Rest API mit Symfony 6.4

Verfasser: Hong Le Nguyen, last update 08.24

Die komplette Implementierung ist bei Git zu finden:
https://github.com/hong1234/sym64_API
https://github.com/hong1234/sym64_orm32_API
Ein React UI App für den Rest-Api : https://github.com/hong1234/BookReviewReactUI_v2

API Anwendungfälle ---

Nehmen wir an die folgenden Anwendungfälle sind duch ein Rest-Api zu unterstützen
- Bücher zeigen und nach Buchtitel suchen
- Ein bestimmtes Buch zeigen und
- Leser haben Möglichkeit Bewertungen zu dem Buch zugeben.

Rest API Design ---

Domain Modell: book – review (1:n relation)

Method-Url zu Endpoints (services / operations on the resources)

```
GET     /books                      // get all books
GET     /books/{bookId}             // get book with id=bookId
GET     /books/search?title=Spring  // all books  with 'Spring' = substring of title
POST    /books                      // create a book
PUT     /books/{bookId}             // update the book id=bookId
DELETE  /books/{bookId}             // delete  the book id=bookId
GET     /reviews/{bookId}           // get all reviews of the book id=bookId
POST    /reviews/{bookId}           // add a review to book id=bookId
DELETE  /reviews/{reviewId}         // delete the review id=reviewId
```

Implementierung mit Symfony 6 ---

routes config --

```
// config/routes.yaml
controllers:
   resource:
      path: ../src/Controller/
      namespace: App\Controller
   type: attribute
```

services config --

```
// config/services.yaml
services:
   _defaults:
      autowire: true      # Automatically injects dependencies in your services.
      autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.
```

autoloader config --

```
// composer.json
"autoload": {
      "psr-4": { "App\\": "src/" }
 },
```

database connect and CORS config --

```
// .env
DATABASE_URL=sqlite:///%kernel.project_dir%/var/app.db
CORS_ALLOW_ORIGIN=^https?://(localhost|127\.0\.0\.1)(:[0-9]+)?$
```

Database Access using Doctrine DBAL

```
<?php
namespace App\Dao;
use Doctrine\DBAL\Connection;
class BaseDao {
   private $conn;
   function __construct(Connection $connection) {
      $this->conn = $connection;
   }
   public function doQuery($sql, $parr=[]) {
      $result = $this->conn->executeQuery($sql, $parr);     // ->fetchAllAssociative();
      return $result;
   }
   public function doSQL($sql, $parr=[]) {
      $rowCount = $this->conn->executeStatement($sql, $parr);  //  returns the affected rows count
      return $rowCount;
   }
}
```

```
<?php
namespace App\Dao;
class BookDao extends BaseDao {
   public function getBook(iterable $params=[]) {
      $sql = "SELECT * FROM books WHERE id = :id";
      return $this->doQuery($sql, $params);
   }
   public function bookInsert(iterable $params=[]){
      $sql = "INSERT INTO books SET title = :title, content = :content, created_on = NOW()";
      return $this->doSQL($sql, $params);
   }
```

injecting BookDao into Controller or Service class

```
<?php
namespace App\Controller;
class BookController extends AbstractController {
   private $bookDao;
   public function __construct(BookDao $bDao, BookService $bs) {
      $this->bookDao = $bDao;
      ..
   }
```

```
<?php
namespace App\Service;
class BookService {
   private $bookDao;
   public function __construct(BookDao $bookDao) {
      $this->bookDao = $bookDao;
   }
```

Datenbank Abfragen mit Doctrine QueryBuilder

Verfasser: Hong Le Nguyen, last update 08.24

Nehmen wir an,
wir wollen alle Objekte vom Anbieter $devId mit Angabe über Anzahl der Anfragen an jeweiligen Objekt in der Zeitraum $startdate bis $enddate holen.

Mit QueryBuilder lässt man Query- Befehl mit Parametern bequem formulieren und anschließend Daten abfragen.

```
$em = $this->getDoctrine()->getManager();

// sub query
    $o_sum_requests = $em
        ->createQueryBuilder()
        ->select('COUNT(rq.id)')
        ->from('Nbk:Request', 'rq')
        ->where('rq.object = o.id')
        ->andWhere('rq.status = 2')
        ->andWhere("DATEFORMAT(rq.inserteddate, '%Y-%m-%d') > '$startdate'")
        ->andWhere("DATEFORMAT(rq.inserteddate, '%Y-%m-%d') < '$enddate'");

// main query
    $qb = $em
        ->createQueryBuilder()
        ->select('o.id')
        ->addSelect('(' . $o_sum_requests->getDQL() . ') requests')
        ->addSelect('o.name')
        ->addSelect("DATEFORMAT(o.inserteddate, '%d.%m.%Y') as o_date")
        ->addSelect('o.status')
        ->addSelect('o.sold')

        ->from('Nbk:Object', 'o')
        ->leftJoin('o.developer', 'd')

        ->where('d.id = :devId')
        ->setParameter('devId', $devId)
       ;

// main query extension

    if ($status != 'all') {
        $qb
            ->andWhere('o.status = :Status')
            ->setParameter('Status', $status);
    }

// final query
    $query = $qb->getQuery();

// get result
    $os = $query->getResult();
```

Securing a Symfony Rest API

Verfasser: Hong Le Nguyen, last update 10.24

1)
Loading the User using Entity User Provider (Symfony Security)

The User class --

Permissions in Symfony are always linked to a user object. If you need to secure (parts of) your application, you need to create a user class. This is a class that implements UserInterface. This is often a Doctrine entity, but you can also use a dedicated Security user class.

The easiest way to generate a user class is using the make:user command from the MakerBundle:

```
php bin/console make:user
```

Do you want to store user data in the database via Doctrine ?: > yes
Enter a property name that will be the unique name for the user (e.g. email, username, uuid): > username
Does this app need to hash/check user passwords ?: > yes
// creating code …
created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: config/packages/security.yaml

```
// src/Entity/User.php
#[ORM\Entity(repositoryClass: UserRepository::class)]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{ … }

// src/Repository/UserRepository.php
class UserRepository extends ServiceEntityRepository implements PasswordUpgraderInterface
{ … }
```

Note:
If your user class is a Doctrine entity and you hash user passwords, the Doctrine repository class related to the user class must implement the PasswordUpgraderInterface.

The User Provider --

Symfony comes with several built-in user providers:
Entity User Provider: Loads users from a database using Doctrine

Besides creating the entity, the make:user command also adds config for a user provider in your security configuration:

```
# config/packages/security.yaml
security:
    # ...
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username # or email
```

This user provider knows how to (re)load users from a storage (e.g. a database) based on a "user identifier" (e.g. the user's email address or username). The configuration above uses Doctrine to load the User entity using the email property as "user identifier".

Registering the User --

Hashing Password

make sure your User class implements the PasswordAuthenticatedUserInterface

```
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
```

Then, configure which password hasher should be used for this class.

```
# config/packages/security.yaml
security:
    # ...
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

User Register implement

Now that Symfony knows how you want to hash the passwords, you can use the UserPasswordHasherInterface service to do this before saving your users to the database:

```
// src/Controller/RegistrationController.php
namespace App\Controller;
// ...
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class RegistrationController extends AbstractController {

    #[Route('/api/register', name: 'register', methods: ['POST'])]
    public function register(UserPasswordHasherInterface $passwordHasher): Response {
        // ... e.g. get the user data from a registration form
        $user = new User(...);
        $plaintextPassword = ...;

        // hash the password (based on the security.yaml config for the $user class)
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
        $user->setPassword($hashedPassword);
        // ...
    }
}
```

You can also manually hash a password by running:

```
php bin/console security:hash-password
```

2)
Login and creating Token using "JSON Login"

The JSON based authentication process --

A client (e.g. the front-end) makes a POST request with the Content-Type: application/json header to "/api/login" with username (even if your identifier is actually an email) and password keys:

```
{
   "username": "dunglas@example.com",
   "password": "MyPassword"
}
```

The security system intercepts the request, checks the user's submitted credentials and authenticates the user. If the credentials are incorrect, an HTTP 401 Unauthorized JSON response is returned, otherwise your controller is run.

Your controller creates the correct response:

```
{
   "user": "dunglas@example.com",
   "token": "45be42..."
}
```

JSON Login (Symfony Security) --

Some applications provide an API that is secured using tokens.
The JSON login authenticator helps you create these tokens based on a username (or email) and password.

Enable the authenticator using the json_login setting:

```
# config/packages/security.yaml
security:
   # ...
   firewalls:
      login:
         pattern: ^/api/login
         stateless: true
         json_login:
            # username_path: email
            check_path: /api/login
            success_handler:
            failure_handler:
```

The pattern option defines the URL pattern that matches the firewall.
The stateless option indicates that the firewall does not use sessions or cookies.
The check_path option defines the URL or route name that will handle the login request.
The success_handler and failure_handler options define the services that will handle the login success and failure events.

The login controller
*This login controller will be called **after** the authenticator successfully authenticates the user*. You can get the authenticated user, generate a token (or whatever you need to return) and return the JSON response.

```
// src/Controller/ApiLoginController.php
namespace App\Controller;
// ..
use App\Entity\User;
use Symfony\Component\Security\Http\Attribute\CurrentUser;
use Lexik\Bundle\JWTAuthenticationBundle\Services\JWTTokenManagerInterface;

class ApiLoginController extends AbstractController {

   #[Route('/api/login', name: 'api_login', methods: ['POST'])]
   public function login(CurrentUser $user, JWTTokenManagerInterface $JWTManager): Response {
```

```
        if (null === $user) {
            return $this->json([
                'message' => 'missing credentials',
            ], Response::HTTP_UNAUTHORIZED);
        }

        // somehow create an API token for $user
        $token = $JWTManager->create($user);

        return $this->json([
            'user'   => $user->getUserIdentifier()
            'token' => $token
        ]);
    }
```

3)
Using the JWT authentication to secure symfony Rest API

So far you have implemented login and (JWT) token generation. Now you haveto implement a JWT authentication system for your symfony rest API. Follow these steps:

Install the LexikJWTAuthenticationBundle using Composer:

```
composer require lexik/jwt-authentication-bundle
```

Generate keypairs for encoding JWTs

```
php bin/console lexik:jwt:generate-keypair
```

Configure SSL keys and JWT passphrase in your .env file

```
###> lexik/jwt-authentication-bundle ###
JWT_SECRET_KEY=%kernel.project_dir%/config/jwt/private.pem
JWT_PUBLIC_KEY=%kernel.project_dir%/config/jwt/public.pem
JWT_PASSPHRASE=26ada95f00d34637585b3f70ba5cd52fe6517f615301a20e5bc2f659ea7413dc
###< lexik/jwt-authentication-bundle ###
```

Configure LexikJWTAuthenticationBundle in config/packages/lexik_jwt_authentication.yaml

```
lexik_jwt_authentication:
  secret_key: '%env(resolve:JWT_SECRET_KEY)%'
  public_key: '%env(resolve:JWT_PUBLIC_KEY)%'
  pass_phrase: '%env(JWT_PASSPHRASE)%'
  token_ttl: 3600
```

Modify your security configuration in config/packages/security.yaml

The API firewall will use the jwt authenticator, which will validate the JWT token and grant access to the protected resources.

```
// config/packages/security.yaml
…
firewalls:
  login:
    pattern: ^/api/login
    stateless: true
    json_login:
      # username_path: email
      check_path: /api/login
      success_handler: lexik_jwt_authentication.handler.authentication_success
      failure_handler: lexik_jwt_authentication.handler.authentication_failure
```

```
api:
  pattern:   ^/api
  stateless: true
  jwt: ~
```

The jwt option enables the JWT authenticator.

Finally, we need to define some access control rules to restrict access to the endpoints based on the user roles. We will use the PUBLIC_ACCESS role to allow anyone to access the login endpoint, and the IS_AUTHENTICATED_FULLY role to require a valid JWT token to access the API endpoints.

To define the access control rules, add the following lines under the access_control key:

```
access_control:
  - { path: ^/api/register, roles: PUBLIC_ACCESS }
  - { path: ^/api/login, roles: PUBLIC_ACCESS }
  - { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
```

The path option defines the URL pattern that matches the rule. The roles option defines the required roles to access the path.

The complete code can be found on Git:

https://github.com/hong1234/sym64_API

Install -----

Creating a New Symfony Project

```
symfony new sym64_rest_api --version="6.4.*"
```

Installing the Required Packages

```
composer require symfony/orm-pack
composer require symfony/maker-bundle --dev
composer require lexik/jwt-authentication-bundle
```

Creating the User Entity

```
php bin/console make:user
```

Configure your database and run the following commands to create the users table:

```
php bin/console doctrine:database:create
php bin/console make:migration
doctrine:migrations:migrate
```

The final step in the database process is to create a user within the database. You need to first hash your password using this command:

```
php bin/console security:hash-password
```

Copy the "Password hash" and paste it into the password column of the user. Set the roles column as [].

Links ----

https://symfony.com/doc/current/security.html#json-login
https://symfony.com/doc/current/reference/configuration/security.html#reference-security-firewall-json-login
https://symfony.com/doc/6.4/reference/configuration/security.html#json-login-authentication
https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords

https://dev.to/daniyaljavani/how-to-add-jwt-login-to-a-symfony-6-project-3elh
https://github.com/lexik/LexikJWTAuthenticationBundle/blob/3.x/Resources/doc/index.rst#configuration

Arbeit mit Bundle in Symfony 5

Verfasser: Hong Le Nguyen, last update 08.24

Bundles sind wiederverwendbare Plugins, die Funktionen für Symfony-Anwendung bereitstellen.
Wir nehmen an, der Bundle “SocialBundle” soll unter Verzeichnis “project/src/Acme” angelegt werden. Die folgenden Kenntnisse helfen uns bei der Integration des Bundle in Symfony-Anwendung.

1-Bundle registration:

```
// config/bundles.php
return [
    // ...
    Acme\SocialBundle\AcmeSocialBundle::class => ['all' => true],
];
```

2- Autoloader für Acme namespace definition

```
//composer.json
{
    ...
    "autoload": {
        "psr-4": {
            ...
            "Acme\\": "src/Acme"
        }
    },
    ...
}
```

3- Routes configuration

```
//config/routes/acme_social.yaml
acme_controllers:
    resource: ../../src/Acme/SocialBundle/Controller/
    type: annotation
```

4- Loading configurations(services.xml, parameters.yml) inside a bundle

```
// src/Acme/SocialBundle/DependencyInjection/AcmeSocialExtension.php
namespace Acme\SocialBundle\DependencyInjection;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Extension\Extension;

class AcmeSocialExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new XmlFileLoader( $container, new FileLocator(__DIR__.'/../Resources/config'));
        $loader->load('services.xml');
        $loader = new YamlFileLoader( $container, new FileLocator(__DIR__.'/../Resources/config'));
        $loader->load('parameters.yml');
    }
}
```

5- Loading values of parameters from outside (bundle configuration file)

```
//config/packages/acme_social.yaml
acme_social:
   twitter:
      client_id: 123
      client_secret: abc

//src/Acme/SocialBundle/Resources/config/parameters.yml
parameters:
   param_one: xxx
   param_two: yyy
```

The Configuration class to handle the sample configuration looks like:

```
// src/Acme/SocialBundle/DependencyInjection/Configuration.php
namespace Acme\SocialBundle\DependencyInjection;
use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
   public function getConfigTreeBuilder()
   {
      $treeBuilder = new TreeBuilder('acme_social');
      $treeBuilder->getRootNode()
         ->children()
            ->arrayNode('twitter')
               ->children()
                  ->integerNode('client_id')->end()
                  ->scalarNode('client_secret')->end()
               ->end()
            ->end() // twitter
         →end() ;
      return $treeBuilder;
   }
}

// src/Acme/SocialBundle/DependencyInjection/AcmeSocialExtension.php
class AcmeSocialExtension extends Extension
{
   public function load(array $configs, ContainerBuilder $container)
   { ...
      $loader->load('parameters.yml');
      $configuration = new Configuration();
      $config = $this->processConfiguration($configuration, $configs);
      // you now have these 2 config keys
      $container->setParameter('param_one', $config['twitter']['client_id']);
      $container->setParameter('param_two', $config['twitter']['client_secret']);
   }
}
```

6- Manipulate other service definitions that have been registered with the service container using Compiler Passes

Nehmen wir an ein Service-Definition “category_menu “ ist gesetzt von anderen Bundle. Wir möchten diesen Definition mit neuen Wert von Bundle SocialBundle aus überschreiben.

```
//src/Acme/SocialBundle/DependencyInjection/Compiler/OverrideServiceCompilerPass.php

namespace Acme\SocialBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class OverrideServiceCompilerPass implements CompilerPassInterface {

   public function process(ContainerBuilder $container)
  {
     // Override the core module 'category_menu' service
      $container->removeDefinition('category_menu');
      $container->setDefinition('category_menu',
                                        $container->getDefinition('acme_social.category_menu'));
   }
}

//src/Acme/SocialBundle/AcmeSocialBundle.php

namespace Acme\SocialBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Acme\SocialBundle\DependencyInjection\Compiler\OverrideServiceCompilerPass;

class AcmeSocialBundle extends Bundle {
   public function build(ContainerBuilder $container)
   {
      parent::build($container);
      $container->addCompilerPass(new OverrideServiceCompilerPass());
   }
}
```