Extending the REST API

The eZ Publish 5 REST API comes with a framework that makes it quite easy to extend the API for your own needs.

> While most of what can be found here will still apply in the future, the structure of the REST API and its components will evolve before the actual release.

Requirements

As of version 2013.7 / 5.2, REST routes are required to use the eZ Publish 5 REST API prefix, `/api/ezp/v2`. You can create new resources below this prefix.

To do so, you will/may need to create

- a Controller that will handle your route actions
- a Route, in your bundle's routing file
- a Controller action
- Optionally, a `ValueObjectVisitor` (if your Controller returns an object that doesn't already have a converter)
- Optionally, an `InputParser`

Controller

To create a REST controller, you need to extend the `ezpublish_rest.controller.base` service, as well as the `eZ\Publish\Core\REST\Server\Controller` class.

Let's create a very simple controller, that has a `sayHello()` method, that takes a name as an argument.

My/Bundle/RestBundle/Rest/Controller/DefaultController.php

```
namespace My\Bundle\RestBundle\Rest\Controller;

use eZ\Publish\Core\REST\Server\Controller as BaseController;

class DefaultController extends BaseController
{
    public function sayHello( $name )
    {
        // @todo Implement me
    }
}
```

Route

As said earlier, your REST routes are required to use the REST URI prefix. To do so, the easiest way is to import your routing file using this prefix.

ezpublish/config/routing.yml

```
myRestBundle_rest_routes:
    resource: "@MyRestBundle/Resources/config/routing_rest.yml"
    prefix:   %ezpublish_rest.path_prefix%
```

Using a distinct file for REST routes allows you to use the prefix for all this file's routes without affecting other routes from your bundle.

Next, you need to create the REST route. We need to define the route's controller as a service since our controller was defined as such.

My/Bundle/RestBundle/Resources/config/routing_rest.yml

```
myRestBundle_hello_world:
    pattern: /my_rest_bundle/hello/{name}
    defaults:
        _controller: myRestBundle.controller.default:sayHello
    methods: [GET]
```

Due to EZP-23016 - Custom REST API routes (v2) are not accessible from the legacy backend CLOSED, custom REST routes must be prefixed with `ezpublish_rest_`, or they won't be detected correctly.

Controller action

Unlike standard Symfony 2 controllers, the REST ones don't return an `HttpFoundation\Response` object, but a `ValueObject`. This object will during the kernel run be converted, using a ValueObjectVisitor, to a proper Symfony 2 response. One benefit is that when multiple controllers return the same object, such as a Content item or a Location, the visitor will be re-used.

Let's say that our Controller will return a `My\Bundle\RestBundle\Rest\Values\Hello`

My/Bundle/RestBundle/Rest/Values/Hello.php

```
namespace My\Bundle\RestBundle\Rest\Values;

class Hello
{
    public $name;

    public function __construct( $name )
    {
        $this->name = $name;
    }
}
```

We will return an instance of this class from our `sayHello()` controller method.

My/Bundle/RestBundle/Rest/Controller/DefaultController.php

```
namespace My\Bundle\RestBundle\Controller;

use eZ\Publish\Core\REST\Server\Controller as BaseController;
use My\Bundle\RestBundle\Rest\Values\Hello as HelloValue;

class DefaultController extends BaseController
{
    public function sayHello( $name )
    {
        return new HelloValue( $name );
    }
}
```

And that's it. Outputting this object in the Response requires that we create a ValueObjectVisitor.

ValueObjectVisitor

A ValueObjectVisitor will take a Value returned by a REST controller, whatever the class, and will transform it into data that can be converted, either to json or XML. Those visitors are registered as services, and tagged with `ezpublish_rest.output.value_object_visitor`. The tag attribute says which class this Visitor applies to.

Let's create the service for our ValueObjectVisitor first.

My/Bundle/RestBundle/Resources/config/services.yml

```
services:
    myRestBundle.value_object_visitor.hello:
        parent: ezpublish_rest.output.value_object_visitor.base
        class: My\Bundle\RestBundle\Rest\ValueObjectVisitor\Hello
        tags:
            - { name: ezpublish_rest.output.value_object_visitor, type:
My\Bundle\RestBundle\Rest\Values\Hello }
```

Let's create our visitor next. It must extend the `eZ\Publish\Core\REST\Common\Output\ValueObjectVisitor` abstract class, and implement the `visit()` method.
It will receive as arguments:

- `$visitor`: The output visitor. Can be used to set custom response headers (`setHeader( $name, $value )`), HTTP status code (`setStatus( $statusCode )`)...
- `$generator`: The actual Response generator. It provides you with a DOM like API.
- `$data`: the visited data, the exact object you returned from the controller

My/Bundle/RestBundle/Rest/Controller/Default.php

```
namespace My\Bundle\RestBundle\Rest\ValueObjectVisitor;

use eZ\Publish\Core\REST\Common\Output\ValueObjectVisitor;
use eZ\Publish\Core\REST\Common\Output\Generator;
use eZ\Publish\Core\REST\Common\Output\Visitor;

class Hello extends ValueObjectVisitor
{
    public function visit( Visitor $visitor, Generator $generator, $data )
    {
        $generator->startValueElement( 'Hello', $data->name );
        $generator->endValueElement( 'Hello' );
    }
}
```

Do not hesitate to look into the built-in ValueObjectVisitors, in `eZ/Publish/Core/REST/Server/Output/ValueObjectVisitor`, for more examples.

Cache handling

The easiest way to handle cache is to re-use the `CachedValue` Value Object. It acts as a proxy, and adds the cache headers, depending on the configuration, for a given object and set of options.

When you want the response to be cached, return an instance of CachedValue, with your Value Object as the argument. You can also pass a location id using the second argument, so that the Response is tagged with it:

```
return new CachedValue($helloValue, ['locationId', 42]);
```

Input parser

What we have seen above covers requests that don't require an input payload, such as GET or DELETE. If you need to provide your controller with parameters, either in JSON or XML, the parameter struct requires an Input Parser so that the payload can be converted to an actual ValueObject.

Each payload is dispatched to its Input Parser based on the request's Content-Type header. For example, a request with a Content-Type of `application/vnd.ez.api.ContentCreate` will be parsed by `eZ\Publish\Core\REST\Server\Input\Parser\ContentCreate`. This parser will build and return a `ContentCreateStruct` that can then be used to create content with the Public API.

Those input parsers are provided with a pre-parsed version of the input payload, as an associative array, and don't have to care about the actual format (XML or JSON).

Let's see what it would look like with a Content-Type of application/vnd.my.Greetings, that would send this as XML:

application/vnd.my.Greetings+xml

```
<?xml version="1.0" encoding="utf-8"?>
<Greetings>
    <name>John doe</name>
</Greetings>
```

First, we need to create a service with the appropriate tag in services.yml.

My/Bundle/RestBundle/Resources/config/services.yml

```
services:
    myRestBundle.input_parser.Greetings:
        parent: ezpublish_rest.input.parser
        class: My\Bundle\RestBundle\Rest\InputParser\Greetings
        tags:
            - { name: ezpublish_rest.input.parser, mediaType:
application/vnd.my.Greetings }
```

The mediaType attribute of the ezpublish_rest.input.parser tag maps our Content Type to the input parser.

Let's implement our parser. It must extend eZ\Publish\Core\REST\Server\Input\Parser, and implement the `parse()` method. It accepts as an argument the input payload, `$data`, as an array, and an instance of `ParsingDispatcher` that can be used to forward parsing of embedded content.

For convenience, we will consider that our input parser returns an instance of our `Value\Hello` class.

My/Bundle/RestBundle/Rest/InputParser/Greetings.php

```
namespace My\Bundle\RestBundle\Rest\InputParser;

use eZ\Publish\Core\REST\Common\Input\BaseParser;
use eZ\Publish\Core\REST\Common\Input\ParsingDispatcher;
use My\Bundle\RestBundle\Rest\Value\Hello;
use eZ\Publish\Core\REST\Common\Exceptions;


class Greetings extends BaseParser
{
    /**
     * @return My\Bundle\RestBundle\Rest\Value\Hello
     */
    public function parse( array $data, ParsingDispatcher $parsingDispatcher )
 {
        // re-using the REST exceptions will make sure that those already have a
ValueObjectVisitor
        if ( !isset( $data['name'] ) )
            throw new Exceptions\Parser( "Missing or invalid 'name' element for
Greetings." );


        return new Hello( $data['name'] );
 }
}
```

My/Bundle/RestBundle/Resources/config/services.yml

```
services:
    myRestBundle.controller.default:
        class: My\Bundle\RestBundle\Rest\Controller\Default
        parent: ezpublish_rest.controller.base
```

Do not hesitate to look into the built-in InputParsers, in `eZ/Publish/Core/REST/Server/Input/Parser`, for more examples.