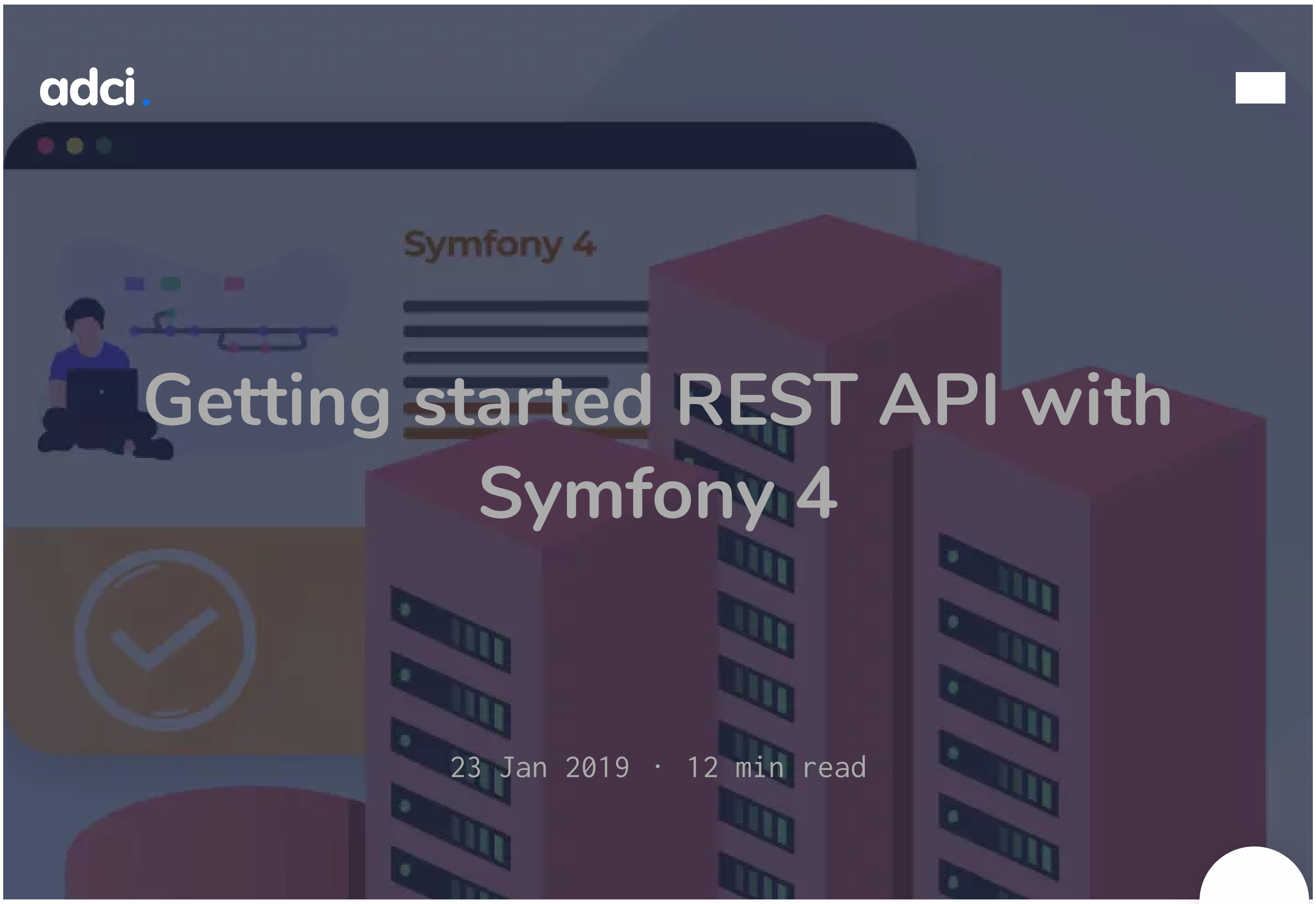
adci.



Getting started REST API with Symfony 4

23 Jan 2019 · 12 min read

The concept of the Internet of things is a trending topic and it has found implementation in numerous aspects. Computers, laptops, mobile phones, and other non-internet-enabled physical devices like cars, or smart grids can interact with each other. To do it they need a standardized interface called API (application programming interface). The advantage of using a unified interface is once you implement such a web service on your server, it can interact with a variety of client devices. In addition, it gives the ability to build a web application with separate back-end and front-end parts. You may use modern JS frameworks to make user interfaces more dynamic and achieve good UX. Many organizations integrate their mobile applications with back-end CRM, ERP, EAM, and other tools to improve their productivity.

We considered how to create API using the **headless Drupal 8** approach in our previous articles, find them in the Useful links block. However, the functionality of a CMS may not have enough flexibility for the projects with complex business logic. In such cases, using a PHP framework may become a more preferable option.

Symfony is one of the most popular Open Source PHP frameworks consisting of separate components. They are used in many large projects, such as Drupal 8 or Laravel. Symfony has an active community and stable support. Also, it has the expandable architecture of reusable components called “bundle”, it extends the functionality of the framework. The latest Symfony version has a lot of improvements, such an automatic configuration of bundles with Symfony Flex and simplified folder structure increase the speed of development.

In this **Symfony 4 tutorial**, we will create a basic server back-end structure for your application using the **REST API** architecture style. We will use a 'FOSRestBundle' bundle as a basis, implement 'get' and 'post' methods to create and show the list of resources respectively. Besides that, we will add the OAuth2 authentication with FOSOAuthServerBundle.

Create a Rest API

Firstly, make sure you have installed PHP 7.1 or a higher version and the Composer package manager to create a new Symfony application. After that, create a new project by executing the following command in the terminal:

```
composer create-project symfony/skeleton rest_api_project
```

We use a basic Symfony skeleton project that is recommended for more barebones applications like microservices and APIs. Symfony 4.x has a new and more simplified directory structure. Here is what it looks like:

The default directory structure of a Symfony 4 project

The config directory contains all bundles configuration files and a list of enabled bundles in the bundles.php file. Symfony 4 will automatically register all bundles after the installation using Symfony flex recipes.
The public folder provides access to the application via the index.php entry point whereas the src folder contains all controllers, custom services, and objects. The var directory contains system logs and cache files. The vendor folder contains all external packages.

Now let’s install some necessary bundles with composer

```
composer require friendsofsymfony/rest-bundle
composer require sensio/framework-extra-bundle
composer require jms/serializer-bundle
composer require symfony/validator
composer require symfony/form
composer require symfony/orm-pack
```

Besides the friendsofsymfony/rest-bundle, we also installed the sensio/framework-extra-bundle. It will help us to make code easier to read by using annotations for defining our routes.

We will use jms/serializer-bundle to serialize and deserialize resources of the application. We do have some validation requirements in our testing entity. Because of this, it’s also necessary to add a validator bundle. The form bundle will help us to handle incoming user data and convert it into a resource entity. We need to install symfony/orm-pack for the integration with Doctrine ORM to connect with a database. Database configuration may be set in the .env file.

```
###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/
# For an SQLite database, use: "sqlite:///%kernel.project_dir%/var/data.db"
# Configure your db driver and server_version in config/packages/doctrine.yaml
DATABASE_URL=mysql://db_user:db_pass@127.0.0.1:3306/db_name
###< doctrine/doctrine-bundle ###
```

As a result, our bundles.php file should look like:

```
<?php
return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['
    FOS\RestBundle\FOSRestBundle::class => ['all' => true],
    JMS\SerializerBundle\JMSSerializerBundle::class => ['all' => true],
    Doctrine\Bundle\DoctrineCacheBundle\DoctrineCacheBundle::class => ['all' =
    Doctrine\Bundle\DoctrineBundle\DoctrineBundle::class => ['all' => true],
    Doctrine\Bundle\MigrationsBundle\DoctrineMigrationsBundle::class => ['
];
```

After we have finished with the installation, let's create a test resource entity. Create a new file called Movie.php inside the src/Entity folder

```
<?php
namespace App\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
/**
 * @ORM\Entity
 * @ORM\Table(name="movie")
 */
class Movie {
  /**
   * @ORM\Column(type="integer")
   * @ORM\Id
   * @ORM\GeneratedValue(strategy="AUTO")
   */
```

```php
    private $id;
    /**
     * @ORM\Column(type="string", length=100)
     * @Assert\NotBlank()
     *
     */
    private $name;
    /**
     * @ORM\Column(type="text")
     * @Assert\NotBlank()
     */
    private $description;
    /**
     * @return mixed
     */
    public function getId()
    {
      return $this->id;
    }
    /**
     * @param mixed $id
     */
    public function setId($id)
    {
      $this->id = $id;
    }
    /**
     * @return mixed
     */
    public function getName()
    {
      return $this->name;
    }
    /**
     * @param mixed $name
     */
    public function setName($name)
    {
      $this->name = $name;
    }
    /**
     * @return mixed
     */
    public function getDescription()
    {
      return $this->description;
    }
    /**
     * @param mixed $description
     */
    public function setDescription($description)
    {
      $this->description = $description;
    }
  }
```

Run the bin/console doctrine:schema: create command in the application directory to create a database structure according to our movie class with Doctrine ORM. We need to create a simple form for the Movie entity inside the src/Form folder to handle and validate a user's request to post a new movie:

```
<?php
namespace App\Form;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\OptionsResolver\OptionsResolver;
use App\Entity\Movie;
class MovieType extends AbstractType
{
  public function buildForm(FormBuilderInterface $builder, array $options)
  {
    $builder
      ->add('name')
      ->add('description')
      ->add('save', SubmitType::class)
    ;
  }
  public function configureOptions(OptionsResolver $resolver)
  {
    $resolver->setDefaults(array(
      'data_class' => Movie::class,
            'csrf_protection' => false
        ));
    }
}
```

At the next step, set the following configuration for the fos_rest bundle

```
# Read the documentation: https://symfony.com/doc/master/bundles/FOSRestBundle
fos_rest:
    routing_loader:
        default_format: json
        include_format: false
    body_listener: true
    format_listener:
          rules:
              - { path: '^/', priorities: ['json'], fallback_format: json, pre
    param_fetcher_listener: true
    access_denied_listener:
        json: true
    view:
        view_response_listener: 'force'
        formats:
            json: true
```

and add this code to the bottom of services the .yaml config file

```
sensio_framework_extra.view.listener:
    alias: Sensio\Bundle\FrameworkExtraBundle\EventListener\TemplateListener
```

to resolve the issue when FOS Rest Bundle is incompatible with the Sensio Framework Extra Bundle (https://github.com/FriendsOfSymfony/FOSRestBundle/issues/1768).

Now it's time to create a controller where we will add two methods to create a new movie and get the list of all the movies. As you understand, it should be placed inside the src/Controller folder.

```
<?php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use FOS\RestBundle\Controller\FOSRestController;
use FOS\RestBundle\Controller\Annotations as Rest;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use App\Entity\Movie;
use App\Form\MovieType;
/**
 * Movie controller.
 * @Route("/api", name="api_")
 */
class MovieController extends FOSRestController
{
  /**
   * Lists all Movies.
   * @Rest\Get("/movies")
   *
   * @return Response
   */
  public function getMovieAction()
  {
    $repository = $this->getDoctrine()->getRepository(Movie::class);
    $movies = $repository->findall();
    return $this->handleView($this->view($movies));
  }
  /**
   * Create Movie.
   * @Rest\Post("/movie")
   *
   * @return Response
   */
  public function postMovieAction(Request $request)
  {
    $movie = new Movie();
    $form = $this->createForm(MovieType::class, $movie);
```

```
        $data = json_decode($request->getContent(), true);
        $form->submit($data);
        if ($form->isSubmitted() && $form->isValid()) {
          $em = $this->getDoctrine()->getManager();
          $em->persist($movie);
          $em->flush();
          return $this->handleView($this->view(['status' => 'ok'], Response::HTTP_
        }
        return $this->handleView($this->view($form->getErrors()));
      }
    }
```

Here we defined two routes, GET '/api/movies' will return to us the list of all the movies. The POST '/api/movie' request will run data validation with a Symfony form and create a new movie resource if data is valid. Ok, we have finished with the basic structure, let's check how it works. First thing, try to create a couple of new resources by sending data in a JSON format.

The example of the post API request

Then we can receive them back using the GET request.

The example of the GET API request

OAuth2 authorization

There are several authorization methods for rest APIs and OAuth2 is one of the most popular. It allows using auth details from an external provider like Google or Facebook to identify yourself. In our application, we will use FOSUserBundle as a user provider, install this bundle with composer, then install FOSOAuthServerBundle as it is shown here:

```
composer require friendsofsymfony/user-bundle
composer require friendsofsymfony/oauth-server-bundle
```

Now you need to create necessary classes for a client and an access token according to the FOSOAuthServerBundle documentation and specify them in the fos_auth_server.yaml config file

```
fos_oauth_server:
    db_driver: orm
    client_class:        App\Entity\Client
    access_token_class:  App\Entity\AccessToken
    refresh_token_class: App\Entity\RefreshToken
    auth_code_class:     App\Entity\AuthCode
    service:
        user_provider: fos_user.user_provider.username
        options:
            access_token_lifetime: 86400
            refresh_token_lifetime: 1209600
            auth_code_lifetime: 30
```

Import routes from fos_auth_server bundle into config/routes.yaml

```
#index:
#    path: /
#    controller: App\Controller\DefaultController::index
fos_oauth_server_token:
    resource: "@FOSOAuthServerBundle/Resources/config/routing/token.xml"

fos_oauth_server_authorize:
    resource: "@FOSOAuthServerBundle/Resources/config/routing/authorize.xml"
```

Create a user class for FOSUserBundle

```
<?php
namespace App\Entity;
use FOS\UserBundle\Model\User as BaseUser;
use Doctrine\ORM\Mapping as ORM;
/**
 * @ORM\Entity
 * @ORM\Table(name="fos_user")
 */
class User extends BaseUser
{
  /**
   * @ORM\Id
   * @ORM\Column(type="integer")
   * @ORM\GeneratedValue(strategy="AUTO")
   */
  protected $id;
  public function __construct()
  {
    parent::__construct();
  }
}
```

and add a config file for UserBundle

```
fos_user:
    db_driver: orm
    firewall_name: main
    user_class: App\Entity\User
    from_email:
      address:        resetting@example.com
      sender_name:    Demo Resetting
```

This code should be in the security.yaml file

```
security:
```

```
encoders:
    FOS\UserBundle\Model\UserInterface: bcrypt

role_hierarchy:
    ROLE_ADMIN:       ROLE_USER
    ROLE_SUPER_ADMIN: ROLE_ADMIN

providers:
    fos_userbundle:
        id: fos_user.user_provider.username

firewalls:
    oauth_token:
        pattern:    ^/oauth/v2/token
        security:   false
    oauth_authorize:
        pattern:    ^/oauth/v2/auth
        form_login:
            provider: fos_userbundle
            check_path: /oauth/v2/auth_login_check
            login_path: /oauth/v2/auth_login
            use_referer: true
    api:
        pattern:    ^/api
        fos_oauth:  true
        stateless:  true
        anonymous:  false
    main:
     pattern: ^/
     form_login:
         provider: fos_userbundle
         csrf_token_generator: security.csrf.token_manager
     anonymous: true

access_control:
    - { path: ^/api, roles: [ IS_AUTHENTICATED_FULLY ] }
    - { path: ^/createClient, roles: [ IS_AUTHENTICATED_ANONYMOUSLY ] }
```

There we have made /api routers available for authenticated users only by adding it in the access_control block.

The example of the unauthorized request

After we finish the bundles' configuration, we need to create an OAuth client and User to generate access tokens. To do this, let's add a test controller

```php
<?php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Request;
use FOS\RestBundle\Controller\FOSRestController;
use FOS\RestBundle\View\View;
use FOS\RestBundle\Controller\Annotations as FOSRest;
use Symfony\Component\HttpFoundation\Response;
use FOS\OAuthServerBundle\Model\ClientManagerInterface;
class SecurityController extends FOSRestController
{
  private $client_manager;
  public function __construct(ClientManagerInterface $client_manager)
  {
    $this->client_manager = $client_manager;
  }
  /**
   * Create Client.
   * @FOSRest\Post("/createClient")
   *
   * @return Response
   */
  public function AuthenticationAction(Request $request)
  {
    $data = json_decode($request->getContent(), true);
    if (empty($data['redirect-uri']) || empty($data['grant-type'])) {
      return $this->handleView($this->view($data));
    }
    $clientManager = $this->client_manager;
    $client = $clientManager->createClient();
    $client->setRedirectUris([$data['redirect-uri']]);
    $client->setAllowedGrantTypes([$data['grant-type']]);
    $clientManager->updateClient($client);
    $rows = [
      'client_id' => $client->getPublicId(), 'client_secret' => $client->getSe
```

```
        ];
        return $this->handleView($this->view($rows));
      }
    }
```

In your request to /createClient, you need to specify the redirect URIs you want to use, and the grant types you want to allow this client to use.

The request to create a new API client

A test user may be created with the console command bin/console fos:user:create test_user.

The command to create a new test user

Now we can get the access token, by making a POST request to the OAuth2 bundle route /oauth/v2/token.

Request to get a new access token

If everything is going right, we should receive the access token that we should send to the API request in headers, as follows

Authorization: Bearer NzQyNDg4YTJmMDFlYTgzMGE0ZWFjYWE0ZTgwMzhlN2YzODAxZjUyZGRlZjQyNzMyY2E5MzY1MDk3OTZhYzJhOQ

That's all for now, our API Platform implementation completed. We have added two endpoints to create and show the list of resources and added OAuth2 authorization. If you have missed something, you can check the code in the repository.

Have fun building your own application!

Useful links

RESTful Web Services in Drupal 8 quick start guide

How to create a headless Drupal site

Web development Symfony

Dmitry Romanovsky · Web Developer

Dmitry is famous for finding beautiful and optimized solutions to solve a client's problem. As a hobby Dmitry studies machine learning and reads books. Also, in his spare time he watches TV series, listens to music, and plays video games sometimes