Content Rendering

Introduction

The ViewController

eZ Platform comes with a native controller to display your content, known as the **`ViewController`** . It is called each time you try to reach a Content item from its **Url Alias** (human readable, translatable URI generated for any content based on URL patterns defined per Content Type) and is able to render any content previously edited in the admin interface or via the Public API Guide.

It can also be called straight by its direct URI:

```
/view/content/<contentId>/full/true/<locationId>
```

```
/view/content/<contentId>
```

A Content item can also have different **view types** (full page, abstract in a list, block in a landing page, etc.). By default the view type is **full** (for full page), but it can be anything (*line*, *block, etc.*).

> **Important note regarding visibility**
> The Location visibility flag, which you can change by hiding/revealing in the Platform UI, is not permission-based and thus acts as a simple potential filter. **It is not meant to restrict access to content**.
>
> If you need to restrict access to a given Content item, use **Sections** or **Object states**, which are permission-based.

Configuration

View provider configuration

The **configured ViewProvider** allows you to configure template selection when using the `ViewController`, either directly from a URL or via a sub-request.

> **eZ Publish 4.x terminology**
> In eZ Publish 4.x, it was known as **template override system *by configuration*** (`override.ini`).
> However this only reflects old overrides for `node/view/*.tpl` and `content/view/*.tpl`.

Principle

The **configured ViewProvider** takes its configuration from your siteaccess in the **`content_view`** section. This configuration is a hash built in the following way:

In this topic:

Related topics:

Injecting parameters in content views

app/config/ezplatform.yml

```
ezpublish:
    system:
        # Can be a valid siteaccess, siteaccess group or
even "global"
        front_siteaccess:
            # Configuring the LocationViewProvider
            content_view:
                # The view type (full/line are standard,
but you can use custom ones)
                full:
                    # A simple unique key for your
matching ruleset
                    folderRuleset:
                        # The template identifier to
load, following the Symfony bundle notation for
templates
                        # See
http://symfony.com/doc/current/book/controller.html#rend
ering-templates
                        template:
eZDemoBundle:full:small_folder.html.twig
                        # Hash of matchers to use, with
their corresponding values to match against
                        match:
                            # Key is the matcher
"identifier" (class name or service identifier)
                            # Value will be passed to
the matcher's setMatchingConfig() method.
                            Identifier\ContentType:
[small_folder, folder]
```

Important note about template matching
Template matching will NOT work if your content contains a Field Type that is not supported by the repository. It can be the case when you are in the process of a migration from eZ Publish 4.x, where custom datatypes have been developed.

In this case the repository will throw an exception, which is caught in the `ViewController`, and *if* you are using LegacyBridge it will end up doing a **fallback to legacy kernel**.

The list of Field Types supported out of the box is available here.

Tip
You can define your template selection rules in a different configuration file. Read the cookbook recipe to learn more about it .

You can also use your own custom controller to render a content/location.

About `location_view` & `content_view`

Until eZ Publish Platform 5.4, the main view action was `location_view`. This is deprecated since eZ Platform 15.12 (1.0). Only `content_view` should be used to view content, with a location as an option.

Existing `location_view` rules will be, *when possible*, converted transparently to `content_view`, with a deprecation notice. However, it is not possible to do so when the rule uses a custom

controller.
In any case, `location_view` rules should be converted to `content_view` ones, as `location_view` will be removed in the next kernel major version.

Matchers

To be able to select the right templates against conditions, the view provider uses matcher objects, all implementing `eZ\Publish\Core\MVC\Symfony\View\ContentViewProvider\Configured\Matcher` interface.

Matcher identifier

The matcher identifier can comply to 3 different formats:

1. **Relative qualified class name** (e.g. `Identifier\ContentType`). This is the most common case and used for native matchers. It will then be relative to `eZ\Publish\Core\MVC\Symfony\Matcher\ContentBased`.
2. **Full qualified class name** (e.g. `\Foo\Bar\MyMatcher`). This is a way to specify a **custom matcher** that doesn't need specific dependency injection. Please note that it **must** start with a `\`.
3. **Service identifier**, as defined in Symfony service container. This is the way to specify a more **complex custom matcher** that has dependencies.

> **Injecting the Repository**
> If your matcher needs the repository, simply make it implement `eZ\Publish\Core\MVC\RepositoryAwareInterface` or extend the `eZ\Publish\Core\MVC\RepositoryAware` abstract class. The repository will then be correctly injected before matching.

Matcher value

The value associated to the matcher is being passed to its `setMatchingConfig()` method and can be anything supported by the matcher.

> In the case of native matchers, they support both **scalar values** or **arrays of scalar values**.
> Passing an array amounts to applying a logical OR.

Combining matchers

It is possible to combine matchers to add additional constraints for matching a template:

```
# ...
match:
    Identifier\ContentType: [small_folder, folder]
    Identifier\ParentContentType: frontpage
```

The example above can be translated as "Match any content which **ContentType** identifier is ***small_folder*** **OR** ***folder*** , **AND** having *frontpage* as **ParentContentType** identifier".

Available matchers

The following table presents all native matchers.

Identifier	Description
`Id\Content`	Matches the ID number of the Content item

`Id\ContentType`	Matches the ID number of the Content Type that the Content item is an instance of
`Id\ContentTypeGroup`	Matches the ID number of the group containing the Content Type that the Content item is an instance of
`Id\Location`	Matches the ID number of a Location. *In the case of a Content item, matched against the main location.*
`Id\ParentContentType`	Matches the ID number of the parent Content Type. *In the case of a Content item, matched against the main location.*
`Id\ParentLocation`	Matches the ID number of the parent Location. *In the case of a Content item, matched against the main location.*
`Id\Remote`	Matches the remoteId of either content or Location, depending on the object matched.
`Id\Section`	Matches the ID number of the Section that the Content item belongs to.
`Id\State`	*Not supported yet.*
`Identifier\ContentType`	Matches the identifier of the Content Type that the Content item is an instance of.
`Identifier\ParentContentType`	Matches the identifier of the parent Content Type. *In the case of a Content item, matched against the main Location.*
`Identifier\Section`	Matches the identifier of the Section that the Content item belongs to.
`Identifier\State`	*Not supported yet.*
`Depth`	Matches the depth of the Location. The depth of a top level Location is 1.
`UrlAlias`	Matches the virtual URL of the Location (i.e. `/My/Content-Uri`). **Important: Matches when the UrlAlias of the location starts with the value passed.** *Not supported for Content (aka content_view).*

Default view templates

Content view uses default templates to render content unless custom view rules are used.

Those templates can be customized by means of container- and siteaccess-aware parameters.

Overriding the default template for common view types

Templates for the most common view types (content/full, line, embed, or block) can be customized by setting one the `ezplatform.default.content_view_templates` variables:

Controller	ViewType	Parameter	Default value
`ez_content:viewAction`	`full`	`ezplatform.default_view_templates.content.full`	`"EzPublishCoreBundle:default:content/full.html.twig"`

`ez_content:viewAction`	`line`	`ezplatform.default_view_templates.content.line`	`"EzPublishCoreBundle:default:content/line.html.twig"`
`ez_content:viewAction`	`embed`	`ezplatform.default_view_templates.content.embed`	`"EzPublishCoreBundle:default:content/embed.html.twig"`
`ez_page:viewAction`	`n/a`	`ezplatform.default_view_templates.block`	`"EzPublishCoreBundle:default:block/block.html.twig"`

Example

Add this configuration to `app/config/config.yml` to use `app/Resources/content/view/full.html.twig` as the default template when viewing Content with the `full` view type:

```
parameters:
    ezplatform.default_view_templates.content.full:
"content/view/full.html.twig"
```

Customizing the default controller

The controller used to render content by default can also be changed. The `ezsettings.default.content_view_defaults` container parameter contains a hash that defines how content is rendered by default. It contains a set of classic content view rules for the common view types. This hash can be redefined to whatever suits your requirements, including custom controllers, or even matchers.

Usage

View selection

To display a Content item, the ViewController uses a view manager which selects the appropriate template depending on matching rules.

> For more information about the **view provider configuration**, please refer to the dedicated section .

> You can also use your own custom controller to render a content item/location.

Content view templates

A content view template is like any other template, with several specific aspects.

Available variables

Variable name	Type	Description
`location`	eZ\Publish\Core\Repository\Values\Content\Location	The Location object. Contains meta information on the content (ContentInfo) (only when accessing a Location)

`content`	eZ\Publish\Core\Repository\Values\Content\Content	The Content item, containing all Fields and version information (VersionInfo)
`noLayout`	Boolean	If true, indicates if the Content item/Location is to be displayed without any pagelayout (i.e. AJAX, sub-requests, etc.). It's generally `false` when displaying a Content item in view type **full**.
`viewBaseLayout`	String	The base layout template to use when the view is requested to be generated outside of the pagelayout (when `noLayout` is true).

Template inheritance and sub-requests

Like any template, a content view template can use template inheritance. However keep in mind that your content may be also requested via sub-requests (see below how to render embedded content objects), in which case you probably don't want the global layout to be used.

If you use different templates for embedded content views, this should not be a problem. If you'd rather use the same template, you can use an extra `noLayout` view parameter for the sub-request, and conditionally extend an empty pagelayout:

```
{% extends noLayout ? viewbaseLayout :
"AcmeDemoBundle::pagelayout.html.twig" %}

{% block content %}
...
{% endblock %}
```

Rendering Content item's Fields

As stated above, a view template receives the requested Content item, holding all Fields.

In order to display the Fields' value the way you want, you can either manipulate the Field Value object itself, or use a custom template.

Getting raw Field value

Having access to the Content item in the template, you can use its public methods to access all the information you need. You can also use the **ez_field_value** helper to get the field types reference . It will return the correct language if there are several, based on language priorities.

```
{# With the following, myFieldValue will be in the
content's main language, regardless of the current
language #}
{% set myFieldValue = content.getFieldValue(
'some_field_identifier' ) %}

{# Here myTranslatedFieldValue will be in the current
language if a translation is available. If not, the
content's main language will be used #}
{% set myTranslatedFieldValue = ez_field_value( content,
'some_field_identifier' ) %}
```

Using the Field Type's template block

All built-in Field Types come with their own Twig template. You can render any Field using this default template using the `ez_render_field()` helper.

```
{{ ez_render_field( content, 'some_field_identifier' )
}}
```

Refer to ez_render_field for further information.

> As this makes use of reusable templates, **using `ez_render_field()` is the recommended way and is to be considered the best practice**.

Rendering Content name

The **name** of a Content item is its generic "title", generated by the repository based on the Content Type's naming pattern. It often takes the form of a normalized value of the first field, but might be a concatenation of several fields. There are 2 different ways to access this special property:

- Through the name property of ContentInfo (not translated).
- Through VersionInfo with the TranslationHelper (translated).

Translated name

The *translated name* is held in a `VersionInfo` object, in the names property which consists of hash indexed by locale. You can easily retrieve it in the right language via the `TranslationHelper` service.

```
<h2>Translated content name: {{ ez_content_name( content
) }}</h2>
<h3>Also works from ContentInfo : {{ ez_content_name(
content.contentInfo ) }}</h3>
```

The helper will by default follow the prioritized languages order. If there is no translation for your prioritized languages, the helper will always return the name in the main language.

You can also **force a locale** in a second argument:

```
{# Force fre-FR locale. #}
<h2>{{ ez_content_name( content, 'fre-FR' ) }}</h2>
```

You can find more information further in this document.

Name property in ContentInfo

This property is the actual content name, but **in main language only** (so it is not translated).

```
<h2>Content name: {{ content.contentInfo.name }}</h2>
```

```
$contentName = $content->contentInfo->name;
```

Exposing additional variables

It is possible to expose additional variables in a content view template. See parameters injection in content views.

Custom controllers

In some cases, displaying a Content item/Location via the built-in `ViewController` is not sufficient to show everything you want. In such cases you may want to **use your own custom controller** to display the current Content item/Location instead.

Typical use cases include access to:

- Settings (coming from `ConfigResolver` or `ServiceContainer`)
- Current Content item's `ContentType` object
- Current Location's parent
- Current Location's children count
- Main Location and alternative Locations for the current Content item
- etc.

There are three ways in which you can apply a custom controller:

- Configure a custom controller alongside regular matcher rules to use **both** your custom controller and the `ViewController` (recommended).
- **Override** the built-in `ViewController` with the custom controller in a specific situation.
- **Replace** the `ViewController` with the custom controller for the whole bundle.

Enriching ViewController with a custom controller

This is the recommended way of using a custom controller

To use your custom controller on top of the built-in `ViewController` you need to point to both the controller and the template in the configuration, for example:

ezplatform.yml

```
ezpublish:
    system:
        default:
            content_view:
                full:
                    article:
                        controller:
AcmeTestBundle:Default:articleViewEnhanced
                        template:
AcmeTestBundle:full:article.html.twig
                        match:
                            Identifier\ContentType:
[article]
```

With this configuration, the following controller will forward the request to the built-in `ViewController` with some additional parameters:

Controller

```
<?php

namespace Acme\TestBundle\Controller;

use Symfony\Component\HttpFoundation\Response;
use eZ\Bundle\EzPublishCoreBundle\Controller;

class DefaultController extends Controller
{
    public function articleViewEnhancedAction(
$locationId, $viewType, $layout = false, array $params =
array() )
    {
        // Add custom parameters to existing ones.
        $params += array( 'myCustomVariable' => "Hey,
I'm a custom message!" );
        // Forward the request to the original
ViewController
        // And get the response. Possibly alter it (here
we change the smax-age for cache).
        $response = $this->get( 'ez_content'
)->viewLocation( $locationId, $viewType, $layout,
$params );
        $response->setSharedMaxAge( 600 );

        return $response;
    }
}
```

Always ensure that you add new parameters to existing `$params` associative array using **+** union operator or `array_merge()`. **Not doing so (e.g. only passing your custom parameters array) can result in unexpected issues with content preview**. Previewed content and other parameters are indeed passed in `$params`.

These parameters can then be used in templates, for example:

article.html.twig

```
{% extends noLayout ? viewbaseLayout :
"eZDemoBundle::pagelayout.html.twig" %}

{% block content %}
    <h1>{{ ez_render_field( content, 'title' ) }}</h1>
    <h2>{{ myCustomVariable }}</h2>
    {{ ez_render_field( content, 'body' ) }}
{% endblock %}
```

Using only your custom controller

If you want to apply only your custom controller in a given match situation and not make use of the `ViewController` at all, in the configuration you need to indicate the controller, but no template, for example:

ezplatform.yml

```
ezpublish:
    system:
        default:
            content_view:
                full:
                    folder:
                        controller:
AcmeTestBundle:Default:viewFolder
                        match:
                            Identifier\ContentType:
[folder]
                            Identifier\Section:
[standard]
```

In this example, as the `ViewController` is not applied, the custom controller takes care of the whole process of displaying content, including pointing to the template to be used (in this case, `AcmeTestBundle::custom_controller_folder.html.twig`:

Controller

```
<?php

namespace Acme\TestBundle\Controller;

use Symfony\Component\HttpFoundation\Response;
use eZ\Bundle\EzPublishCoreBundle\Controller;

class DefaultController extends Controller
{
    public function viewFolderAction( $locationId,
$layout = false, $params = array() )
    {
        $repository = $this->getRepository();
        $location =
$repository->getLocationService()->loadLocation(
$locationId );
        // Check if content is not already passed. Can
be the case when using content preview.
        $content = isset( $params['content'] ) ?
$params['content'] :
$repository->getContentService()->loadContentByContentIn
fo( $location->getContentInfo() )
        $response = new Response();
        $response->headers->set( 'X-Location-Id',
$locationId );
        // Caching for 1h and make the cache vary on
user hash
        $response->setSharedMaxAge( 3600 );
        $response->setVary( 'X-User-Hash' );
        return $this->render(

'AcmeTestBundle::custom_controller_folder.html.twig',
            array(
                'location' => $location,
                'content' => $content,
                'foo' => 'Hey world!!!',
                'osTypes' => array( 'osx', 'linux',
'windows' )
            ) + $params
        );
    }
}
```

Here again custom parameters can be used in the template, e.g.:

custom_controller_folder.html.twig

```
{% extends "eZDemoBundle::pagelayout.html.twig" %}

{% block content %}
<h1>{{ ez_render_field( content, 'title' ) }}</h1>
    <h1>{{ foo }}</h1>
    <ul>
    {% for os in osTypes %}
        <li>{{ os }}</li>
    {% endfor %}
    </ul>
{% endblock %}
```

Overriding the built-in ViewController

One other way to keep control of what is passed to the view is to use your own controller instead of the built-in `ViewController`. As base `ViewController` is defined as a service, with a service alias, this can be easily achieved from your bundle's configuration:

```
parameters:
    my.custom.view_controller.class:
Acme\TestBundle\MyViewController

services:
    my.custom.view_controller:
        class: %my.custom.view_controller.class%
        arguments: [@some_dependency, @other_dependency]

    # Change the alias here and make it point to your
own controller
    ez_content:
        alias: my.custom.view_controller
```

> Doing so will completely override the built-in `ViewController`! Use this at your own risk!

Custom controller structure

Your custom controller can be any kind of controller supported by Symfony (including controllers as a service).

The only requirement here is that your action method must have a similar signature to `ViewController::viewLocation()` or `ViewController::viewContent()` (depending on what you're matching of course). However, note that not all arguments are mandatory, since Symfony is clever enough to know what to inject into your action method. That is why **you aren't forced to mimic the `ViewController`'s signature strictly**. For example, if you omit `$layout` and `$params` arguments, it will still be valid. Symfony will just avoid injecting them into your action method.

Built-in ViewController signatures

viewLocation() signature

```
/**
 * Main action for viewing content through a location in
the repository.
 *
 * @param int $locationId
 * @param string $viewType
 * @param boolean $layout
 * @param array $params
 *
 * @throws
\Symfony\Component\Security\Core\Exception\AccessDeniedE
xception
 * @throws \Exception
 *
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function viewLocation( $locationId, $viewType,
$layout = false, array $params = array() )
```

viewContent() signature

```
/**
 * Main action for viewing content.
 *
 * @param int $contentId
 * @param string $viewType
 * @param boolean $layout
 * @param array $params
 *
 * @throws
\Symfony\Component\Security\Core\Exception\AccessDeniedE
xception
 * @throws \Exception
 *
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function viewContent( $contentId, $viewType,
$layout = false, array $params = array() )
```

Controller selection doesn't apply to `block_view` since you can already use your own controller to display blocks.

Caching
When you use your own controller, **it is your responsibility to define cache rules**, like with every custom controller!

So don't forget to **set cache rules** and the appropriate **`X-Location-Id` header** in the returned `Response` object.

See built-in ViewController for more details on this.

Embedded images

V1.4

The Rich Text Field allows you to embed other Content items within the Field.

Content items that are identified as images will be rendered in the Rich Text Field using a dedicated template.

You can determine which Content Types will be treated as images and rendered using this template in the `ezplatform.content_view.image_embed_content_types_identifiers` parameter. By default it is set to cover the Image Content Type, but you can add other types that you want to be treated as images, for example:

```
parameters:

ezplatform.content_view.image_embed_content_types_identi
fiers: ['image', 'photo', 'banner']
```

The template used when rendering embedded images can be set in the `ezplatform.default_view_templates.content.embed_image` container parameter:

```
parameters:

ezplatform.default_view_templates.content.embed_image:
'content/view/embed/image.html.twig
```

Query controller

V1.4

The Query controller is a predefined custom content view controller that runs a Repository Query.

It is meant to be used as a custom controller in a view configuration, along with match rules. It can use properties of the viewed content or location as parameters to the Query. It makes it easy to retrieve content without writing custom PHP code and display the results in a template.

Use-case examples

- List of Blog posts in a Blog
- List of Images in a Gallery

Usage example

We will use the blog posts use case mentioned above as an example. It assumes a "Blog" container that contains a set of "Blog post" items. The goal is, when viewing a Blog, to list the Blog posts it contains.

Three items are required:

- a `LocationChildren` QueryType
  Will generate a Query retrieving the children of a given location id
- a View template
  Will render the Blog, and list the Blog posts it contains
- a `content_view` configuration
  Will instruct Platform, when viewing a Content item of type Blog, to use the Query Controller, the view template, and the LocationChildren QueryType. It will also map the id of the viewed Blog to the QueryType parameters, and set which twig variable the results will be assigned to.

The LocationChildren QueryType

QueryTypes are described in more detail in the next section. In short, a QueryType can build a Query object, optionally based on a set of parameters. The following example will build a Query that retrieves the sub-items of a Location:

src/AppBundle/QueryType/LocationChildrenQueryType.php

```
<?php
namespace AppBundle\QueryType;

use
eZ\Publish\API\Repository\Values\Content\LocationQuery;
use
eZ\Publish\API\Repository\Values\Content\Query\Criterion
\ParentLocationId;
use eZ\Publish\Core\QueryType\QueryType;

class LocationChildrenQueryType implements QueryType
{
    public function getQuery(array $parameters = [])
    {
        return new LocationQuery([
            'filter' => new
ParentLocationId($parameters['parentLocationId']),
        ]);
    }

    public function getSupportedParameters()
    {
        return ['parentLocationId'];
    }

    public static function getName()
    {
        return 'LocationChildren';
    }
}
```

Any class will be registered as a QueryType when it:

- implements the QueryType interface,
- is located in the QueryType subfolder of a bundle, and in a file named "SomethingQueryType.php"

If the QueryType has dependencies, it can be manually tagged as a service using the `ezpublish.query_type` service tag, but it is not required in that case.

The content_view configuration

We now need a view configuration that matches content items of type "Blog", and uses the QueryController to fetch the blog posts:

app/config/ezplatform.yml

```
ezpublish:
      system:
            site_group:
                content_view:
               full:
      blog:
                      controller:
"ez_query:locationQueryAction"
       template: "content/view/full/blog.html.twig"
       match:
        Identifier\ContentType: "blog"
                    params:
                              query:
                                  query_type:
'LocationChildren'
                             parameters:
                                 parentLocationId:
"@=location.id"
                                  assign_results_to:
'blog_posts'
```

The view's controller action is set to the QueryController's locationQuery action (`ez_query:locationQueryAction`). Other actions are available that run a different type of search (contentInfo or content).

The QueryController is configured in the `query` array, inside the `params` of the content_view block:

- `query_type` specifies the QueryType to use, based on its name.
- `parameters` is a hash where parameters from the QueryType are set. Arbitrary values can be used, as well as properties from the currently viewed location and content. In that case, the id of the currently viewed location is mapped to the QueryType's `parentLocationId` parameter: `parentLocationId: "@=location.id"`
- `assign_results_to` sets which twig variable the search results will be assigned to.

The view template

Results from the search are assigned to the `blog_posts` variable as a `SearchResult` object. In addition, since the usual view controller is used, the currently viewed `location` and `content` are also available.

app/Resources/views/content/full/blog.html.twig

```
<h1>{{ ez_content_name(content) }}</h1>

{% for blog_post in blog_posts.searchHits %}
  <h2>{{
ez_content_name(blog_post.valueObject.contentInfo)
}}</h2>
{% endfor %}
```

Configuration details

`controller`

Three Controller Actions are available, each for a different type of search:

- `locationQueryAction` runs a Location Search
- `contentQueryAction` runs a Content Search
- `contentInfoQueryAction` runs a Content Info search

> See Search documentation page for more details about different types of search

`params`

The Query is configured in a `query` hash in `params`, you could specify the QueryType name, additional parameters and the Twig variable that you will assign the results to for use in the template.

- **`query_type`**
  - Name of the Query Type that will be used to run the query, defined by the class name.
- **`parameters`**
  - Query Type parameters that can be provided in two ways:
  1. As scalar values, for example an identifier, an id, etc.
  2. Using the Expression language. This simple script language, similar to Twig syntax, lets you write expressions that get value from the current content and/or location:
     - For example, `@=location.id` will be evaluated to the currently viewed location's ID.
       `content`, `location` and `view` are available as variables in expressions.
- **`assign_results_to`**
  - This is the name of the Twig variable that will be assigned the results.
  - Note that the results are the SearchResult object returned by the SearchService.

Query Types objects

QueryTypes are objects that build a Query. They are different from Public API queries.

To make a new QueryType available to the Query Controller, you need to create a PHP class that implements the QueryType interface, then register it as such in the Service Container.

> For more information about the Service Container, read the page

The QueryType interface

There you can view the PHP QueryType interface. Three methods are described:

1. getQuery()
2. getSupportedParameters()
3. getName()

Expand source

```
interface QueryType
{
 /**
  * Builds and returns the Query object
  *
  * The Query can be either a Content or a Location one.
  *
  * @param array $parameters A hash of parameters that
will be used to build the Query
  * @return
\eZ\Publish\API\Repository\Values\Content\Query
  */
 public function getQuery(array $parameters = []);

 /**
  * Returns an array listing the parameters supported by
the QueryType
  * @return array
  */
 public function getSupportedParameters();

 /**
  * Returns the QueryType name
  * @return string
  */
 public static function getName();
}
```

Parameters

A QueryType may accept parameters, including string, arrays and other types, depending on the implementation. They can be used in any way, such as:

- customizing an element's value (limit, ContentType identifier, etc)
- conditionally adding/removing criteria from the query
- setting the limit/offset

The implementations should use Symfony's `OptionsResolver` for parameters handling and resolution.

QueryType example: latest content

Let's see an example for a QueryType creation.

This QueryType returns a Query that searches for **the 10 last published Content items, order by reverse publishing date**.
It accepts an optional `type` parameter, that can be set to a ContentType identifier:

```php
<?php
namespace AppBundle\QueryType;
use eZ\Publish\Core\QueryType\QueryType;
use eZ\Publish\API\Repository\Values\Content\Query;
class LatestContentQueryType implements QueryType
{
    public function getQuery(array $parameters = [])
    {
        $criteria[] = new
Query\Criterion\Visibility(Query\Criterion\Visibility::V
ISIBLE);
        if (isset($parameters['type'])) {
            $criteria[] = new
Query\Criterion\ContentTypeIdentifier($parameters['type'
]);
        }
        // 10 is the default limit we set, but you can
have one defined in the parameters
        return new Query([
            'filter' => new
Query\Criterion\LogicalAnd($criteria),
            'sortClauses' => [new
Query\SortClause\DatePublished()],
            'limit' => isset($parameters['limit']) ?
$parameters['limit'] : 10,
        ]);
    }
    public static function getName()
    {
        return 'AppBundle:LatestContent';
    }
    /**
     * Returns an array listing the parameters supported
by the QueryType.
     * @return array
     */
    public function getSupportedParameters()
    {
        return ['type', 'limit'];
    }
}
```

Naming of QueryTypes

Each QueryType is named after what is returned by `getName()`. **Names must be unique.** A warning will be thrown during compilation if there is a conflict, and the resulting behavior will be unpredictable.

QueryType names should use a unique namespace, in order to avoid conflicts with other bundles. We recommend that the name is prefixed with the bundle's name: `AcmeBundle:LatestContent`. A vendor/company's name could also work for QueryTypes that are reusable throughout projects: `Acme:LatestContent`.

Registering the QueryType into the service container

In addition to creating a class for a `QueryType`, you must also register the QueryType with the Service Container. This can be done in two ways: by convention, and with a service tag.

By convention

Any class named `<Bundle>\QueryType\*QueryType`, that implements the QueryType interface, will be registered as a custom QueryType.
Example: `AppBundle\QueryType\LatestContentQueryType`.

Using a service tag

If the proposed convention doesn't work for you, QueryTypes can be manually tagged in the service declaration:

```
acme.query.latest_content:
    class: AppBundle\Query\LatestContent
    tags:
        - {name: ezpublish.query_type}
```

The effect is exactly the same than registering by convention.

> **More content...**
> Follow the FieldType creation Tutorial and learn how to Register the Field Type as a service

The OptionsResolverBasedQueryType abstract class

An abstract class based on Symfony's `OptionsResolver` eases implementation of QueryTypes with parameters.

It provides final implementations of `getQuery()` and `getDefinedParameters()`.

A `doGetQuery()` method must be implemented instead of `getQuery()`. It is called with the parameters processed by the OptionsResolver, meaning that the values have been validated, and default values have been set.

In addition, the `configureOptions(OptionsResolver $resolver)` method must configure the OptionsResolver.

The LatestContentQueryType can benefit from the abstract implementation:

- validate that `type` is a string, but make it optional
- validate that `limit` is an int, with a default value of 10

> For further information see the Symfony's Options Resolver documentation page

```
<?php
namespace AppBundle\QueryType;
use eZ\Publish\API\Repository\Values\Content\Query;
use Symfony\Component\OptionsResolver\OptionsResolver;
class OptionsBasedLatestContentQueryType extends
OptionsResolverBasedQueryType implements QueryType
{
    protected function doGetQuery(array $parameters)
    {
        $criteria[] = new
Query\Criterion\Visibility(Query\Criterion\Visibility::V
ISIBLE);
        if (isset($parameters['type'])) {
            $criteria[] = new
Query\Criterion\ContentTypeIdentifier($parameters['type'
]);
        }
        return new Query([
            'criterion' => new
Query\Criterion\LogicalAnd($criteria),
            'sortClauses' => [new
Query\SortClause\DatePublished()],
            'limit' => $parameters,
        ]);
    }
    public static function getName()
    {
        return 'AppBundle:LatestContent';
    }
    protected function configureOptions(OptionsResolver
$resolver)
    {
        $resolver->setAllowedTypes('type', 'string');
        $resolver->setAllowedValues('limit', 'int');
        $resolver->setDefault('limit', 10);
    }
}
```

Using QueryTypes from PHP code

All QueryTypes are registered in a registry, the QueryType registry.

It is available from the container as `ezpublish.query_type.registry`

```
<?php
class MyCommand extends ContainerAwareCommand
{
    protected function execute(InputInterface $input,
OutputInterface $output)
    {
        $queryType     =
$this->getContainer()->get('ezpublish.query_type.registr
y')->getQueryType('AcmeBundle:LatestContent');
        $query         = $queryType->getQuery(['type' =>
'article']);
        $searchResults =
$this->getContainer()->get('ezpublish.api.service.search
')->findContent($query);
        foreach ($searchResults->searchHits as
$searchHit) {

$output->writeln($searchHit->valueObject->contentInfo->n
ame);
        }
    }
}
```

Making links to other locations

Linking to other locations is fairly easy and is done with a native `path()` Twig helper (or `url()` if you want to generate absolute URLs). You just have to pass it the Location object and `path()` will generate the URLAlias for you.

```
{# Assuming "location" variable is a valid
eZ\Publish\API\Repository\Values\Content\Location object
#}
<a href="{{ path( location ) }}">Some link to a
location</a>
```

If you don't have the Location object, but only its ID, you can generate the URLAlias the following way:

```
<a href="{{ path( "ez_urlalias", {"locationId": 123} )
}}">Some link to a location, with its Id only</a>
```

You can also use the Content ID. In that case the generated link will point to the Content item's main Location.

```
<a href="{{ path( "ez_urlalias", {"contentId": 456} )
}}">Some link from a contentId</a>
```

Under the hood
In the backend, `path()` uses the Router to generate links.

This makes it also easy to generate links from PHP, via the `router` service.

See also: Cross-siteaccess links

Render embedded content objects

Rendering an embedded content from a Twig template is pretty straight forward as you just need to **do a subrequest with `ez_content` controller**.

Using `ez_content` controller

This controller is exactly the same as the ViewController presented above. It has one main `viewAction`, that renders a Content item.

You can use this controller from templates with the following syntax:

```
{{ render(controller("ez_content:viewAction",
{"contentId": 123, "viewType": "line"})) }}
```

The example above renders the Content whose ID is **123**, with the view type **line**.

Reference of `ez_content` controller follows the syntax of *controllers as a service*, as explained in Symfony documentation.

Available arguments

As with any controller, you can pass arguments to `ez_content:viewLocation` or `ez_content:viewContent` to fit your needs.

Name	Description	Type	Default value
`contentId`	ID of the Content item you want to render. **Only for `ez_content:viewContent`**	integer	N/A
`locationId`	ID of the Location you want to render. **Only for `ez_content:viewLocation`**	integer	Content item's main location, if defined
`viewType`	The view type you want to render your Content item/Location in. Will be used by the ViewManager to select a corresponding template, according to defined rules. Example: full, line, my_custom_view, etc.	string	full
`layout`	Indicates if the sub-view needs to use the main layout (see available variables in a view template)	boolean	false

`params`	Hash of variables you want to inject to sub-template, key being the exposed variable name. `{{ render( controller( "ez_content:viewAction", { "contentId": 123, "viewType": "line", "params": { "some_variable": "some_value" } } ) ) }}`	hash	empty hash

Render block

You can specify which controller will be called for a specific block view match, much like defining custom controllers for location view or content view match.

Also, since there are two possible actions with which one can view a block: `ez_page:viewBlock` and `ez_page:viewBlockById`, it is possible to specify a controller action with a signature matching either one of the original actions.

Example of configuration in `app/config/ezplatform.yml`:

```
ezpublish:
    system:
        eng_frontend_group:
            block_view:
                ContentGrid:
                    template:
NetgenSiteBundle:block:content_grid.html.twig
                    controller:
NetgenSiteBundle:Block:viewContentGridBlock
                    match:
                        Type: ContentGrid
```

ESI

Just as for regular Symfony controllers, you can take advantage of ESI and use different cache levels:

Using ESI

```
{{ render_esi(controller("ez_content:viewAction",
{"contentId": 123, "viewType": "line"})) }}
```

Only scalable variables can be sent via render_esi (not object)

Asynchronous rendering

Symfony also supports asynchronous content rendering with the help of hinclude.js library.

Asynchronous rendering

```
{{ render_hinclude(controller("ez_content:viewAction",
{"contentId": 123, "viewType": "line"})) }}
```

Only scalable variables can be sent via render_hinclude (not object)

Display a default text

If you want to display a default text while a controller is loaded asynchronously, you have to pass a second parameter to your render_hinclude twig function.

Display a default text during asynchronous loading of a controller

```
{{
render_hinclude(controller('EzCorporateDesignBundle:Header:userLinks'), {'default': "<div
style='color:red'>loading</div>"}) }}
```

See also: Custom controllers

> hinclude.js needs to be properly included in your layout to work.
>
> Please refer to Symfony documentation for all available options.

Content and Location view providers

View\Manager & View\Provider

The role of the `(eZ\Publish\Core\MVC\Symfony\)View\Manager` is to select the right template for displaying a given content item or location. It aggregates objects called *content and location view providers* which respectively implement `eZ\Publish\Core\MVC\Symfony\View\Provider\Content` and `eZ\Publish\Core\MVC\Symfony\View\Provider\Location` interfaces.

Each time a content item is to be displayed through the `Content\ViewController`, the `View\Manager` iterates over the registered content or location `View\Provider` objects and calls `getView()`.

Provided View\Provider implementations

Name	Usage
`View provider configuration`	Based on application configuration. Formerly known as *Template override system*.

`eZ\Publish\Core\MVC\Legacy\View\Provider\Content`
 	
`eZ\Publish\Core\MVC\Legacy\View\Provider\Location`
	Forwards view selection to the legacy kernel by running the old content/view module.
	Pagelayout used is the one configured in `ezpublish_legacy.<scope>.view_default_layout`.
	For more details about the `<scope>` please refer to the scope configuration documentation.

Custom View\Provider

Difference between `View\Provider\Location` and `View\Provider\Content`

- A `View\Provider\Location` only deals with `Location` objects and implements `eZ\Publish\Core\MVC\Symfony\View\Provider\Location` interface.
- A `View\Provider\Content` only deals with `ContentInfo` objects and implements `eZ\Publish\Core\MVC\Symfony\View\Provider\Content` interface.

When to develop a custom `View\Provider\(Location|Content)`

- You want a custom template selection based on a very specific state of your application
- You depend on external resources for view selection
- You want to override the default one (based on configuration) for some reason

`View\Provider` objects need to be properly registered in the service container with the **`ezpublish.location_view_provider`** or **`ezpublish.content_view_provider`** service tag.

```
parameters:
    acme.location_view_provider.class:
Acme\DemoBundle\Content\MyLocationViewProvider

services:
    acme.location_view_provider:
        class: %ezdemo.location_view_provider.class%
        tags:
            - {name: ezpublish.location_view_provider,
priority: 30}
```

Tag attribute name	Usage
priority	An integer giving the priority to the `View\Provider\(Content|Location)` in the `View\Manager`. The priority range is **from -255 to 255**

Example

Custom View\Provider\Location

```
<?php

namespace Acme\DemoBundle\Content;

use eZ\Publish\Core\MVC\Symfony\View\ContentView;
use eZ\Publish\Core\MVC\Symfony\View\Provider\Location
as LocationViewProvider;
use eZ\Publish\API\Repository\Values\Content\Location;

class MyLocationViewProvider implements
LocationViewProvider
{
    /**
     * Returns a ContentView object corresponding to
$location, or void if not applicable
     *
     * @param
\eZ\Publish\API\Repository\Values\Content\Location
$location
     * @param string $viewType
     * @return
\eZ\Publish\Core\MVC\Symfony\View\ContentView|null
     */
    public function getView( Location $location,
$viewType )
    {
        // Let's check location Id
        switch ( $location->id )
        {
            // Special template for home page, passing
"foo" variable to the template
            case 2:
                return new ContentView(
"AcmeDemoBundle:$viewType:home.html.twig", array( 'foo'
=> 'bar' ) );
        }

        // ContentType identifier (formerly "class
identifier")
        switch (
$location->contentInfo->contentType->identifier )
        {
            // For view full, it will load
AcmeDemoBundle:full:small_folder.html.twig
            case 'folder':
                return new ContentView(
"AcmeDemoBundle:$viewType:small_folder.html.twig" );
        }
    }
}
```

Binary and Media download

Unlike image files, files stored in BinaryFile or Media Fields may be limited to certain User Roles. As such, they are not publicly downloadable from disk, and are instead served by Symfony, using a custom route that runs the necessary checks. This route is automatically generated as the `url` property for those Fields values.

The content/download route

The route follows this pattern: `/content/download/{contentId}/{fieldIdentifier}/{filename}`. Example: `/content/download/68/file/My-file.pdf`.

It also accepts optional query parameters:

- `version`: the version number that the file must be downloaded for. Requires the versionview permission. If not specified, the published version is used.
- `inLanguage`: The language the file should be downloaded in. If not specified, the most prioritized language for the siteaccess will be used.

The ez_render_field twig helper will by default generate a working link.

REST API: The `uri` property contains a valid download URL

The `uri` property of Binary Fields in REST contain a valid URL, of the same format than the Public API, prefixed with the same host than the REST Request.

> For more information about REST API see the documentation.

Reference

> **Symfony & Twig template functions/filters/tags**
> For template functionality provided by Symfony Framework, see Symfony Twig Extensions Reference page. For those provided by the underlying Twig template engine, see Twig Reference page

Twig functions reference

See Twig Functions Reference for detailed information on all available Twig functions.

Extensibility

Events

Introduction

This page presents the events that are triggered by eZ Platform.

eZ Publish Core

Event name	Triggered when...	Usage

ezpublish.siteaccess	After the SiteAccess matching has occurred.	Gives further control on the matched SiteAccess. The event listener method receives an `eZ\Publish\Core\MVC\Symfony\Event\PostSiteAccessMatchEvent` object.
ezpublish.pre_content_view	Right before a view is rendered for a content item, via the content view controller.	This event is triggered by the view manager and allows you to inject additional parameters to the content view template. The event listener method receives an `eZ\Publish\Core\MVC\Symfony\Event\PreContentViewEvent` object.
ezpublish.api.contentException	The API throws an exception that could not be caught internally (missing field type, internal error...).	This event allows further programmatic handling (like rendering a custom view) for the exception thrown. The event listener method receives an `eZ\Publish\Core\MVC\Symfony\Event\APIContentExceptionEvent` object.