2. Browsing, finding, viewing

We will start by going through the various ways to find and retrieve content from eZ Platform using the API. While this will be covered in further dedicated documentation, it is necessary to explain a few basic concepts of the Public API. In the following recipes, you will learn about the general principles of the API as they are introduced in individual recipes.

Displaying values from a Content item

In this recipe, we will see how to fetch a Content item from the repository, and obtain its Field's content.

Let's first see the full code. You can see the Command line version at https://github.com/ezsystems/CookbookBundle/blob/master/Command/ViewContentCommand.php.

Viewing content

```
$repository = $this->getContainer()->get( 'ezpublish.api.repository' );
$contentService = $repository->getContentService();
$contentTypeService = $repository->getContentTypeService();
$fieldTypeService = $repository->getFieldTypeService();

try
{
    $content = $contentService->loadContent( 66 );
    $contentType = $contentTypeService->loadContentType(
$content->contentInfo->contentTypeId );
    // iterate over the field definitions of the content type and print out each
field's identifier and value
    foreach( $contentType->fieldDefinitions as $fieldDefinition )
    {
        $output->write( $fieldDefinition->identifier . ": " );
        $fieldType = $fieldTypeService->getFieldType(
$fieldDefinition->fieldTypeIdentifier );
        $field = $content->getField( $fieldDefinition->identifier );

  // We use the Field's toHash() method to get readable content out of the Field
        $valueHash = $fieldType->toHash( $field->value );
        $output->writeln( $valueHash );
    }
}
catch( \eZ\Publish\API\Repository\Exceptions\NotFoundException $e )
{
    // if the id is not found
    $output->writeln( "No content with id $contentId" );
}
catch( \eZ\Publish\API\Repository\Exceptions\UnauthorizedException $e )
{
    // not allowed to read this content
    $output->writeln( "Anonymous users are not allowed to read content with id
$contentId" );
}
```

Let's analyze this code block by block.

```
$repository = $this->getContainer()->get( 'ezpublish.api.repository' );
$contentService = $repository->getContentService();
$contentTypeService = $repository->getContentTypeService();
$fieldTypeService = $repository->getFieldTypeService();
```

This is the initialization part. As explained above, everything in the Public API goes through the repository via dedicated services. We get the repository from the service container, using the method `get()` of our container, obtained via `$this->getContainer()`. Using our `$repository` variable, we fetch the two services we will need using `getContentService()` and `getFieldTypeService()`.

```
try
{
    // iterate over the field definitions of the content type and print out each
field's identifier and value
    $content = $contentService->loadContent( 66 );
```

Everything starting from line 5 is about getting our Content and iterating over its Fields. You can see that the whole logic is part of a `try/catch` b lock. Since the Public API uses Exceptions for error handling, this is strongly encouraged, as it will allow you to conditionally catch the various errors that may happen. We will cover the exceptions we expect in a later paragraph.

The first thing we do is use the Content Service to load a Content item using its ID, 66: `$contentService->loadContent ( 66 )`. As you can see on the API doc page, this method expects a Content ID, and returns a Content Value Object.

```
foreach( $contentType->fieldDefinitions as $fieldDefinition )
{
    // ignore ezpage
    if( $fieldDefinition->fieldTypeIdentifier == 'ezpage' )
        continue;
    $output->write( $fieldDefinition->identifier . ": " );
    $fieldType = $fieldTypeService->getFieldType(
$fieldDefinition->fieldTypeIdentifier );
    $fieldValue = $content->getFieldValue( $fieldDefinition->identifier );
    $valueHash = $fieldType->toHash( $fieldValue );
    $output->writeln( $valueHash );
}
```

This block is the one that actually displays the value.

It iterates over the Content item's Fields using the Content Type's FieldDefinitions (`$contentType->fieldDefinitions`).

For each Field Definition, we start by displaying its identifier (`$fieldDefinition->identifier`). We then get the Field Type instance using the Field Type Service (`$fieldTypeService->getFieldType( $fieldDefinition->fieldTypeIdentifier )`). This method expects the requested Field Type's identifier, as a string (ezstring, ezxmltext, etc.), and returns an `eZ\Publish\API\Repository\FieldType` object.

The Field Value object is obtained using the `getFieldValue()` method of the Content Value Object which we obtained using `ContentServic e::loadContent()`.

Using the Field Type object, we can convert the Field Value to a hash using the `toHash()` method, provided by every Field Type. This method returns a primitive type (string, hash) out of a Field instance.

With this example, you should get a first idea on how you interact with the API. Everything is done through services, each service being responsible for a specific part of the repository (Content, Field Type, etc.).

Loading Content in different languages
Since we didn't specify any language code, our Field object is returned in the default language, depending on your language settings in `ezplatform.yml`. If you want to use a non-default language, you can specify a language code in the `getField()` call:

```
$content->getFieldValue( $fieldDefinition->identifier, 'fre-FR' )
```

Exceptions handling

```
catch ( \eZ\Publish\API\Repository\Exceptions\NotFoundException $e )
{
    $output->writeln( "<error>No content with id $contentId found</error>" );
}
catch ( \eZ\Publish\API\Repository\Exceptions\UnauthorizedException $e )
{
    $output->writeln( "<error>Permission denied on content with id $contentId</error>"
);
}
```

As said earlier, the Public API uses Exceptions to handle errors. Each method of the API may throw different exceptions, depending on what it does. Which exceptions can be thrown is usually documented for each method. In our case, `loadContent()` may throw two types of exceptions: `NotFoundException`, if the requested ID isn't found, and `UnauthorizedException` if the currently logged in user isn't allowed to view the requested content.

It is a good practice to cover each exception you expect to happen. In this case, since our Command takes the Content ID as a parameter, this ID may either not exist, or the referenced Content item may not be visible to our user. Both cases are covered with explicit error messages.

Traversing a Location subtree

This recipe will show how to traverse a Location's subtree. The full code implements a command that takes a Location ID as an argument and recursively prints this location's subtree.

Full code
https://github.com/ezsystems/CookbookBundle/blob/master/Command/BrowseLocationsCommand.php

In this code, we introduce the LocationService. This service is used to interact with Locations. We use two methods from this service: `loadLocation()` , and `loadLocationChildren()` .

Loading a Location

```
try
{
    // load the starting location and browse
    $location = $this->locationService->loadLocation( $locationId );
    $this->browseLocation( $location, $output );
}
catch ( \eZ\Publish\API\Repository\Exceptions\NotFoundException $e )
{
    $output->writeln( "<error>No location found with id $locationId</error>" );
}
catch( \eZ\Publish\API\Repository\Exceptions\UnauthorizedException $e )
{
    $output->writeln( "<error>Current users are not allowed to read location with id
$locationId</error>" );
}
```

As for the ContentService, `loadLocation()` returns a Value Object, here a `Location` . Errors are handled with exceptions: `NotFoundException` if the Location ID couldn't be found, and `UnauthorizedException` if the current repository user isn't allowed to view this Location.

Iterating over a Location's children

```
private function browseLocation( Location $location, OutputInterface $output, $depth =
0 )
{
    $childLocationList = $this->locationService->loadLocationChildren( $location,
$offset = 0, $limit = -1 );
    // If offset and limit had been specified to something else then "all", then
$childLocationList->totalCount contains the total count for iteration use
    foreach ( $childLocationList->locations as $childLocation )
    {
        $this->browseLocation( $childLocation, $output, $depth + 1 );
    }
}
```

`LocationService::loadLocationChildren()` returns a LocationList Value Objects that we can iterate over.

Note that unlike `loadLocation()`, we don't need to care for permissions here: the currently logged-in user's permissions will be respected when loading children, and Locations that can't be viewed won't be returned at all.

> **Full code**
> Should you need more advanced children fetching methods, the `SearchService` is what you are looking for.

Viewing Content Metadata

Content is a central piece in the Public API. You will often need to start from a Content item, and dig in from its metadata. Basic content metadata is made available through `ContentInfo` objects. This Value Object mostly provides primitive fields: `contentTypeId`, `publishedDate` or `mainLocationId`. But it is also used to request further Content-related Value Objects from various services.

The full example implements an `ezpublish:cookbook:view_content_metadata` command that prints out all the available metadata, given a Content ID.

> **Full code**
> https://github.com/ezsystems/CookbookBundle/blob/master/Command/ViewContentMetaDataCommand.php

We introduce here several new services: `URLAliasService` , `UserService` and `SectionService` . The concept should be familiar to you now.

Services initialization

```
/** @var $repository \eZ\Publish\API\Repository\Repository */
$repository = $this->getContainer()->get( 'ezpublish.api.repository' );
$contentService = $repository->getContentService();
$locationService = $repository->getLocationService();
$urlAliasService = $repository->getURLAliasService();
$sectionService = $repository->getSectionService();
$userService = $repository->getUserService();
```

Setting the Repository User

> In a command line script, the repository runs as if executed by the anonymous user. In order to identify it as a different user, you need to use the `UserService` as follows:

```
$administratorUser = $userService->loadUser( 14 );
$repository->setCurrentUser( $administratorUser );
```

This may be crucial when writing maintenance or synchronization scripts.

This is of course not required in template functions or controller code, as the HTTP layer will take care of identifying the user, and automatically set it in the repository.

The ContentInfo Value Object

We will now load a `ContentInfo` object using the provided ID and use it to get our Content item's metadata

```
$contentInfo = $contentService->loadContentInfo( $contentId );
```

Locations

Getting Content Locations

```
// show all locations of the content
$locations = $locationService->loadLocations( $contentInfo );
$output->writeln( "<info>LOCATIONS</info>" );
foreach ( $locations as $location )
{
    $urlAlias = $urlAliasService->reverseLookup( $location );
    $output->writeln( "  $location->pathString  ($urlAlias->path)" );
}
```

We first use `LocationService::loadLocations()` to **get** the **Locations** for our `ContentInfo`. This method returns an array of `Location` Value Objects. In this example, we print out the Location's path string (/path/to/content). We also use URLAliasService::reverseLookup() to get the Location's main URLAlias.

Relations

We now want to list relations from and to our Content. Since relations are versioned, we need to feed the `ContentService::loadRelations()` with a `VersionInfo` object. We can get the current version's `VersionInfo` using `ContentService::loadVersionInfo()`. If we had been looking for an archived version, we could have specified the version number as the second argument to this method.

Browsing a Content's relations

```
// show all relations of the current version
$versionInfo = $contentService->loadVersionInfo( $contentInfo );
$relations = $contentService->loadRelations( $versionInfo );
if ( count( $relations ) )
{
    $output->writeln( "<info>RELATIONS</info>" );
    foreach ( $relations as $relation )
    {
        $name = $relation->destinationContentInfo->name;
        $output->write( "  Relation of type " . $this->outputRelationType(
$relation->type ) . " to content $name" );
    }
}
```

We can iterate over the Relation objects array we got from `loadRelations()`, and use these Value Objects to get data about our relations. It has two main properties: `destinationContentInfo`, and `sourceContentInfo`. They also hold the relation type (embed, common, etc.), and the optional Field this relations is made with.

ContentInfo properties

We can of course get our Content item's metadata by using the Value Object's properties.

Primitive object metadata

```
// show meta data
$output->writeln( "\n<info>METADATA</info>" );
$output->writeln( "  <info>Name:</info> " . $contentInfo->name );
$output->writeln( "  <info>Type:</info> " . $contentType->identifier );
$output->writeln( "  <info>Last modified:</info> " .
$contentInfo->modificationDate->format( 'Y-m-d' ) );
$output->writeln( "  <info>Published:</info> ". $contentInfo->publishedDate->format(
'Y-m-d' ) );
$output->writeln( "  <info>RemoteId:</info> $contentInfo->remoteId" );
$output->writeln( "  <info>Main Language:</info> $contentInfo->mainLanguageCode" );
$output->writeln( "  <info>Always available:</info> " . (
$contentInfo->alwaysAvailable ? 'Yes' : 'No' ) );
```

Owning user

We can use `UserService::loadUser()` with Content `ownerId` property of our `ContentInfo` to load the Content's owner as a `User` Value Object.

```
$owner = $userService->loadUser( $contentInfo->ownerId );
$output->writeln( "  <info>Owner:</info> " . $owner->contentInfo->name );
```

To get the current version's creator, and not the content's owner, you need to use the `creatorId` property from the current version's `VersionInfo` object.

Section

The Section's ID can be found in the `sectionId` property of the `ContentInfo` object. To get the matching Section Value Object, you need to use the `SectionService::loadSection()` method.

```
$section = $sectionService->loadSection( $contentInfo->sectionId );
$output->writeln( "  <info>Section:</info> $section->name" );
```

Versions

To conclude we can also iterate over the Content's version, as `VersionInfo` Value Objects.

```
$versionInfoArray = $contentService->loadVersions( $contentInfo );
if ( count( $versionInfoArray ) )
{
    $output->writeln( "\n<info>VERSIONS</info>" );
    foreach ( $versionInfoArray as $versionInfo )
    {
        $creator = $userService->loadUser( $versionInfo->creatorId );
        $output->write( "  Version $versionInfo->versionNo " );
        $output->write( " by " . $creator->contentInfo->name );
        $output->writeln( " " . $this->outputStatus( $versionInfo->status ) . " " .
$versionInfo->initialLanguageCode );
    }
}
```

We use the `ContentService::loadVersions()` method and get an array of `VersionInfo` objects.

Search

In this section we will cover how the `SearchService` can be used to search for Content, by using a `Query` and a combinations of `Criteria` y ou will get a `SearchResult` object back containing list of Content and count of total hits. In the future this object will also include facets, spell checking and "more like this" when running on a backend that supports it (for instance Solr).

Performing a simple full text search

> **Full code**
> https://github.com/ezsystems/CookbookBundle/blob/master/Command/FindContentCommand.php

In this recipe, we will run a simple full text search over every compatible attribute.

Query and Criterion objects

We introduce here a new object: `Query`. It is used to build up a Content query based on a set of `Criterion` objects.

```
$query = new \eZ\Publish\API\Repository\Values\Content\Query();
$query->filter = new Query\Criterion\FullText( $text );
```

Multiple criteria can be grouped together using "logical criteria", such as LogicalAnd or LogicalOr. Since in this case we only want to run a text

search, we simply use a `FullText` criterion object.

> The full list of criteria can be found on your installation in the following directory vendor/ezsystems/ezpublish-kernel/eZ/Publish/API/Repository/Values/Content/Query/Criterion. Additionally you may look at integration tests like vendor/ezsystems/ezpublish-kernel/eZ/Publish/API/Repository/Tests/SearchServiceTest.php for more details on how these are used.

Running the search query and using the results

The `Query` object is given as an argument to `SearchService::findContent()` . This method returns a `SearchResult` object. This object provides you with various information about the search operation (number of results, time taken, spelling suggestions, or facets, as well as, of course, the results themselves.

```
$result = $searchService->findContent( $query );
$output->writeln( 'Found ' . $result->totalCount . ' items' );
foreach ( $result->searchHits as $searchHit )
{
    $output->writeln( $searchHit->valueObject->contentInfo->name );
}
```

The `searchHits` properties of the `SearchResult` object is an array of `SearchHit` objects. In `valueObject` property of `SearchHit`, you will find the `Content` object that matches the given `Query`.

> **Tip**
> If you you are searching using a unique identifier, for instance using the Content ID or Content remote ID criterion, then you can use `SearchService::findSingle()` , this takes a Criterion and returns a single Content item, or throws a `NotFound` exception if none is found.

Performing an advanced search

> **Full code**
> https://github.com/ezsystems/CookbookBundle/blob/master/Command/FindContent2Command.php

As explained in the previous chapter, Criterion objects are grouped together using logical criteria. We will now see how multiple criteria objects can be combined into a fine grained search `Query`.

```
use eZ\Publish\API\Repository\Values\Content\Query\Criterion;
use eZ\Publish\API\Repository\Values\Content;

// [...]

$query = new Query();
$criterion1 = new Criterion\Subtree( $locationService->loadLocation( 2 )->pathString
);
$criterion2 = new Criterion\ContentTypeIdentifier( 'folder' );
$query->criterion = new Criterion\LogicalAnd(
    array( $criterion1, $criterion2 )
);

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

A `Subtree` criterion limits the search to the subtree with pathString, which looks like: `/1/2/`. A `ContentTypeId` Criterion to limit the search to

Content of Content Type 1. Those two criteria are grouped with a `LogicalAnd` operator. The query is executed as before, with `SearchService::findContent()`.

Performing a fetch like search

> **Full code**
> https://github.com/ezsystems/CookbookBundle/blob/master/Command/FindContent3Command.php

A search isn't only meant for searching, it also provides the future interface for what you in eZ Publish 4.x would know as a content "fetch". And as this is totally backend agnostic, in future versions this will be powered by either Solr or ElasticSearch meaning it also replaces "ezfind" fetch functions.

Following the examples above we now change it a bit to combine several criteria with both an AND and an OR condition.

```
use eZ\Publish\API\Repository\Values\Content\Query\Criterion;
use eZ\Publish\API\Repository\Values\Content;

// [...]

$query = new Query();
$query->criterion = new Criterion\LogicalAnd(
    array(
        new Criterion\ParentLocationId( 2 ),
        new Criterion\LogicalOr(
            array(
                new Criterion\ContentTypeIdentifier( 'folder' ),
                new Criterion\ContentTypeId( 2 )
            )
        )
    )
);

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

A `ParentLocationId` criterion limits the search to the children of location 2. An array of "`ContentTypeId`" Criteria to limit the search to Content of ContentType's with id 1 or 2 grouped in a `LogicalOr` operator. Those two criteria are grouped with a `LogicalAnd` operator. As always the query is executed as before, with `SearchService::findContent()`.

> Want to do a subtree filter? Change the location filter to use the Subtree criterion filter as shown in the advanced search example above.

Using in() instead of OR

The above example is fine, but it can be optimized a bit by taking advantage of the fact that all filter criteria support being given an array of values (IN operator) instead of a single value (EQ operator).

You can also use the `ContentTypeIdentifier` Criterion:

```
use eZ\Publish\API\Repository\Values\Content\Query\Criterion;
use eZ\Publish\API\Repository\Values\Content;

// [...]

$query = new Query();
$query->criterion = new Criterion\LogicalAnd(
    array(
        new Criterion\ParentLocationId( 2 ),
        new Criterion\ContentTypeIdentifier( array( 'article', 'folder' ) )
    )
);

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

Tip
All filter criteria are capable of doing an "IN" selection, the ParentLocationId above could, for example, have been provided "array(2, 43)" to include second level children in both your content tree (2) and your media tree (43).

Performing a pure search count

In many cases you might need the number of Content items matching a search, but with no need to do anything else with the results.

Thanks to the fact that the " searchHits " property of the `SearchResult` object always refers to the total amount, it is enough to run a standard search and set $limit to 0. This way no results will be retrieved, and the search will not be slowed down, even when the number of matching results is huge.

```
use eZ\Publish\API\Repository\Values\Content\Query;

// [...]

$query = new Query();
$query->limit = 0;

// [...] ( Add criteria as shown above )

$resultCount = $searchService->findContent( $query )->totalCount;
```