmpty()) {
            $hasRelatedItems = true;
        }

        // Your other custom logic here
        // ...

        // Add variables to the view
        $view->addParameters([
            'has_related_items' => $hasRelatedItems,
        ]);

        return $view;
    }
}
```

Next, register your controller with the DI container. The base controller expects that two setter methods are called on instantiation: `setContainer()` and `setSite()`. You can do this manually:

```
app.controller.demo:
    class: AppBundle\Controller\DemoController
    calls:
        - [setContainer, ['@service_container']]
        - [setSite, ['@netgen.ezplatform_site.core.site']]
```

Or by extending the base definition:

```
app.controller.demo:
    parent: netgen.ezplatform_site.controller.base
    class: AppBundle\Controller\DemoController
```

Now you can use your custom controller in the view configuration:

```
ezpublish:
    system:
        frontend_group:
            ngcontent_view:
                full:
                    article:
                        template: "@App/content/full/article.html.twig"
                        controller: "app.controller.demo:viewArticleAction"
                        match:
                            Identifier\ContentType: article
```

2.1.8 Migration

If you are starting with a new project on top of vanilla eZ Platform, then you're starting with a clean slate and of course there is no need to migrate anything. In that case it's enough to *install* and *configure* the Site API and you can start

working with it.

If that's the case, we recommend that you look into our Media Site, which is built with Site API and will provide you with a comprehensive base for building a web project on eZ Platform.

On the other hand if you want to add the Site API to an existing project or you have a base site of your own, read on to find out about your options.

Choosing your migration strategy

You can *install* the Site API on a existing project without worrying that something will break – everything should just keep working as before. However, nothing will use the Site API – you will first have to develop new features or migrate existing ones.

At this point, you can:

1. use Site API services as you would normally do in a Symfony application. For example you could use it in a custom route.
2. use Site API's view *configuration*, available under `ngcontent_view` key. You need to know that eZ Platform URL alias routes still won't be handled through it at this point. Until you explicitly turn that on for a siteaccess, you can only use it by making a subrequest to Site API's Content view controller `ng_content:viewAction`.

Handling eZ Platform URL alias routes through Site API's view configuration has to be enabled per siteaccess, with the following configuration:

```
netgen_ez_platform_site_api:
    system:
        frontend_group:
            override_url_alias_view_action: true
```

Once you do this, all URL alias routes on the siteaccess will be handled through Site API's view configuration. That means you will need to migrate or adapt all full view templates, otherwise expect that things will break. Similar to the point 2. from above will be valid for eZ Platform's view configuration, available under `content_view` key. You will still be able to use it, but only through explicit subrequests to eZ Platform's view controller `ez_content:viewAction`.

All Site API objects contain their eZ Platform counterparts. This will enable initial mixing of both Site API and vanilla eZ Platform ways of doing things, which means you will be able to migrate your project one step at a time.

Knowing all that gives you quite some flexibility in choosing exactly how you want to adapt your project to use Site API.

Comparison with eZ Platform

Here's a comparison table of Site API and eZ Platform Twig functions to provide a quick overview of changes needed in the templates.

eZ Platform	Netgen's Site API
`{{ ez_content_name( content ) }}`	`{{ content.name }}`
`{{ ez_field_name( content, 'title' ) }}`	`{{ content.fields.title.name }}`
`{{ ez_field_description( content, 'title' ) }}`	`{{ content.fields.title.description }}`
`{{ ez_field( content, 'title' ) }}`	`{{ content.fields.title }}`
`{{ ez_render_field( content, 'title' ) }}`	`{{ ng_render_field( content.fields.title ) }}`
`{{ ez_field_value( content, 'title' ) }}`	`{{ content.fields.title.value }}`
`{{ ez_is_field_empty( content, 'title' ) }}`	`{{ content.fields.title.empty }}`
`{{ ez_image_alias( content.field( 'image' ), content.versionInfo, 'large' ) }}`	`{{ ng_image_alias( content.fields.image, 'large' ) }}`

Search and replace regexes

Here are some regular expressions that you can use to migrate your Twig templates. The list is not complete, but it should get you started. If you're using PHP Storm, follow the steps:

1. Open your PHPStorm
2. Navigate to template
3. Press CTRL + R or Command + R
4. Enter the one of the search/replace pairs from below and replace away

ez_is_field_empty

search for	`ez_is_field_empty\s*\(\s*([a-zA-Z0-9\_]+)\s*,\s*['"]([a-zA-Z0-9\_]+)['"]\s*\)`
replace with	`$1.fields.$2.empty`

ez_field_value

search for	`ez_field_value\s*\(\s*([a-zA-Z0-9\_]+)\s*,\s*['"]([a-zA-Z0-9\_]+)['"]\s*\)`
replace with	`$1.fields.$2.value`

ez_render_field

search for	`ez_render_field[ ]?\(\s+([a-zA-Z0-9\_]+),\s+['"]([a-zA-Z0-9\_]+)['"](.*?)?\)`
replace with	`ng_render_field( $1.fields.$2$3 )`

- *Installation*
- *Configuration*
- *Templating*
- *Query Types*
- *Objects*
- *Services*
- *Custom controllers*
- *Migration*
- *Installation*
- *Configuration*
- *Templating*
- *Query Types*
- *Objects*
- *Services*
- *Custom controllers*
- *Migration*

CHAPTER 3

Upgrades

3.1 Upgrades

3.1.1 Upgrading from 2.3.0 to 2.4.0

Controllers that extend from `Netgen\Bundle\EzPlatformSiteApiBundle\Controller\Controller` and are registered inside dependency injection container should set two setter injection `calls`:

```
app.demo.controller.demo_controller:
    class: Acme\Bundle\DemoBundle\Controller\DemoController
    calls:
        - [setContainer, ['@service_container']]
        - [setSite, ['@netgen.ezplatform_site.site']]
```

Or if you want to avoid setter calls, just set `parent` service:

```
app.demo.controller.demo_controller:
    parent: netgen.ezplatform_site.controller.base
    class: Acme\Bundle\DemoBundle\Controller\DemoController
```

3.1.2 Upgrading from 1.0.0 to 2.0.0

eZ Platform Site API introduces a slight breaking change to `ContentView` value object, hence the bump to version 2.0.

- Site API `ContentView` view object does not extend from eZ Platform `ContentView` value object any more to allow implementation of custom view providers. Class signature did not change, however, since all required interfaces are now implemented directly on Site API `ContentView` value object.
- Also, `Netgen\Bundle\EzPlatformSiteApiBundle\View\ContentValueView` interface does not contain `getSiteLocation` method any more. It is moved to a new interface, `LocationValueView`, in the same namespace. If you used this method in your code, make sure to check for this new interface. This was done to keep in line on how eZ kernel uses its `ContentView` value object and its interfaces.