Blog: REST Controllers, Routing, JSON Responses

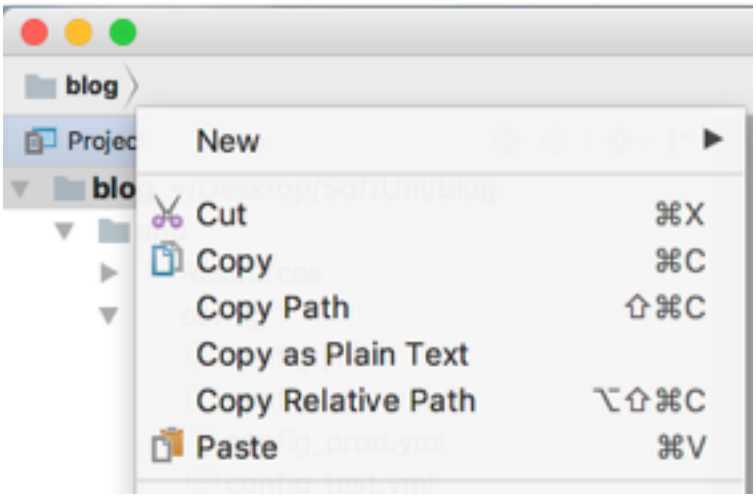
This document defines a complete walkthrough of creating a REST API for our Blog applicatio, from setting up the serializer bundle, creating our own API bundle, ending up with fully functioning REST API.

I. Install serialization bundle

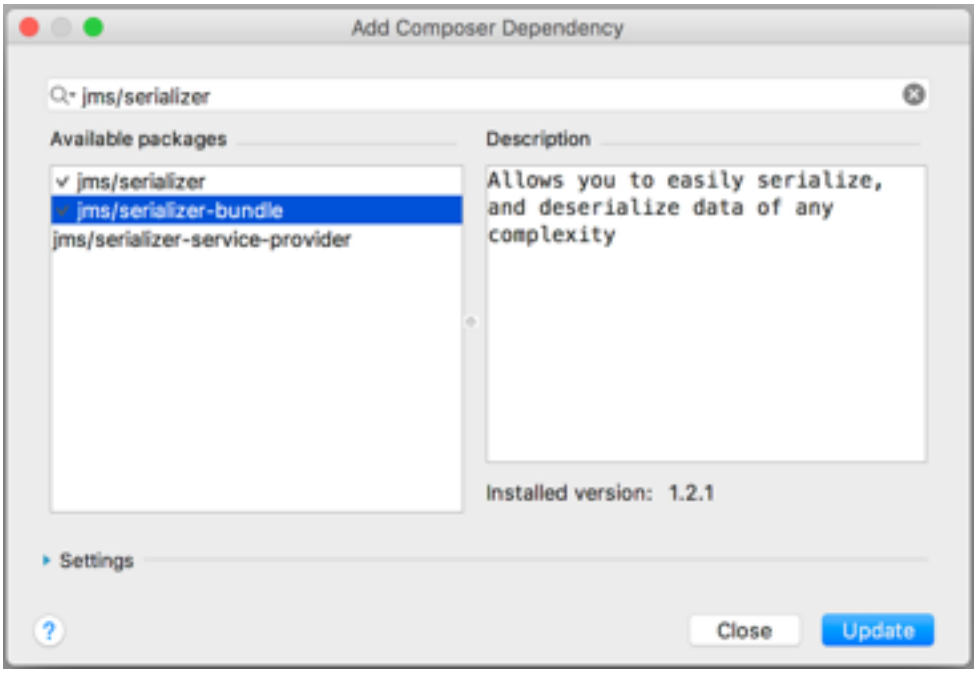
For the purpose of this exercise we need functionallity which serialize/deserialize our Doctrine entities to JSON and vice versa - JMSSerializerBundle

1. Install the Bundle

Install from PHPStorm



In the search box type **jms/serializer-bundle** and click **Update**

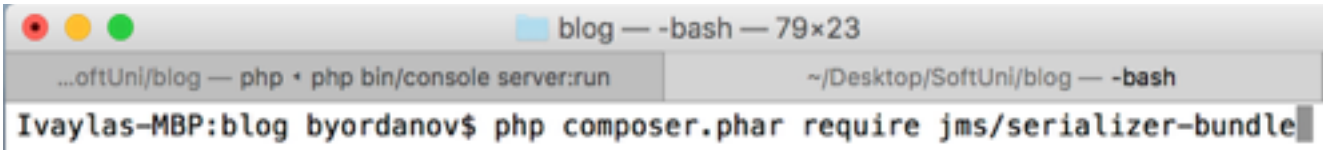


Install using Composer

Open terminal / console and go to the directory of your project

If you still don't have **Composer** you can download it from here: http://getcomposer.org

In terminal / cmd type following command: **php composer.phar require jms/serializer-bundle**



The command will download the bundle and incude it automatically to the autoloader. You don't need to include anything anywhere.

2. Activate the Bundle

After instalation every bundle needs to be added to our application.

Open **app/AppKernel.php** file and add the following line:

new JMS\SerializerBundle\JMSSerializerBundle(),

```
$bundles = [
    new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
    new Symfony\Bundle\SecurityBundle\SecurityBundle(),
    new Symfony\Bundle\TwigBundle\TwigBundle(),
    new Symfony\Bundle\MonologBundle\MonologBundle(),
    new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
    new SoftUniBlogBundle\SoftUniBlogBundle(),
    new JMS\SerializerBundle\JMSSerializerBundle(),
];
```

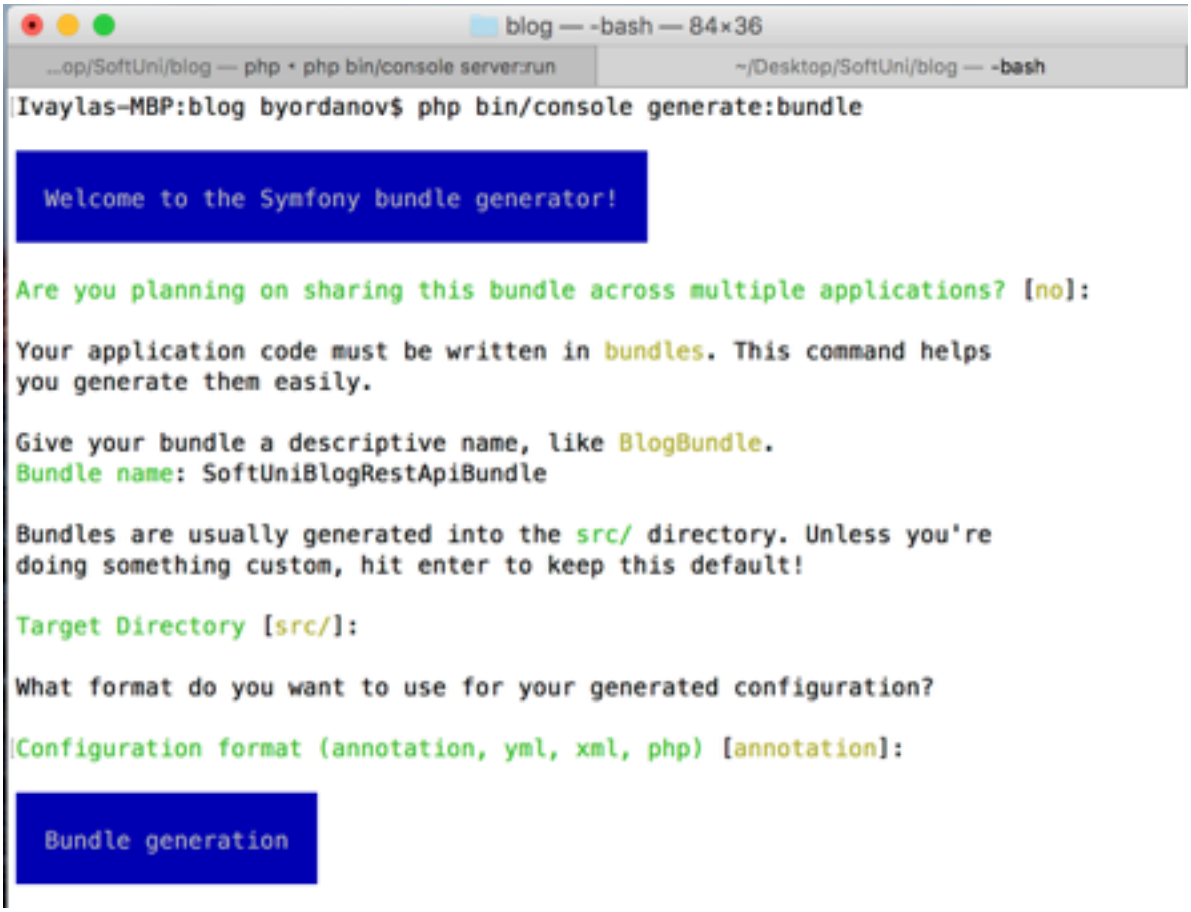
This is it! Now check your project status. If everything is running correctly, then go to the next page.

II. Create new Symfony Bundle for our REST Constrollers

In terminal / console type following command:

php bin/console generate:bundle

When asked for Bundle name, type: SoftUniBlogRestApiBundle



Don't forget that everybundle must be activated after creation/ installation!

```
$bundles = [
    new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
    new Symfony\Bundle\SecurityBundle\SecurityBundle(),
    new Symfony\Bundle\TwigBundle\TwigBundle(),
    new Symfony\Bundle\MonologBundle\MonologBundle(),
    new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
    new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
    new SoftUniBlogBundle\SoftUniBlogBundle(),
    new JMS\SerializerBundle\JMSSerializerBundle(),
    new SoftUniBlogRestApiBundle\SoftUniBlogRestApiBundle(),
];
```

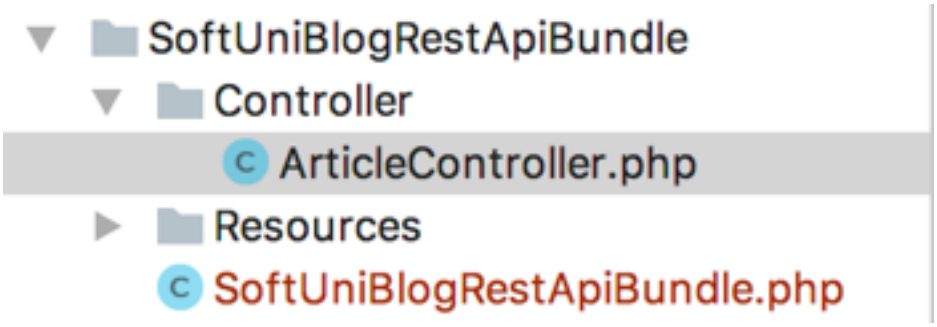
Now to avoid conflict in routes open app/config/routing.yml file.

Modify the **prefix: /api**

```
soft_uni_blog_rest_api:
    resource: "@SoftUniBlogRestApiBundle/Controller/"
    type:     annotation
    prefix:   /api
```

This configuration tells Symfony to set different uri for our new bundle. The prefix can be anything you want. In our case **/api**

III. Create ArticleController in the new bundle.



Create **articlesAction() - get list of articles**

ArticleController.php

```
<?php

namespace SoftUniBlogRestApiBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use SoftUniBlogBundle\Entity\Article;
use Symfony\Component\HttpFoundation\Response;

class ArticleController extends Controller
{
    /**
     * @Route("/articles", name="rest_api_articles")
     */
    public function articlesAction()
    {

        $articles = $this->getDoctrine()->getRepository(Article::class)-
>findAll();

        $serializer = $this->container->get('jms_serializer');
        $json = $serializer->serialize($articles, 'json');

        return new Response($json,
                    Response::HTTP_OK,
                    array('content-type' => 'application/json')
        );

    }
}
```

The @Route annotation defines the URI for our action. Don't forget that we configured a prefix **/api** for our REST Bundle in **app/config/routing.yml**

We may have multiple bundles with the same URIs for their actions but different prefixes!

Get a list of all articles in our database

```
$articles = $this->getDoctrine()->getRepository(Article::class)->findAll();
```

The following line calls jms_serializer service which we have installed before.

```
$serializer = $this->container->get('jms_serializer');
```

Now tell jms_serializer to convert our Doctrine entities to JSON objects.

```
$json = $serializer->serialize($articles, 'json');
```

With the last line we return the Response object to our browser.

```
return new Response($json,
            Response::HTTP_OK,
            array('content-type' => 'application/json')
    );
```

$json - Our Doctrine entities encoded to JSON
Response::HTTP_OK - Response code for successful query
`array('content-type' => 'application/json')` - When returning JSON objects to the browser we need to set the correct content-type.

Check if everything works correctly.

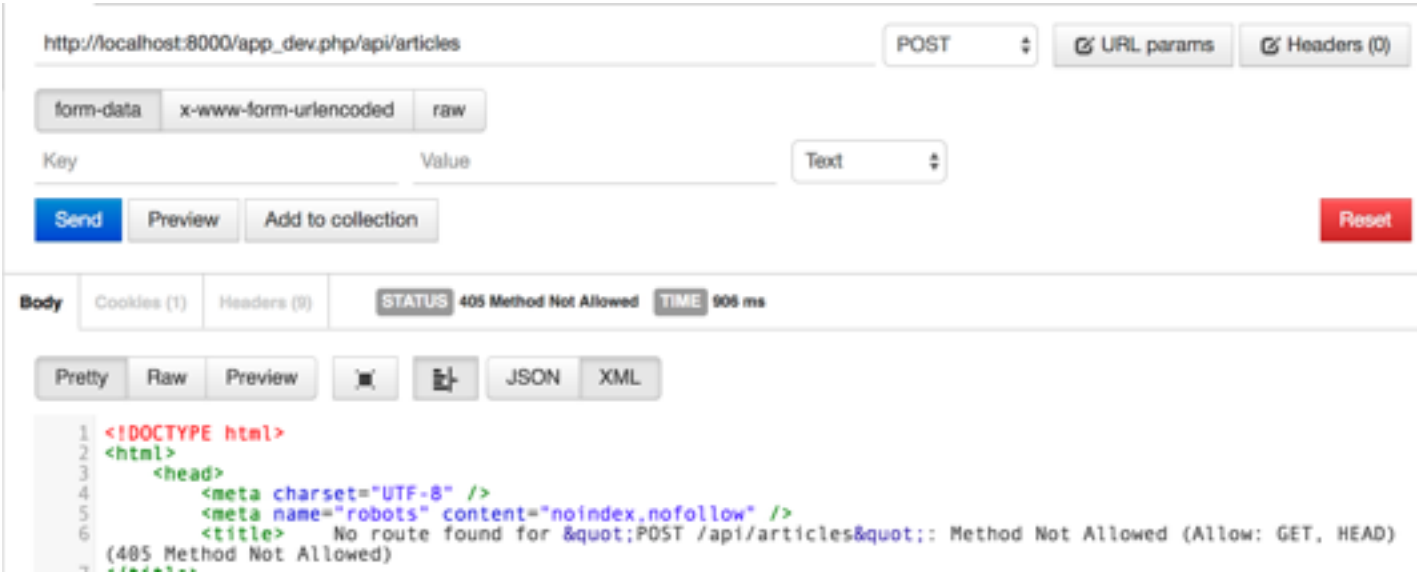
We are still not ready. Unfortunately, wether it's GET, POST or DELETE request, we always get the same results. This should not happen in REST.

Modify the controller by importing a new **@Method annotation** and assigning in to our **articlesAction()**

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
```

```
/**
 * @Route("/articles", name="rest_api_articles")
 * @Method({"GET"})
 */
public function articlesAction()
```

Now try to send POST or DELETE request to the same action and you should see similar result



Continue on the next page…

Create new **articleAction() - get single article**

ArticleController.php

```
<?php

namespace SoftUniBlogRestApiBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use SoftUniBlogBundle\Entity\Article;
use Symfony\Component\HttpFoundation\Response;

class ArticleController extends Controller
{

   /**
     * @Route("/articles/{id}", name="rest_api_article")
     * @Method({"GET"})
     * @param $id article id
     * @return JsonResponse
     */
    public function articleAction($id)
    {
        $article = $this->getDoctrine()->getRepository(Article::class)-
>find($id);

        if(null === $article)   {
            return new Response(json_encode(array('error' => 'resource not
found')),
                Response::HTTP_NOT_FOUND,
                array('content-type' => 'application/json')
            );
        }

        $serializer = $this->container->get('jms_serializer');
        $articleJson = $serializer->serialize($article, 'json');

        return new Response($articleJson,
            Response::HTTP_OK,
            array('content-type' => 'application/json')
        );

    }
```

The code is pretty much the same as previous one with a little difference.

If requested article is not found **Response** object is returned containing error message encoded in **JSON**

The **Response code** for not found resources is **404** or in Symfony **Response::HTTP_NOT_FOUND**

For successful requests response code should be **200 OK .**

In Symfony - **Resonse::HTTP_OK**

Create new action: **createAction() - responsible for creating new articles**

```
/**
 * @Route("/articles/create", name="rest_api_article_create")
 * @Method({"POST"})
 * @param $request Request
 * @return Response
 */
public function createAction(Request $request)
{
    try {
        //process submitted data
        $this->createNewArticle($request);

        return new Response(null,Response::HTTP_CREATED);

    }   catch (\Exception $e)   {

        return new Response(json_encode(['error' => $e->getMessage()]),
            Response::HTTP_BAD_REQUEST,
            array('content-type' => 'application/json')
        );
    }
}
```

Again not many differences.

`@Method({“POST”})` – the action will be executed on POST request

```
/**
 * Creates new article from request parameters and persists it
 * @param Request $request
 * @return Article – persisted article
 */
protected function createNewArticle(Request $request) {
    $article = new Article();
    $parameters = $request->request->all();
    $persistedType = $this->processForm($article, $parameters, 'POST');
    return $persistedType;
}
```

Method **createNewArticle()** creates **new Article() object**, populate array with request parameters and pass the data to our form processor
$parameters = $request->request->all(); - get all request parameters and put them array

The **$this->processForm($article, $parameters, 'POST')** method performs simple validation of the request parameters, creates a Symfony ArticleType form, validate it's data and save it to the database. **$this->proccessForm()** code is listed bellow

$form = $this->createForm(ArticleType::*class*, $article, ['method' => $method]);
The method creates a Symfony form of **ArticleType** and connect the form with **$article** entity and finally sets the form method attribute.

```
/**
 * Processes the form.
 * @param Request $request
 * @return Article
 * @throws \Exception
 */
private function processForm($article, $params, $method = 'PUT') {

    foreach($params as $param => $paramValue)  {
        if(null === $paramValue || 0 === strlen(trim($paramValue)))  {
            throw new \Exception("invalid data: $param");
        }
    }

    if(!array_key_exists('authorId', $params))   {
        throw new \Exception('invalid data: authorId');
    }

    $user = $this->getDoctrine()
        ->getRepository(User::class)
        ->find($params['authorId']);

    if(null === $user)  {
        throw new \Exception('invalid user id');
    }

    $form = $this->createForm(ArticleType::class, $article, ['method'
=> $method]);
    $form->submit($params);

    if ($form->isSubmitted()) {
        $article->setAuthor($user);
        //get entity manager
        $em = $this->getDoctrine()->getManager();
        $em->persist($article);
        $em->flush();
        return $article;
    }
    throw new \Exception('submitted data is invalid');
}
```

$form->submit($params) method submits the parameters with the form. If the form is valid and submitted, our **$article** entity should be populated with correct data.

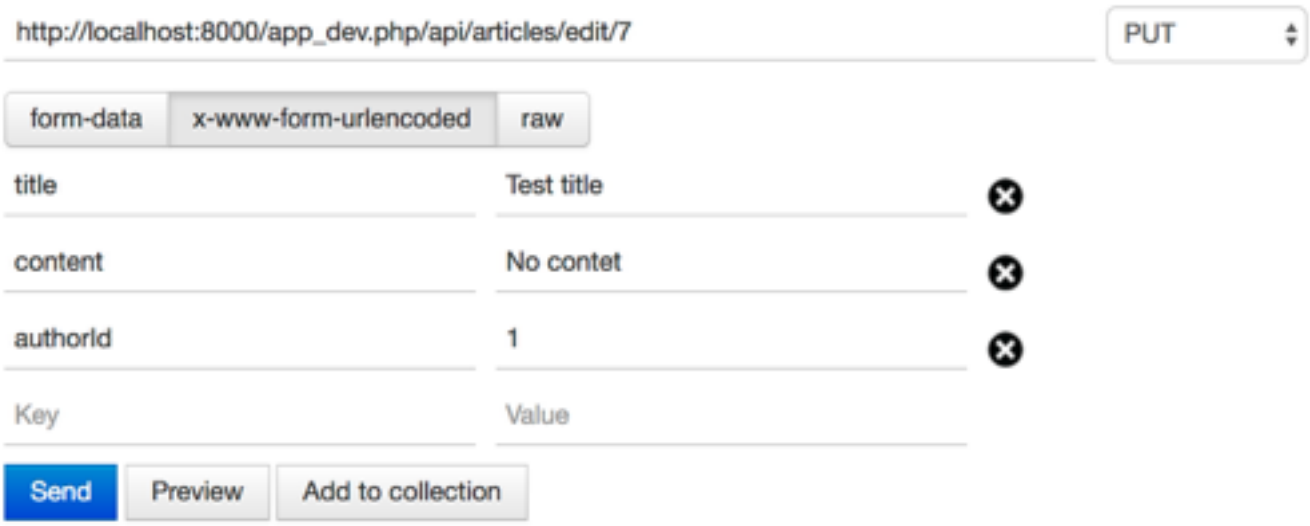
Create new action: **editAction() - edit article**

```
/**
 * @Route("/articles/{id}", name="rest_api_article_edit")
 * @Method({"PUT"})
 * @param $request Request
 * @return Response
 */
public function editAction(Request $request, $id)
{
    try {

        $article = $this->getDoctrine()->getRepository(Article::class)->find($id);
        if(null === $article)   {
            //create new article
            $this->createNewArticle($request);
            $statusCode = Response::HTTP_CREATED;
        }   else    {
            //update existing article
            $this->processForm($article, $request->request->all(), 'PUT');
            $statusCode = Response::HTTP_NO_CONTENT;
        }

        return new Response(null,$statusCode);

    }   catch (\Exception $e)   {

        return new Response(json_encode(['error' => $e->getMessage()]),
            Response::HTTP_BAD_REQUEST,
            array('content-type' => 'application/json')
        );
    }
}
```

Search for the given article.
If not found, new article is created with **$this->createNewArticle($request)**, otherwise the existing one is updated.

Continue on the next page…

Attention! When testing 'PUT' with Postman, make sure the button x-www-form-urlencoded is selected.



The response code for PUT requests should be **204 No Content.**
In Symfony - **Response::HTTP_NO_CONTENT**
For any other error you can return **Response::HTTP_BAD_REQUEST**

Create new **deleteAction() - delete resource**

```php
/**
 * @Route("/articles/{id}", name="rest_api_article_edit")
 * @Method({"DELETE"})
 * @param $request Request
 * @return Response
 */
public function deleteAction(Request $request, $id)
{
    try {

        $article = $this->getDoctrine()->getRepository(Article::class)->find($id);
        if(null === $article)   {
            $statusCode = Response::HTTP_NOT_FOUND;
        }   else    {

            $em = $this->getDoctrine()->getManager();
            $em->remove($article);
            $em->flush();

            $statusCode = Response::HTTP_NO_CONTENT;
        }

        return new Response(null,$statusCode);

    }   catch (\Exception $e)   {

        return new Response(json_encode(['error' => $e->getMessage()]),
            Response::HTTP_BAD_REQUEST,
            array('content-type' => 'application/json')
        );
    }
}
```

Almost the same code as before
@Method({"DELETE"}) annotation tells Symfony to execute the action only on DELETE request.
On success return **response code 204**
In Symfony - **Response::HTTP_NO_CONTENT**

Now we have fully functional REST Api for our blog.
The bad thing now is that we don't have control of which database columns will be displayed.
jms/serializer-bundle is here to help us!

Import following annotations in every Doctrine entity that you wish to use in your REST Api

```php
use JMS\Serializer\Annotation\ExclusionPolicy;
use JMS\Serializer\Annotation\Expose;
```

Add **@ExclusionPolicy annotation** after the **@ORM annotations**

```
/**
 * Article
 *
 * @ORM\Table(name="articles")
 * @ORM\Entity(repositoryClass="SoftUniBlogBundle\Repository\ArticleRepository")
 * @ExclusionPolicy("all")
 */
class Article
```

This simple tells jms/serializer-bundle - don't show any columns from this entity

Now for every column / property that you wish to be visible in your api add **@Expose annotation** as shown on the picture

```
/**
 * @var string
 *
 * @ORM\Column(name="title", type="string", length=255, unique=true)
 * @Expose
 */
private $title;
```

This will display only properties that you need.