

# Server-side Web Development

## Unit 13. Web services and security.

Fidel Oltra, Ricardo Sánchez



IES Jaume II El Just  
Tavernes de la Valldigna  
Departament d'Informàtica  
Curs 2024-25

## Index

<b>1</b>	<b>Web services with Symfony</b>	<b>2</b>
<b>2</b>	<b>Building the API REST manually</b>	<b>2</b>
2.1	Installing required bundles . . . . .	2
2.2	Transforming objects to/from JSON . . . . .	3
2.3	Making the services . . . . .	4
2.3.1	Getting all the contacts (GET) . . . . .	5
2.3.2	Getting one contact (GET) . . . . .	8
2.3.3	Posting one contact (POST) . . . . .	10
2.3.4	Modifying a contact (PUT) . . . . .	12
2.3.5	Removing a contact (DELETE) . . . . .	14
2.4	Validation . . . . .	16
<b>3</b>	<b>API REST with API Platform</b>	<b>19</b>
3.1	Standards used by API Platform . . . . .	19
3.2	Building the API REST with API Platform . . . . .	22
3.3	Modifying the API . . . . .	30
3.4	Serialization . . . . .	32
3.5	Documenting the API . . . . .	34
3.6	Subresources . . . . .	35
3.7	Filters . . . . .	38
3.8	Validation . . . . .	40
<b>4</b>	<b>Security</b>	<b>42</b>
4.1	The <i>security.yaml</i> file . . . . .	42
4.2	The User . . . . .	43
4.2.1	Hashing Passwords . . . . .	45
4.3	Token based authentication . . . . .	45
4.4	Configuring the API . . . . .	46
4.5	Registering users . . . . .	48
4.6	Access control . . . . .	53
4.7	Usage . . . . .	56

## 1 Web services with Symfony

We have already work with REST API's in unit 8. Just remember the key concepts:

### Rest services:

METHOD	FUNCTION
GET	Retrieve information from the service
POST	Create an new resource
PUT	Update a particular resource
DELETE	Remove a particular resource

[HTTP response status codes](#)

[JSON syntax](#)

[JSON data types](#)

## 2 Building the API REST manually

In this example we will make a REST web service based on our Contacts application. It follows the last guided practice.

In the databases unit we made entity classes and the database access functionalities. These parts will remain the same, the only thing we have to do is to add API REST functionality.

Before doing this point, it's recommended to make a new branch, named, for instance, 'manual-rest'.

### 2.1 Installing required bundles

```
composer config extra.symfony.allow-contrib true
composer require jms/serializer-bundle
composer require friendsofsymfony/rest-bundle
```

The first command is to allow contrib bundles. We will use the `friendsofsymfony/rest-bundle` bundle to specify the REST methods as annotations. The `serializer-bundle` is a dependency for the `rest-bundle`.

## 2.2 Transforming objects to/from JSON

As we need the objects we send and get via the API to be encoded as JSON, we will do this task in the entity classes.

First, we will encode a contact object as an array. Go to the Contact class and add the next method:

```
public function toArray(): array
{
    $phoneList = [];
    foreach ($this->phones as $phone) {
        $phoneList[] = [
            'number' => $phone->getNumber(),
            'type' => $phone->getType(),
        ];
    }
    $contactArray = [
        'id' => $this->id,
        'title' => $this->title,
        'name' => $this->name,
        'surname' => $this->surname,
        'email' => $this->email,
        'birthdate' => $this->birthdate->format('Y-m-d'),
        'phones' => $phoneList,
    ];
    return $contactArray;
}
```

As you can see, this function transforms a contact object in an array that we will transform later.

Although you can do this task with the [Serializer component](#) of Symfony, this way is more comprehensive and we have more control over the process.

We also need to transform a JSON encoded object into a contact object, with the next method:

```
public function fromJson($content): void
{
    $content = json_decode($content, true);
    $this->title = $content['title'];
    $this->name = $content['name'];
    $this->surname = $content['surname'];
    $this->email = $content['email'];
    $date = DateTime::createFromFormat('Y-m-d', $content['birthdate']);
}
```

```
if($date){
    $this->birthdate = $date;
} else {
    $this->birthdate = null;
}
}
```

We use the `json_decode` function to transform the JSON content to an array and then assign the data values to the object. Note that here we aren't inserting or updating the phones. You can implement that if you want.

An important thing is that, if some key is incorrect or is not present, the system will launch an error. However, we will manage it in the controller class.

## 2.3 Making the services

First, make a new controller for the contact api:

```
symfony console make:controller ApiContact
```

We don't need the template created automatically, so you can delete the `api_contact` folder with the Twig file inside.

Modify the `ApiContactController` class:

```
<?php

namespace App\Controller;

use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;
use FOS\RestBundle\Controller\Annotations as Rest;
use App\Entity>Contact;

#[Route('/api/contact')]
class ApiContactController extends AbstractController
{
```

```
}
```

The class has a `Route` annotation, which means that any route we specify inside will have that prefix (in this case, all the routes of the internal methods will have the prefix `/api/contact`).

### 2.3.1 Getting all the contacts (GET)

To return all the contacts, we first must to fetch them from the database, add them to an array and return it as a `JsonResponse` with the code 200:

```
#[Rest\Get('/', name: 'contact_api_list')]
public function contactApiList(EntityManagerInterface $entityManager):
    JsonResponse
{
    $contacts = $entityManager->getRepository(Contact::class)->findAll();
    $contactsList = [];

    if (count($contacts) > 0) {
        foreach($contacts as $contact) {
            $contactsList[] = $contact->toArray();
        }
        $response = [
            'ok' => true,
            'contacts' => $contactsList,
        ];
    } else {
        $response = [
            'ok' => false,
            'error' => 'No contacts found',
        ];
    }

    return new JsonResponse($response, 200);
}
```

Note how we are specifying the route with an attribute:

```
#[Rest\Get('/', name: 'api_contact_list')]
```

In this attribute, the path / is added to the class path /api/contact.

We also send with the response an *ok* code to know if the fetching of the data has been successful and an error message if not.

We can check the route simply writing the URL in a browser. You can also use Postman to do that. :

JSON

Dades sense processar

Capçaleres

Desa

Copia

Redueix-ho tot

Amplia-ho tot

Filtra JSON

ok:

true

▼

contacts:

▼

0:

id:

9

title:

"Mr."

name:

"Loryyy"

surname:

"Grimes"

email:

"logrimes@mail.com"

birthdate:

"2018-02-08"

▼

phones:

▼

0:

number:

"664444557"

type:

"Work"

▼

1:

number:

"667889888"

type:

"Mobile"

▼

1:

id:

11

title:

"Mr."

name:

"Mike"

surname:

"Molina"

email:

"molina@mail.com"

birthdate:

"1975-10-21"

▼

phones:

▼

0:

number:

"295667788"

type:

"Landline"

▼

1:

number:

"666557744"

type:

"Mobile"

**Figure 1:** All the contacts in Firefox



### 2.3.2 Getting one contact (GET)

This is similar to the previous request, but adding the id of the requested contact to the route.

```
#[Rest\Get('/{id<\d+>}', name: 'single_contact_api')]
public function index(EntityManagerInterface $entityManager, $id=''):
    JsonResponse
{
    $contact = $entityManager->getRepository(Contact::class)->find($id);
    if ($contact) {
        $contactArray = $contact->toArray();
        $response = [
            'ok' => true,
            'contact' => $contactArray,
        ];
        return new JsonResponse($response, 200);
    } else {
        $response = [
            'ok' => false,
            'error' => 'No contact found with id '.$id,
        ];
        return new JsonResponse($response, 404);
    }
}
```

JSON	Dades sense processar	Capçaleres
Desa	Copia	Redueix-ho tot
Amplia-ho tot	Filtra JSON	
ok:	true	
▼ contact:		
id:	11	
title:	"Mr."	
name:	"Mike"	
surname:	"Molina"	
email:	"molina@mail.com"	
birthdate:	"1975-10-21"	
▼ phones:		
▼ 0:		
number:	"295667788"	
type:	"Landline"	
▼ 1:		
number:	"666557744"	
type:	"Mobile"	

**Figure 2:** JSON response for the contact with id=11

### 2.3.3 Posting one contact (POST)

In this example we are going to insert a single contact without phones, but you can easily extend the functionality adding phones to an existing contact.

To insert a contact we use the POST method and the default route (/api/contact). As the method is different from GET, we can reuse the same route with a different operation.

Here, the `$request->getContent()` method is responsible for getting the json string.

```
#[Rest\Post('/', name: 'contact_api_new_contact')]
public function newContact(EntityManagerInterface $entityManager, Request
    $request): JsonResponse {

    try {
        $content = $request->getContent();
        $contact = new Contact();
        $contact->fromJson($content);

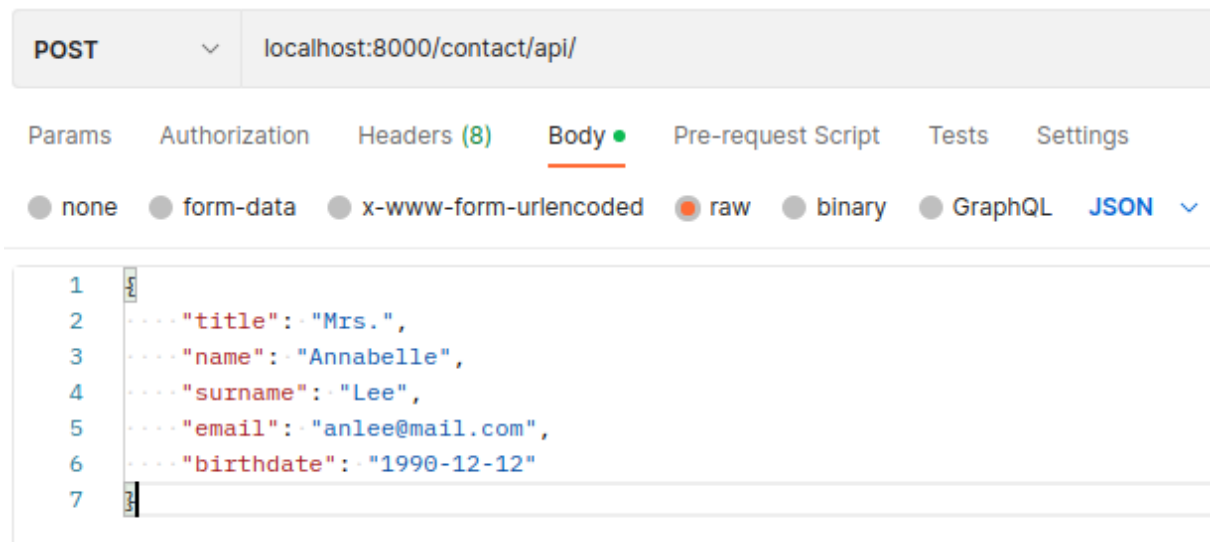
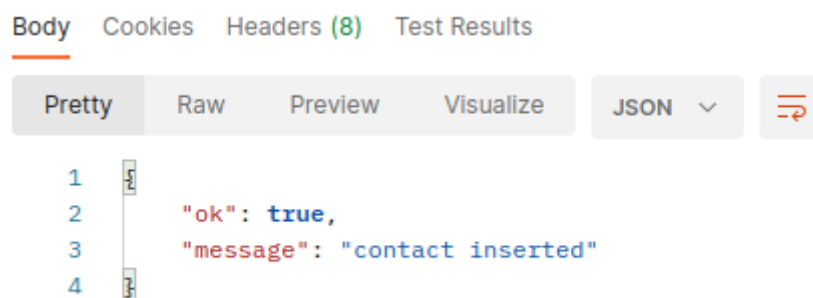
        $entityManager->persist($contact);
        $entityManager->flush();

        $response = [
            'ok' => true,
            'message' => 'contact inserted',
        ];
        return new JsonResponse($response, 201);
    } catch (\Throwable $e) {
        $response = [
            'ok' => false,
            'error' => 'Failed to insert contact: '.$e->getMessage(),
        ];
        return new JsonResponse($response, 400);
    }
}
```

To use the Request class, you need to import the `Symfony\Component\HttpFoundation\Request` library.

Note how we are handling the operation's errors with a try-catch block, because the `fromJson` method will launch an error if some key is inexistent.

To make a POST request with Postman, change the method to POST and write in the **Body** section, in **raw** format, the data that you want to insert:

**Figure 3:** POST request with Postman**Figure 4:** Result of the POST request

### 2.3.4 Modifying a contact (PUT)

This method is similar to the previous one, but fetching first the contact by its `id` and modifying it later with the request data. We are using the same base route but changing the method and adding the `id` of the contact.

```
#[Rest\Put('/{id<\d+>}', name: 'contact_api_edit_contact')]
public function editContact(EntityManagerInterface $entityManager, Request
    ↳ $request, $id=''): JsonResponse {

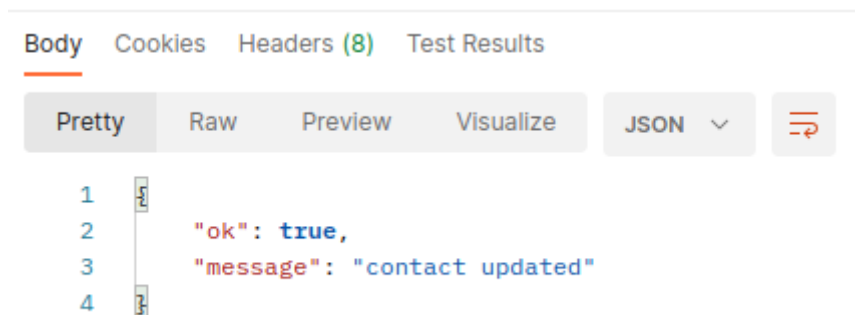
    try {
        $content = $request->getContent();

        $contact =
            ↳ $entityManager->getRepository(Contact::class)->find($id);
        $contact->fromJson($content);

        $entityManager->flush();

        $response = [
            'ok' => true,
            'message' => 'contact updated',
        ];
        return new JsonResponse($response, 201);
    } catch (\Throwable $e) {
        $response = [
            'ok' => false,
            'error' => 'Failed to update contact: '.$e->getMessage(),
        ];
        return new JsonResponse($response, 400);
    }
}
```

The PUT request with Postman can be done changing to the PUT method and writing the data with the desired changes:

**Figure 5:** PUT request with Postman**Figure 6:** Result of the PUT request

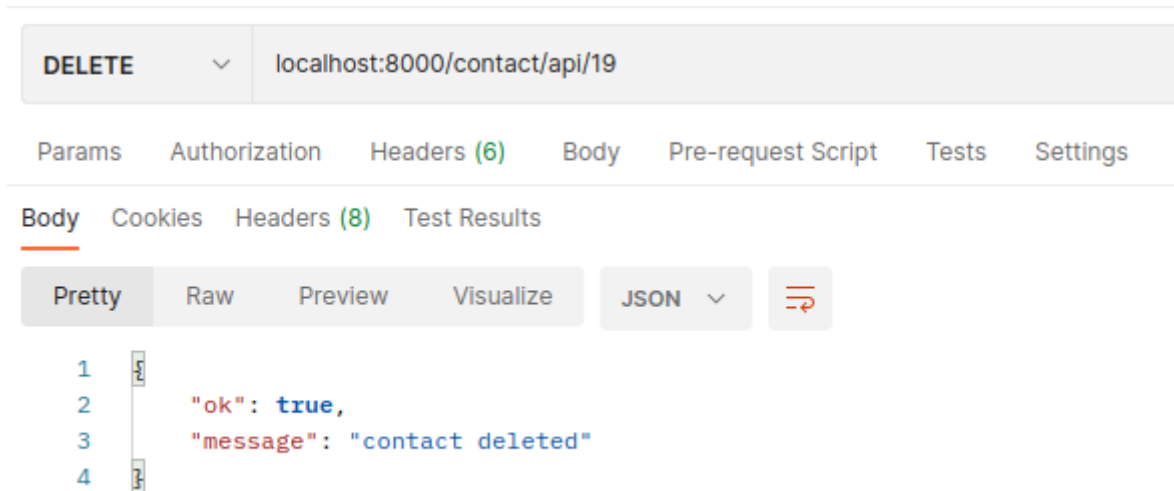
### 2.3.5 Removing a contact (DELETE)

The same as before, but using the DELETE method. Here we are deleting the contact's phones so that the DBMS doesn't throw an error.

```
#[Rest\Delete('/{id<\d+>}', name: 'contact_api_delete_contact')]
public function deleteContact(EntityManagerInterface $entityManager,
    ↪ $id=''): JsonResponse {
    $contact = $entityManager->getRepository(Contact::class)->find($id);
    if ($contact) {
        foreach($contact->getPhones() as $phone){
            $entityManager->remove($phone);
        }
        $entityManager->remove($contact);
        $entityManager->flush();

        $response = [
            'ok' => true,
            'message' => 'contact deleted',
        ];
        return new JsonResponse($response, 200);
    } else {
        $response = [
            'ok' => false,
            'error' => 'Delete failed: contact not found',
        ];
        return new JsonResponse($response, 404);
    }
}
```

And to do the DELETE request with Postman, we only need to specify the DELETE method and to pass the id with the URL:



**Figure 7:** DELETE request and result with Postman

Remember to save your requests in order to reuse them later.



## 2.4 Validation

In Symfony, the validation is done in the underlying objects. To do the data validation, we are going to use the Symfony validator. First, install the package in your project:

```
composer require symfony/validator
```

We will use it to validate that the data you receive from a service (POST or PUT) are correct before carrying out the corresponding insertions or modifications.

Validation is done by adding a set of rules, called **constraints**, to a class.

The first thing is to import the validation constraints on the Contact and Phone classes:

```
use Symfony\Component\Validator\Constraints as Assert;
```

Then, add the constraints as **asserts** to each property with attributes:

```
class Contact
{
    ...
    #[ORM\Column(length: 5, options: ["default"=> "Mr."])]
    #[Assert\NotBlank]
    private ?string $title = null;

    #[ORM\Column(length: 100)]
    #[Assert\NotBlank]
    private ?string $name = null;

    #[ORM\Column(length: 100, nullable: true)]
    private ?string $surname = null;

    #[ORM\Column(type: Types::DATE_MUTABLE)]
    #[Assert\NotBlank]
    private ?\DateTimeInterface $birthdate = null;

    #[ORM\Column(length: 50, nullable: true)]
    #[Assert>Email]
    private ?string $email = null;
}

class Phone
```

```
{
...
#[ORM\Id]
#[ORM\Column(length: 20)]
#[Assert\NotBlank]
private ?string $number = null;

#[ORM\Column(length: 10, options: ["default"=> "Mobile"])]
#[Assert\NotBlank]
private ?string $type = null;
```

If you want you can customize the error message:

```
#[Assert\Email(
    message: 'The email {{ value }} is not a valid email.',
)]
private ?string $email = null;
```

Here, a list of all the constraints: [Symfony validation constraints](#)

Then, in the `ApiContactController` class, import the `ValidatorInterface` and inject it in the PUT and POST methods for using it to validate the received data. The validator returns a list with the errors detected, so, if that number is zero, there are not errors and we can continue with the database operation. Each error is an object, so we use the methods `getPropertyPath` and `getMessage` to get the property and the correspondent error message:

```
use Symfony\Component\Validator\Validator\ValidatorInterface;

...

#[Rest\Post('/', name: 'contact_api_new_contact')]
public function newContact(EntityManagerInterface $entityManager, Request
    ↪ $request, ValidatorInterface $validator): JsonResponse {

    try {
        $content = $request->getContent();
        $contact = new Contact();
        $contact->fromJson($content);
        $errors = $validator->validate($contact);

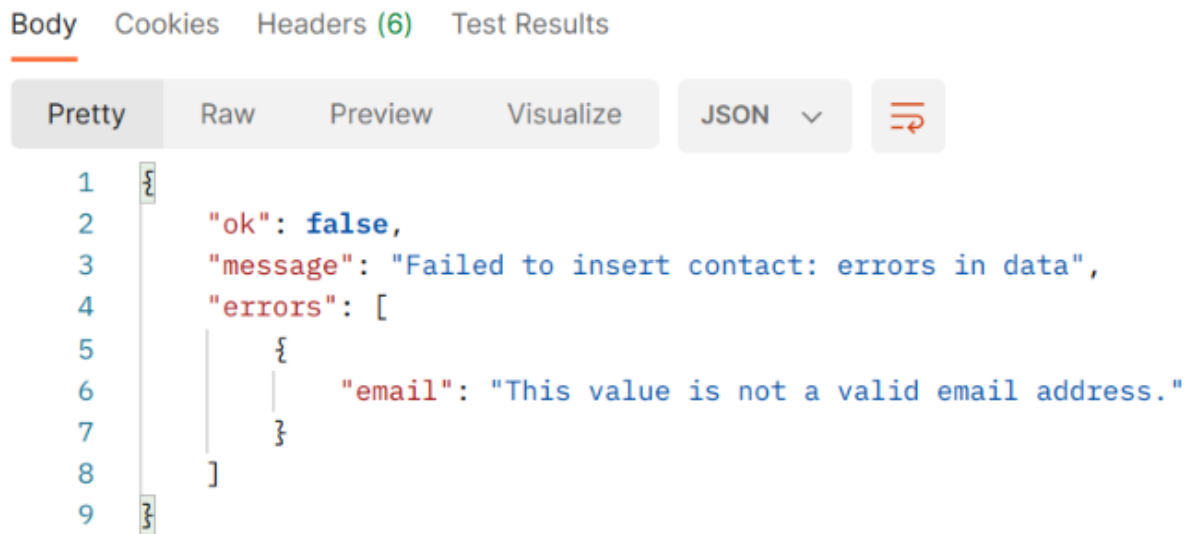
        if(count($errors) == 0)
```

```
{
    $entityManager->persist($contact);
    $entityManager->flush();

    $response = [
        'ok' => true,
        'message' => 'contact inserted',
    ];
    return new JsonResponse($response, 201);
} else {
    $errorsArray = [];
    foreach ($errors as $error){
        $errorsArray[] = array(
            $error->getPropertyPath() => $error->getMessage()
        );
    }
    $response = [
        'ok' => false,
        'message' => 'Failed to insert contact: errors in data',
        'errors' => $errorsArray ,
    ];
    return new JsonResponse($response, 400);
}
...
}
```

The code for the PUT method is similar.

Now, if we wanted to post a new contact with invalid data, the operation would not be done:



**Figure 8:** POST request with invalid email data

You can get all the code from the [GitHub repository](#).

### 3 API REST with API Platform

**API Platform** is an Open Source web framework for API-first projects. Describe the API's data model or import an existing one from Schema.org and get instantly a fully featured read/write API with REST operations, data validation, pagination, sorting, filtering, etc.

In this section, we will see an introduction to API Platform. You can expand your knowledge in their webpage: <https://api-platform.com/>

#### 3.1 Standards used by API Platform

API Platform generates the API in JSON, but it uses several technologies in addition. Let's see the more important:

- **JSON-LD:** JavaScript Object Notation for Linked Data. JSON-LD is designed around the concept of a **"context"** to provide additional mappings from JSON to an [RDF](#) model. So, with JSON-LD, we can define the schema of a JSON document.

<https://www.w3.org/TR/json-ld/>

- **Hydra:** Hydra is a vocabulary to simplify the development and consumption of web APIs. It's like JSON-LD, but with standardized operations. The core of Hydra is the **Hydra Core Vocabulary**, which describes the most common API operations.

<https://www.hydra-cg.com/>

```
{#381 ▼
+ "@context": "/contexts/Book"
+ "@id": "/books"
+ "@type": "hydra:Collection"
+ "hydra:member": array:1 [ ▼
  0 => {#619 ▼
    + "@id": "/books/1"
    + "@type": "Book"
    + "id": 1
    + "isbn": "9783161484100"
    + "title": "1st Book"
    + "description": "This is my first book synopsis"
    + "author": "Hemingroad"
    + "publicationDate": "2018-02-16T14:15:58+00:00"
    + "reviews": []
  }
+ "hydra:totalItems": 1
}
```

**Figure 9:** JSON-LD and Hydra fields

- **JSON Schema:** JSON Schema specifies a JSON-based format to define the structure of JSON data for validation, documentation, and interaction control. It is based on the concepts from XML Schema (XSD) but is JSON-based.

<https://json-schema.org/>

- **Swagger:** Swagger is an API Specification based on OpenAPI and a suite of tools based on it. In API Platform, Swagger is responsible for an interface for the JSON, JSON-LD, Hydra, JSON Schema, etc, data.

```
Contact.jsonld ^ Collapse all object
A contact
@context > Expand all read-only (string | object)
@id read-only string
@type read-only string
id read-only integer
title > Expand all string
name string
surname string | null
birthdate string date-time
email string | null
phones > Expand all array<string>
```

**Figure 10:** JSON Schema

## Test API application 1.0.0 OAS3

Description of Test API application

### Customer

GET	/api/customers	Retrieves the collection of Customer resources.
POST	/api/customers	Creates a Customer resource.
GET	/api/customers/{id}	Retrieves a Customer resource.
DELETE	/api/customers/{id}	Removes the Customer resource.

#### Schemas

- Customer >
- Customer:jsonld >

**Figure 11:** Swagger

### 3.2 Building the API REST with API Platform

To show how to work with API Platform, we will create a new project from scratch. We are going to do the menu of an Italian restaurant, with dishes grouped by categories.

The first thing is to create the project and the required dependencies:

```
symfony new italian_restaurant --version="7.1.*"  
cd italian_restaurant  
composer require --dev symfony/maker-bundle  
composer require api
```

As you can see, we only require the maker bundle, because we don't need views (Doctrine is installed with API Platform). The last packet, 'api', is the API Platform dependency.

Then, create the `.env.local` file to configure the database connection, as seen in the previous unit, and create the database:

```
DATABASE_URL="mysql://restaurant:restaurant@127.0.0.1:3306/restaurant?serverVersion=8.0.33"
or
DATABASE_URL="mysql://restaurant:restaurant@127.0.0.1:3306/restaurant?serverVersion=10.1.3-MariaDB&charset=utf8mb4"
```

```
CREATE USER 'restaurant'@'%' IDENTIFIED BY 'restaurant';  
CREATE DATABASE `restaurant`;  
GRANT ALL PRIVILEGES ON `restaurant`.* TO 'restaurant'@'%';
```

Finally, create the entities `dish` and `category`:

```
symfony console make:entity  
  
Class name of the entity to create or update:  
> Category  
  
Mark this class as an API Platform resource (expose a CRUD API for it)  
→ (yes/no) [no]:  
> yes
```

```
created: src/Entity/Category.php
created: src/Repository/CategoryRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this
↪ command.

New property name (press <return> to stop adding fields):
> name

Field type (enter ? to see all types) [string]:
>

Field length [255]:
> 100

Can this field be null in the database (nullable) (yes/no) [no]:
> no

updated: src/Entity/Category.php

Add another property? Enter the property name (or press <return> to stop
↪ adding fields):
> description

Field type (enter ? to see all types) [string]:
>

Field length [255]:
> 255

Can this field be null in the database (nullable) (yes/no) [no]:
> yes

updated: src/Entity/Category.php

Add another property? Enter the property name (or press <return> to stop
↪ adding fields):
>

Success!
```



```
symfony console make:entity Dish

created: src/Entity/Dish.php
created: src/Repository/DishRepository.php

New property name (press <return> to stop adding fields):
> name

Field type (enter ? to see all types) [string]:
> string

Field length [255]:
> 100

Can this field be null in the database (nullable) (yes/no) [no]:
> no

updated: src/Entity/Dish.php

Add another property? Enter the property name (or press <return> to stop
↩ adding fields):
> ingredients

Field type (enter ? to see all types) [string]:
> string

Field length [255]:
> 255

Can this field be null in the database (nullable) (yes/no) [no]:
> yes

updated: src/Entity/Dish.php

Add another property? Enter the property name (or press <return> to stop
↩ adding fields):
> price

Field type (enter ? to see all types) [string]:
> decimal

Precision (total number of digits stored: 100.00 would be 5) [10]:
```

```
> 10

Scale (number of decimals to store: 100.00 would be 2) [0]:
> 2

Can this field be null in the database (nullable) (yes/no) [no]:
> no

updated: src/Entity/Dish.php

Add another property? Enter the property name (or press <return> to stop
↵ adding fields):
> category

Field type (enter ? to see all types) [string]:
> relation

What class should this entity be related to?:
> Category

What type of relationship is this?
...
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne

Is the Dish.category property allowed to be null (nullable)? (yes/no)
↵ [yes]:
> no

Do you want to add a new property to Category so that you can
↵ access/update Dish objects from it - e.g. $category->getDishes()?
↵ (yes/no) [yes]:
> yes

A new property will also be added to the Category class so that you can
↵ access the related Dish objects from it.

New field name inside Category [dishes]:
> dishes

Do you want to automatically delete orphaned App\Entity\Dish objects
↵ (orphanRemoval)? (yes/no) [no]:
```

```
> no

updated: src/Entity/Dish.php
updated: src/Entity/Category.php

Add another property? Enter the property name (or press <return> to stop
↩ adding fields):
>

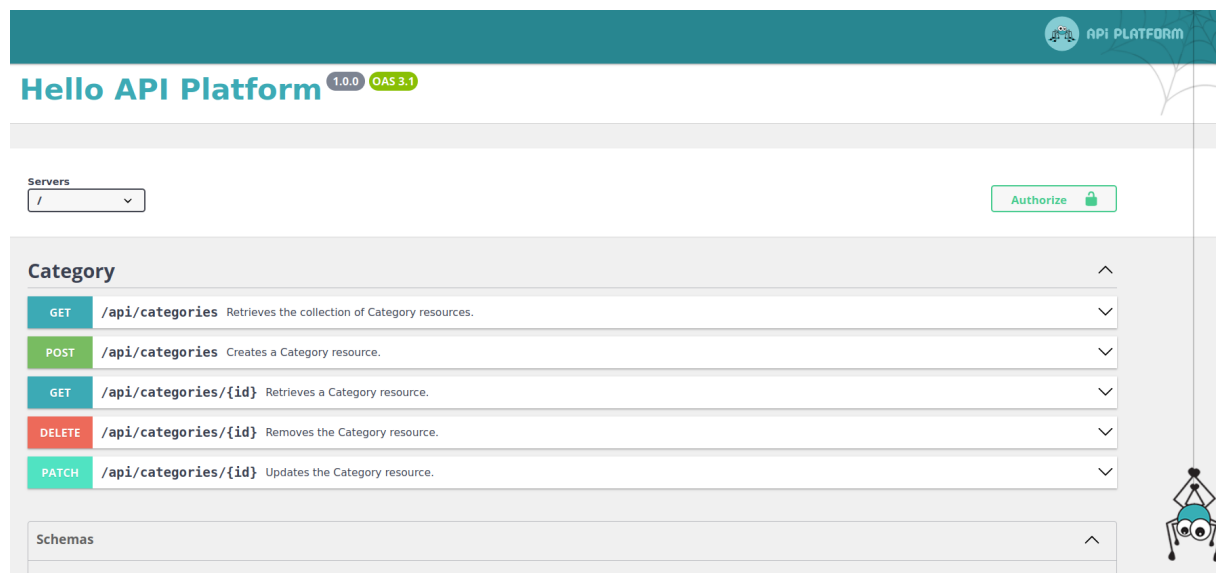
Success!
```

If you can't see the question Mark this class as an API Platform resource?, don't worry, we solve that soon.

And do the migration:

```
symfony console make:migration
symfony console doctrine:migrations:migrate
```

Now, run the server and go to the url <http://127.0.0.1:8002/api>. You will see something similar to:



**Figure 12:** API Platform main page

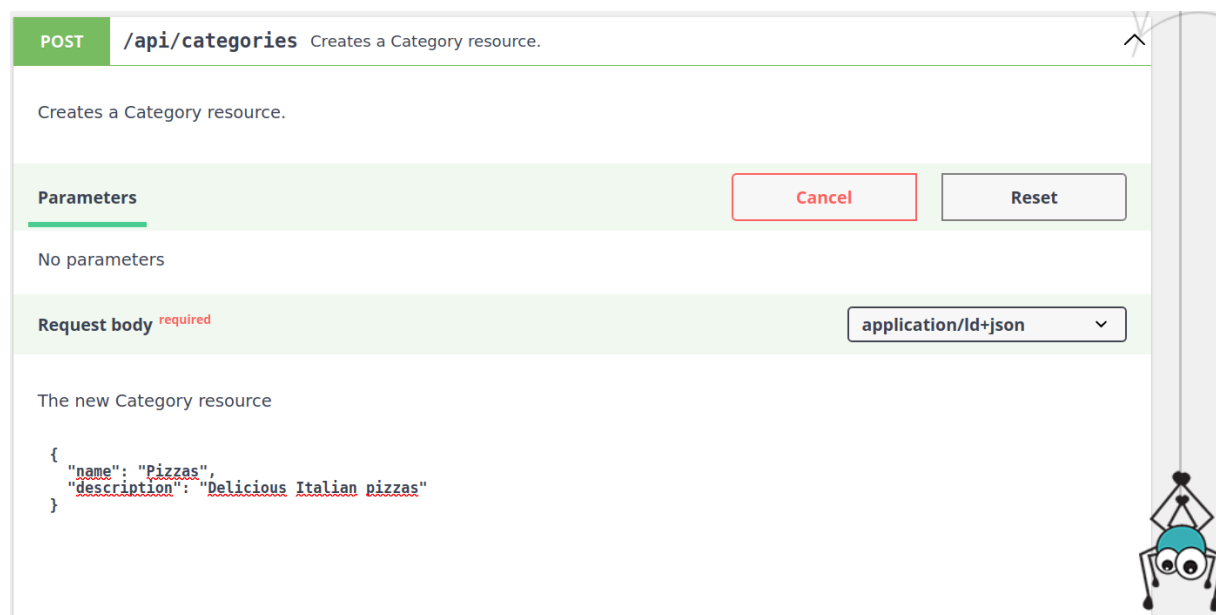
As you can see, we have all the usual REST operations for the Category entity.

But the Dish entity doesn't appear. We can solve it easily editing the Dish class, importing the ApiPlatform\Metadata\ApiResource component and adding the attribute #[ApiResource] to the class:

```
use ApiPlatform\Metadata\ApiResource;
...

#[ORM\Entity(repositoryClass: DishRepository::class)]
#[ApiResource]
class Dish
{
    ...
}
```

Now, refresh the page and you will see the Category operations. Time for adding a few categories and dishes to doing tests. You can do that with Postman or directly in the API Platform page: open the POST /api/categories item -> Try it out and enter the data for the categories you want (you don't have to put anything in the dishes array):



POST /api/categories Creates a Category resource.

Creates a Category resource.

Parameters

No parameters

Request body required

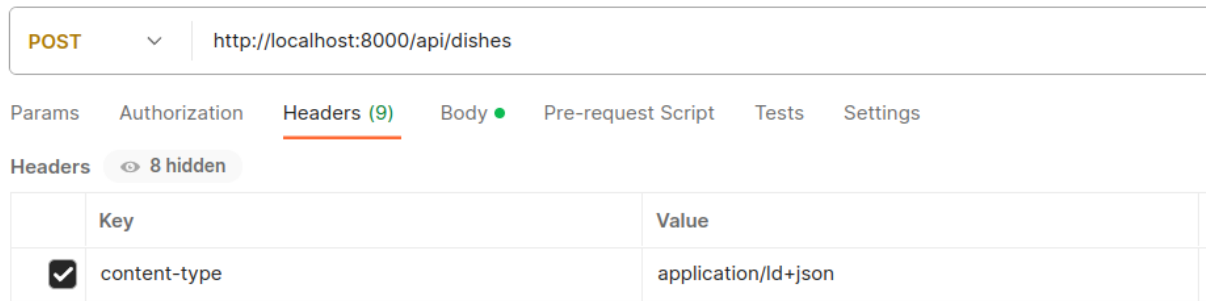
application/json

The new Category resource

```
{
  "name": "Pizzas",
  "description": "Delicious Italian pizzas"
}
```

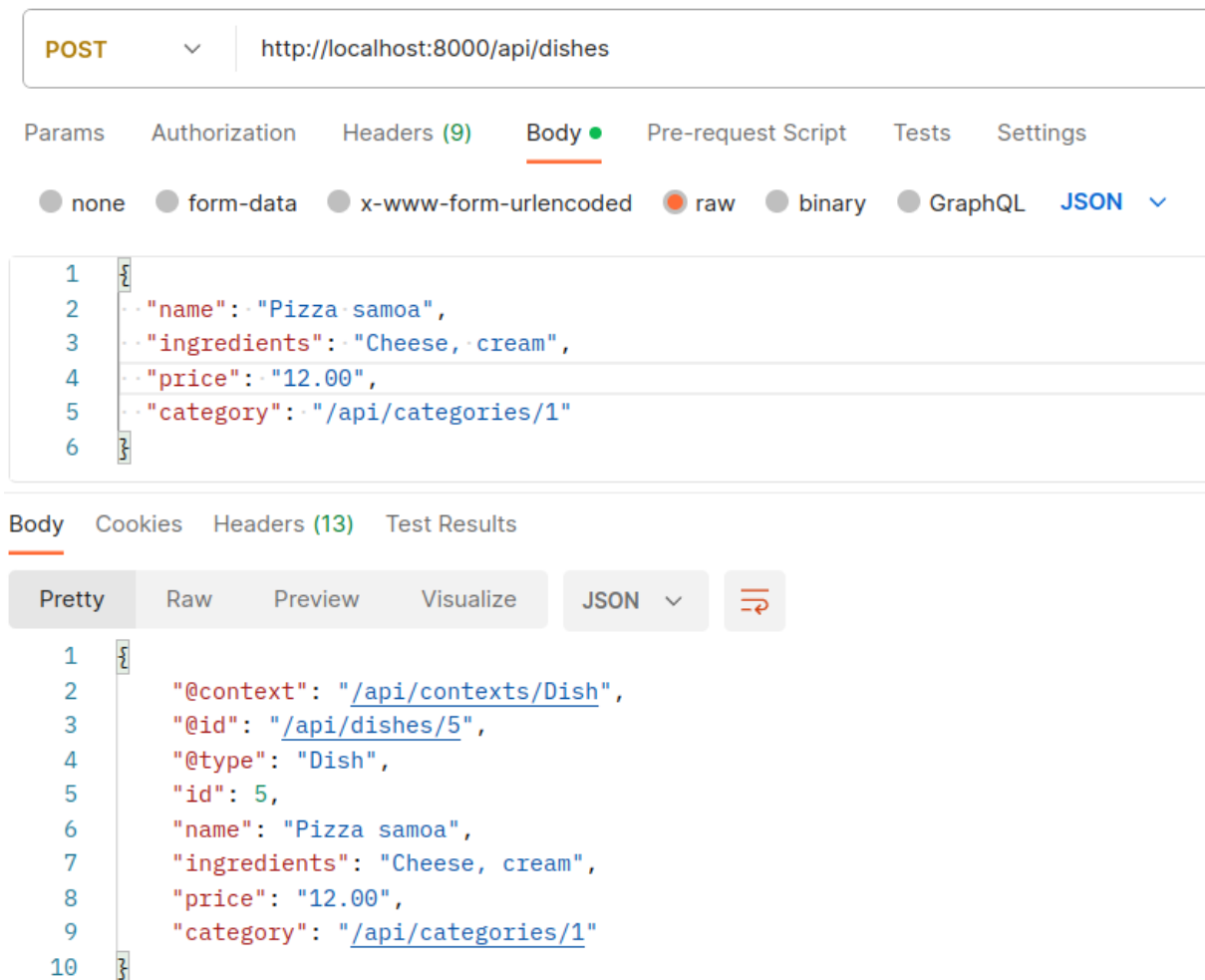
**Figure 13:** API Platform POST

To do the operations POST and PUT in Postman, you need to establish the **content-type** to application/json in the Headers section:



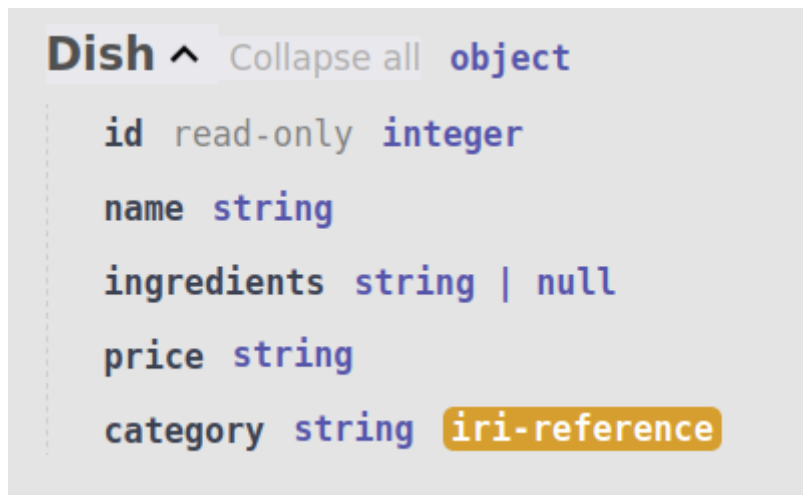
**Figure 14:** Header content-type

Do the request as usual, but note that the price and the category are strings. For the category you also need to indicate the route:

**Figure 15:** POST to API Platform

Check all the operations with Postman and the web interface. Note how now the GET responses are a bit different, because they include the JSON-LD (@context, @id, @type) and Hydra (hydra:totalItems and hydra:member) elements.

Finally, go to the bottom of the page and inspect the JSON Schema created automatically by API Platform:



**Figure 16:** JSON Schema of the Dish entity

### 3.3 Modifying the API

Probably you have noted that a new operation has appeared, PATCH. PATCH is similar to PUT but is intended for partial modifications. As PUT is also capable of doing that, we can remove PATCH from our API.

To do that, we need to expand the information of the ApiResource attribute, adding only the operations we want:

```
use ApiPlatform\Metadata\ApiResource;
use ApiPlatform\Metadata\Delete;
use ApiPlatform\Metadata\Get;
use ApiPlatform\Metadata\GetCollection;
use ApiPlatform\Metadata\Post;
use ApiPlatform\Metadata\Put;

...

#[ORM\Entity(repositoryClass: DishRepository::class)]
```

```
#[ApiResource (
    operations: [
        new Get(),
        new GetCollection(),
        new Post(),
        new Put(),
        new Delete(),
    ],
)]
class Dish
...
```

The same in the Category class.

Now, we have only the selected operations.

Category		
GET	/api/categories	Retrieves the collection of Category resources.
POST	/api/categories	Creates a Category resource.
GET	/api/categories/{id}	Retrieves a Category resource.
PUT	/api/categories/{id}	Replaces the Category resource.
DELETE	/api/categories/{id}	Removes the Category resource.
Dish		
GET	/api/dishes	Retrieves the collection of Dish resources.
POST	/api/dishes	Creates a Dish resource.
GET	/api/dishes/{id}	Retrieves a Dish resource.
PUT	/api/dishes/{id}	Replaces the Dish resource.
DELETE	/api/dishes/{id}	Removes the Dish resource.

**Figure 17:** Basic operations



We can also configure the operations. For instance in the next example, in the GET dishes by id operation, we can specify a different route with `uriTemplate` and we can require the id to be an integer with `requirements`:

```
#[ApiResponse(operations: [  
new Get(  
    uriTemplate: '/food/{id}',  
    requirements: ['id' => '\d+'],  
)  
...]
```

If you want to prefix all routes to all operations, add the `routePrefix` attribute for the whole entity:

```
#[ApiResponse(routePrefix: '/menu')]  
class Dish  
{
```

### 3.4 Serialization

Another thing we can do is to hide some fields, or to make them read-only or write-only. To do that we need to know the concepts of serialization, normalization and de-normalization.

Symfony uses **serialization** to convert entities (objects representing data in the application) into a format that can be easily transmitted via HTTP, such as JSON.

Serialization in API Platform involves the following key concepts:

- **Normalization:** The process of converting your data (objects or arrays) into a format suitable for serialization. It involves transforming data structures into JSON.
- **De-normalization:** The reverse process of normalization, where the serialized data received from the client, in JSON format, is converted back into complex data structures within your application.
- **Context:** The serialization and de-normalization processes can be influenced by a context that provides additional information or constraints. For example, you might want to include or exclude specific properties based on the API endpoint, user roles, or other conditions. The **normalizationContext** and **denormalizationContext** options in API Platform allow you to customize this behavior.

To specify what groups to use in the API system:

1. Add the `normalizationContext` and `denormalizationContext` attributes to the resource, and specify which groups to use. Here you see that we add *read* and *write*, respectively, but you can use any group names you wish.
2. Apply the groups to properties in the object.

```
use Symfony\Component\Serializer\Annotation\Groups;

...

#[ApiResponse (
    ...
    normalizationContext: ['groups' => ['read']],
    denormalizationContext: ['groups' => ['write']],
)]
class Category
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    #[Groups(['read', 'write'])]
    private ?int $id = null;

    #[ORM\Column(length: 100)]
    #[Groups(['read', 'write'])]
    private ?string $name = null;

    #[ORM\Column(length: 255, nullable: true)]
    #[Groups(['read', 'write'])]
    private ?string $description = null;

    #[ORM\OneToMany(mappedBy: 'category', targetEntity: Dish::class)]
    private Collection $dishes;

    ...
}
```

In the previous example, we have created the *read* group for normalization (when we transform our data to JSON), and the *write* group for de-normalization (when we transform our JSON to data to entities). Then we've applied the groups to all the properties except for `$dishes`. If you go to the Swagger UI and do any operation you will see that the `$dishes` array doesn't appear. If you want to see the array only in the get operations, simply add the `#[Groups(['read'])]` attribute to `$dishes`.

### 3.5 Documenting the API

The Swagger interface also works as a documentation page. We can add annotation comments to our classes and they will be visible on the frontend.

For instance, add the next comments to the Category class:

```
/**
 * The name of the category
 */
#[ORM\Column(length: 100)]
private ?string $name = null;

/**
 * The category description
 */
#[ORM\Column(length: 255, nullable: true)]
private ?string $description = null;

/**
 * A list with the dishes of each category
 */
#[ORM\OneToMany(mappedBy: 'category', targetEntity: Dish::class)]
private Collection $dishes;
```

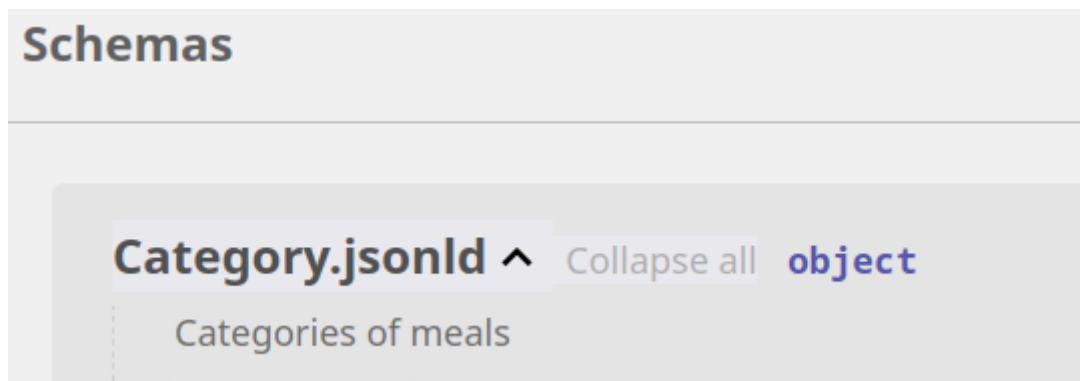
Now the descriptions appear on the Schema section:



**Figure 18:** Schema comments

We can also describe each class adding a description element to the ApiResource attribute:

```
...
#[ApiResource (
  operations: [
    new Get(),
    new GetCollection(),
    new Post(),
    new Put(),
    new Delete(),
  ],
  description: 'Categories of meals',
)]
class Category
...
```



**Figure 19:** Class description

### 3.6 Subresources

When you do a GET operation for `/api/dishes`, you get something like:

```
{
  "@id": "/api/menu/food/1",
  "@type": "Dish",
  "id": 1,
  "name": "Pepperoni pizza",
  "ingredients": "Mozzarella, pepperoni, etc",
  "price": "10.50",
  "category": "/api/categories/1"
}
```

But, what does mean `"/api/categories/1"`? It will be better if we could see the category name instead of their route.

It is possible to embed related objects (or only some of their properties) directly in the parent response through the use of serialization groups. By using the following serialization groups attribute (`#[Groups]`), a JSON representation of the category is embedded in the dish response. As soon as any of the category's attributes is in the dish group, the category will be embedded:

```
// entity/Dish

#[ApiResponse (
    ...
    normalizationContext: ['groups' => ['dish']]
)]
class Dish
{
    ...
    #[Groups('dish')]
    private ?int $id = null;

    ...
    #[Groups('dish')]
    private ?string $name = null;

    ...
    #[Groups('dish')]
    private ?string $ingredients = null;

    ...
    #[Groups('dish')]
    private ?string $price = null;

    ...
    #[Groups('dish')]
    private ?Category $category = null;
    ...
}

//entity/Category

#[ApiResponse (
    ...
    normalizationContext: ['groups' => ['read']],
    ...
)]
```

```
denormalizationContext: ['groups' => ['write']],
)]
class Category
{
    ..
    #[Groups(['read', 'write'])]
    private ?int $id = null;

    ...
    #[Groups(['read', 'write', 'dish'])]
    private ?string $name = null;

    ...
    #[Groups(['read', 'write', 'dish'])]
    private ?string $description = null;
    ...
}
```

Now, you can get the category name and description for each dish:

```
{
  "@id": "/api/menu/food/1",
  "@type": "Dish",
  "id": 1,
  "name": "Pepperoni pizza",
  "ingredients": "Mozzarella, pepperoni, etc",
  "price": "10.50",
  "category": {
    "@id": "/api/categories/1",
    "@type": "Category",
    "name": "Pizzas",
    "description": "Delicious Italian pizzas"
  }
}
```

Another way to visualize related entities is creating a new route in which we can get the entities related by another.

In our example we are going to make a new route to show all the dishes related to a category. The route will be `/categories/{id}/dishes`:

```
// Entity/Dish

use ApiPlatform\Metadata\Link;

...

#[ApiResponse(
    uriTemplate: '/categories/{id}/dishes',
    operations: [ new GetCollection() ],
    uriVariables: [
        'id' => new Link(fromClass: Category::class, toProperty:
            => 'category'),
    ],
)]
class Dish
...
```

We use the `uriTemplate` attribute to create the route, `operations` to specify that we want to get a collection of dishes and `uriVariables` to link the `Category` class to the `category` property in `Dish`. If you go to the route `/api/categories/2/dishes`, for instance, you'll get all the dishes of the category with id 2.

### 3.7 Filters

We can use the Api Platform filters to search specific data in our api.

To use them, we can add the `ApiFilter` attribute and the required dependencies:

```
use ApiPlatform\Metadata\ApiResource;
use ApiPlatform\Metadata\ApiFilter;
use ApiPlatform\Doctrine\Orm\Filter\SearchFilter;

..

#[ApiFilter(SearchFilter::class, properties: ['name' => 'partial'])]
class Dish
...
```

With this code we can search by a custom string in the `name` property. For example, if we go to the GET `/api/dishes` request in the Swagger frontend, we can find a new field for searching in the `name` property:

The screenshot shows a web interface for the 'Dish' API. At the top, there's a header 'Dish'. Below it, a teal bar contains 'GET' and the endpoint '/api/dishes' with a description 'Retrieves the collection of Dish resources.' Below this, a light blue box labeled 'Parameters' contains two input fields. The first is 'page' (integer, query) with a value of '1' and a checkbox for 'Send empty value'. The second is 'name' (string, query) with a value of 'pizza'. A red 'Cancel' button is in the top right of the parameters section.

**Figure 20:** Search filter

If we write `p i z z a`, we retrieve all the dishes with the `p i z z a` string in their name.

If we want to use the api endpoint directly, we can write the query in the query string:

`http://127.0.0.1:8000/api/dishes?name=pizza`

The search filter supports **exact**, **partial**, **start**, **end**, and **word\_start** matching strategies:

- **exact** searches for fields with the exact string.
- **partial** strategy uses `LIKE %text%` to search for fields that contain **text**.
- **start** strategy uses `LIKE text%` to search for fields that start with **text**.
- **end** strategy uses `LIKE %text` to search for fields that end with **text**.
- **word\_start** strategy uses `LIKE text% OR LIKE % text%` to search for fields that contain words starting with **text**.

In MySQL the commonly used **utf8\_unicode\_ci collation** (or **utf8mb4\_unicode\_ci**) performs case-insensitive searches by default.

We can specify several search filters:

```
#[ApiFilter(SearchFilter::class, properties: ['name' => 'partial',
  ↳ 'ingredients' => 'partial'])]
class Dish
```



And for searching:

`http://127.0.0.1:8000/api/dishes?name=pizza&ingredients=mozzarella`

We also have **date**, **boolean** and **numeric** filters for searching with these types:

[Api Platform Doctrine filters](#)

The **order filter** allows sorting a collection against the given properties:

```
use ApiPlatform\Doctrine\Orm\FILTER\OrderFilter;
...
#[ApiFilter(OrderFilter::class, properties: ['name', 'ingredients'])]
class Dish
...
```

`http://127.0.0.1:8000/api/dishes?order[name]=asc&order[ingredients]=desc`

More about [order filters](#).

### 3.8 Validation

API Platform takes care of validating the data sent to the API, relying on the Symfony Validator Component. As the validator package is included with API Platform, we only need to add the library and the asserts to each class:

```
use Symfony\Component\Validator\Constraints as Assert;
...
#[ORM\Column(length: 100)]
#[Assert\NotBlank]
private ?string $name = null;
```

If you try to post a category with an empty name, you will get an error message generated by the validator:

422

Error: Unprocessable Entity

Response body

```
{
  "@id": "/api/validation_errors/c1051bb4-d103-4f74-8988-acbcafc7fdc3",
  "@type": "ConstraintViolationList",
  "status": 422,
  "violations": [
    {
      "propertyPath": "name",
      "message": "This value should not be blank.",
      "code": "c1051bb4-d103-4f74-8988-acbcafc7fdc3"
    }
  ],
  "detail": "name: This value should not be blank.",
  "hydra:title": "An error occurred",
  "hydra:description": "name: This value should not be blank.",
  "type": "/validation_errors/c1051bb4-d103-4f74-8988-acbcafc7fdc3",
  "title": "An error occurred"
}
```

**Figure 21:** Validation error

Write the next asserts in the Dish class and check them sending data with POST and PUT:

```
#[Assert\NotBlank]
private ?string $name = null;
...
#[Assert\Type('numeric')]
#[Assert\PositiveOrZero]
private ?string $price = null;
...
#[Assert\NotBlank]
private ?Category $category = null;
```

## 4 Security

Symfony's security system is very powerful and versatile, although it can also be complicated to understand and configure. In this unit we will learn its main elements:

- The **authentication mechanism**: how the users can access the app and where their credentials are stored.
- The **authorization mechanism**: once the user has been logged in correctly, determine their permissions and what resources can access and which can't.

### 4.1 The `security.yaml` file

The `config/packages/security.yaml` file stores the general configuration of the security system of our Symfony application. Its default content is:

```
security:

    providers:
        users_in_memory: { memory: null }

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory

    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

Their main sections are:

- The User **providers**: The user provider loads users from any storage (e.g. the database) based on a "user identifier" (e.g. the user's email address).
- The **Firewall**: The firewall is the core of securing your application. Every request within the firewall is checked if it needs an authenticated user. The firewall also takes care of authenticating this user (e.g. using a login form).
- **Access Control** (Authorization): Using access control and authorization, you control the required permissions to perform a specific action or visit a specific URL.

## 4.2 The User

Permissions in Symfony are always linked to a **user** object. If you need to secure your application, you need to create a user class. This is a class that implements `UserInterface`. This is often a Doctrine entity.

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

```
symfony console make:user
The name of the security user class (e.g. User) [User]:
> User

Do you want to store user data in the database (via Doctrine)? (yes/no)
↳ [yes]:
> yes

Enter a property name that will be the unique "display" name for the user
↳ (e.g. email, username, uuid) [email]:
> email

Will this app need to hash/check user passwords? Choose No if passwords
↳ are not needed or will be checked/hashed by some other system (e.g. a
↳ single sign-on server).

Does this app need to hash/check user passwords? (yes/no) [yes]:
> yes

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

This is the normal procedure: we tell to the maker the **user class name** (**User**), if we want to store the user's data in the database via Doctrine (**yes**), the field used to login or display name (**'email'**) and if the system must hash the password before store it (**yes**).

Once finished the process, the maker has created 2 files, `src/Entity/User.php` and `src/Repository/UserRepository.php`, and has updated the providers section in the `security.yaml` reflecting the new users provider:

```
providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email
```

Now, the system knows which class must be used to store and to check users (*App\Entity\User*) and which property is used to identify the user (*email*).

User providers are used in a couple places during the security lifecycle:

- **Load the User based on an identifier:** During login, the provider loads the user based on the user id.
- **Reload the User from the session:** At the beginning of each request, the user is loaded from the session. The provider “refreshes” the user from the database to make sure all user information is up to date.

Once created the User class, the next step is to do the migration process. But before, you can add more properties to the class, such as name, surname, address, phone, etc, with `symfony console make:entity User` and telling the assistant the new properties you want.

If your class is OK, do the migration progress as usual:

```
symfony console make:migration
symfony console doctrine:migrations:migrate
```

And the table user will be created in the database:

```
mysql> describe user;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra           |
+-----+-----+-----+-----+-----+-----+
| id         | int           | NO   | PRI | NULL    | auto_increment |
| email      | varchar(180)  | NO   | UNI | NULL    |                 |
| roles      | json          | NO   |     | NULL    |                 |
| password   | varchar(255)  | NO   |     | NULL    |                 |
+-----+-----+-----+-----+-----+-----+
```

In MariaDB the roles field is done as a longtext type instead of json, but it works in the same way.

#### 4.2.1 Hashing Passwords

The *SecurityBundle* provides password hashing and verification functionality. If you have created your user as shown before, your user class should implement the `PasswordAuthenticatedUserInterface`:

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

And your `security.yaml` file should have been configured:

```
security:  
password_hashers:  
    Sym-  
    fony\Component\Security\Core\User>PasswordAuthenticatedUserInterface:  
    'auto'
```

#### 4.3 Token based authentication

Traditional authentication mechanisms in web apps are based on sessions: the user submits his credentials through some form, the server validates and stores the data of the logged in user in the session, so that, while the session doesn't expire or isn't closed by the user, they can continue accessing without having to re-login.

However, this type of authentication has the limitation of being exclusive to web applications. If we want to adapt the application to mobile or desktop versions, we need another mechanism.

To get over this, we can use **token-based authentication**. This is a *stateless* authentication, which means that nothing is stored between client and server to continue accessing authenticated (no sessions). What is done is the following:

1. The client sends to the server its credentials (username and password).

2. The server validates them, and if they are correct, generates an encrypted string called **token**, which contains the user's validation, plus some additional information (such as the user's login, for example). This token is sent back to the user as a response to their authentication.
3. From this point on, whenever the client wants to authenticate against the server to request a resource, it needs to send the provided token. The server will verify it and grant or deny access.

Like sessions, tokens can also expire, which is indicated within the token itself. After the expiration time, if the server receives the token, it will discard it as invalid (expired), and the client will again be unauthenticated.

A JWT token has the next components:

- **Header:** has two parts, the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- **Payload:** the data encoded in Base64Url.
- **Signature:** created with the encoded header, the encoded payload, a secret and the algorithm specified in the header.

You can see an example of a JWT token in <https://jwt.io/>

## 4.4 Configuring the API

We will use the [lexik/jwt-authentication-bundle](#). Install it with composer:

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

In Linux, you probably need the **php sodium** extension.

Then, generate the SSL keys:

```
symfony console lexik:jwt:generate-keypair
```

The keys will be stored in `config/jwt/private.pem` and `config/jwt/public.pem`. Check that they are also in the `.gitignore` file for security reasons.

Next, move the SSL keys paths and passphrase from your `.env` file to your `.env.local` file for the same reason:

```
JWT_SECRET_KEY=%kernel.project_dir%/config/jwt/private.pem
JWT_PUBLIC_KEY=%kernel.project_dir%/config/jwt/public.pem
JWT_PASSPHRASE=autogenerated_passphrase
```

And write in `config/packages/lexik_jwt_authentication.yaml`:

```
lexik_jwt_authentication:
  secret_key: '%env(resolve:JWT_SECRET_KEY)%' # required for token
  ↪ creation
  public_key: '%env(resolve:JWT_PUBLIC_KEY)%' # required for token
  ↪ verification
  pass_phrase: '%env(JWT_PASSPHRASE)%' # required for token creation
  token_ttl: 3600 # ttl (duration) in seconds, default is 3600
```

To configure the API security firewalls in `security.yaml`, we need to place the `api` and the `main` firewalls as shown below:

```
firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  api:
    pattern: ^/api/
    stateless: true
    provider: app_user_provider
    jwt: ~
  main:
    json_login:
      check_path: auth
      username_path: email
      password_path: password
      success_handler:
  ↪ lexik_jwt_authentication.handler.authentication_success
      failure_handler:
  ↪ lexik_jwt_authentication.handler.authentication_failure
```

And, in the `access_control` section of the same file:

```
access_control:
  - { path: ^/api$, roles: PUBLIC_ACCESS } # Allows accessing the Swagger
  ↪ UI
  - { path: ^/auth, roles: PUBLIC_ACCESS }
  - { path: ^/api/users$, roles: PUBLIC_ACCESS, methods: POST }
  - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
```



You must also declare the route used for /auth in `config/routes.yaml`:

```
auth:
  path: /auth
  methods: ['POST']
```

With these paths any user can access the Swagger frontend and to the /auth path, but if you try to access any path of the API, you'll get the response:

```
{
  "code": 401,
  "message": "JWT Token not found"
}
```

## 4.5 Registering users

If you go to the new path `http://localhost:8000/auth` (using Postman) and enter an email and a password, you'll get the response:

```
{
  "code": 401,
  "message": "Invalid credentials."
}
```

This is because we don't have any registered user. We need to modify the User entity to allow registrations through it:

```
<?php
# api/src/Entity/User.php

namespace App\Entity;

use ApiPlatform\Metadata\ApiResource;
use ApiPlatform\Metadata\Post;
use ApiPlatform\Metadata\Put;
use Doctrine\ORM\Mapping as ORM;
use App\Repository\UserRepository;
use App\State\UserPasswordHasher;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

```
use
    ↳ Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Serializer\Annotation\Groups;
use Symfony\Component\Validator\Constraints as Assert;

#[ApiResponse(
    operations: [
        new Post(processor: UserPasswordHasher::class, validationContext:
            ↳ ['groups' => ['Default', 'user:create']]),
        new Put(processor: UserPasswordHasher::class),
    ],
    normalizationContext: ['groups' => ['user:read']],
    denormalizationContext: ['groups' => ['user:create', 'user:update']],
)]
#[ORM\Entity(repositoryClass: UserRepository::class)]
#[ORM\Table(name: '`user`')]
#[UniqueEntity('email')]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[Groups(['user:read'])]
    #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    #[ORM\GeneratedValue]
    private ?int $id = null;

    #[Assert\NotBlank]
    #[Assert\Email]
    #[Groups(['user:read', 'user:create', 'user:update'])]
    #[ORM\Column(length: 180, unique: true)]
    private ?string $email = null;

    #[ORM\Column]
    private ?string $password = null;

    #[Assert\NotBlank(groups: ['user:create'])]
    #[Groups(['user:create', 'user:update'])]
    private ?string $plainPassword = null;

    #[ORM\Column(type: 'json')]
    private array $roles = [];
```

```
public function getId(): ?int
{
    return $this->id;
}

public function getEmail(): ?string
{
    return $this->email;
}

public function setEmail(string $email): self
{
    $this->email = $email;

    return $this;
}

/**
 * @see PasswordAuthenticatedUserInterface
 */
public function getPassword(): string
{
    return $this->password;
}

public function setPassword(string $password): self
{
    $this->password = $password;

    return $this;
}

public function getPlainPassword(): ?string
{
    return $this->plainPassword;
}

public function setPlainPassword(?string $plainPassword): self
{
    $this->plainPassword = $plainPassword;

    return $this;
}
```

```
}

/**
 * @see UserInterface
 */
public function getRoles(): array
{
    $roles = $this->roles;

    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}

public function setRoles(array $roles): self
{
    $this->roles = $roles;

    return $this;
}

/**
 * A visual identifier that represents this user.
 *
 * @see UserInterface
 */
public function getUserIdentifier(): string
{
    return (string) $this->email;
}

/**
 * @see UserInterface
 */
public function eraseCredentials(): void
{
    $this->plainPassword = null;
}
}
```

The key concepts of the modification are:

- In the POST and PUT operations, we use a processor called UserPasswordHasher. This processor will catch the data in these operations and process them to hash the plain text password.
- We only expose the fields email and plainPassword.

Next, in src/State, create the processor UserPasswordHasher.php:

```
<?php
# api/src/State/UserPasswordHasher.php

namespace App\State;

use ApiPlatform\Metadata\Operation;
use ApiPlatform\State\ProcessorInterface;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

final class UserPasswordHasher implements ProcessorInterface
{
    public function __construct(private readonly ProcessorInterface
        ↪ $processor, private readonly UserPasswordHasherInterface
        ↪ $passwordHasher)
    {
    }

    public function process($data, Operation $operation, array $uriVariables
        ↪ = [], array $context = [])
    {
        if (!$data->getPlainPassword()) {
            return $this->processor->process($data, $operation,
                ↪ $uriVariables, $context);
        }

        $hashedPassword = $this->passwordHasher->hashPassword(
            $data,
            $data->getPlainPassword()
        );
        $data->setPassword($hashedPassword);
        $data->eraseCredentials();

        return $this->processor->process($data, $operation, $uriVariables,
            ↪ $context);
    }
}
```

This processor captures the data sent to the server and takes the `plainPassword` field to generate a hashed password, which is stored in the `password` field. Finally, deletes the plain password for security.

The last step is to register the processor in the file `config/services.yaml`:

```
# api/config/services.yaml
services:

    App\State\UserPasswordHasher:
        bind:
            $processor:
                → '@api_platform.doctrine.orm.state.persist_processor'
```

## 4.6 Access control

For our app we want the next roles and permissions:

- Unauthenticated users: can access to the registration (POST /api/users) and to the authentication (POST /auth) routes.
- Normal users (ROLE\_USER): can access to all the GET routes.
- Admin users (ROLE\_ADMIN): can access to all the operation routes.

First, we need at least one admin user. A simple way to do that without doing an extensive user management system, is adding the next lines to the `UserPasswordHasher` processor:

```
public function process($data, Operation $operation, array $uriVariables =
→ [], array $context = [])
{
    ...

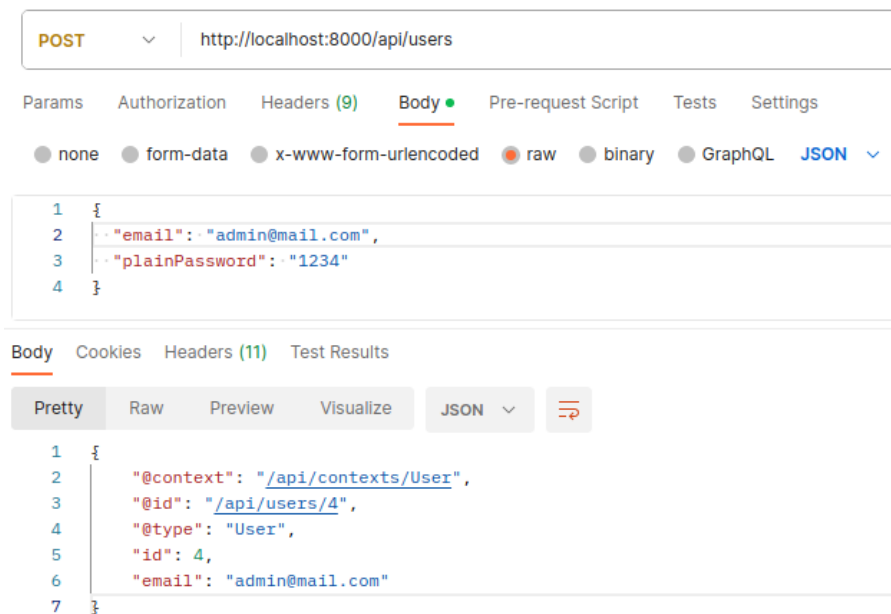
    $data->eraseCredentials();

    //Set admin rol for the next registered user
    //Comment these lines to return to normal users
    $roles[] = 'ROLE_ADMIN';
    $data->setRoles($roles);

    return $this->processor->process($data, $operation, $uriVariables,
→ $context);
}
```

Don't forget to comment the lines once you have the desired admin users.

To register users with Postman go th the POST `/api/users` route and send the users data, email and plainPassword. You will get a response with the id of the new user and their email:



**Figure 22:** Registering users

On your database now you can see the registered users and their roles (the `ROLE_USER` is added automatically every time we get a user from the database, see the `getRoles()` method in the `User` class):

id	email	roles	password
2	pepe@mail.com	[ ]	\$2y\$13\$NlEGNIPA8bnRx2du3TyNN.HB6esuaDQgGHY5NT8/IRET9jrlfdQza
4	admin@mail.com	[ "ROLE_ADMIN" ]	\$2y\$13\$tD67PFggETCTSNoma9mZR.tVciyCL.R2.rS0jDfwuMgYabZzBJG9K

**Figure 23:** Users in the DB

Once we have users with the admin and normal roles, we only need to add security attributes to the related classes:

```
// /Entity/Category

#[ApiResponse (
    description: 'Categories of meals',
    operations: [
```

```
        new Get(),
        new GetCollection(),
        new Post(security: "is_granted('ROLE_ADMIN')"),
        new Put(security: "is_granted('ROLE_ADMIN')"),
        new Delete(security: "is_granted('ROLE_ADMIN')"),
    ],
    security: "is_granted('ROLE_USER')",
    ...
)]
class Category
```

```
// /Entity/Dish

#[ApiResponse (
    operations: [
        new Get(
            uriTemplate: '/food/{id}',
            requirements: ['id' => '\d+'],
        ),
        new GetCollection(),
        new Post(
            defaults: ['ingredients' => 'Pasta'],
            security: "is_granted('ROLE_ADMIN')",
        ),
        new Put(security: "is_granted('ROLE_ADMIN')"),
        new Delete(security: "is_granted('ROLE_ADMIN')"),
    ],
    security: "is_granted('ROLE_USER')",
    ...
)]
class Dish
```

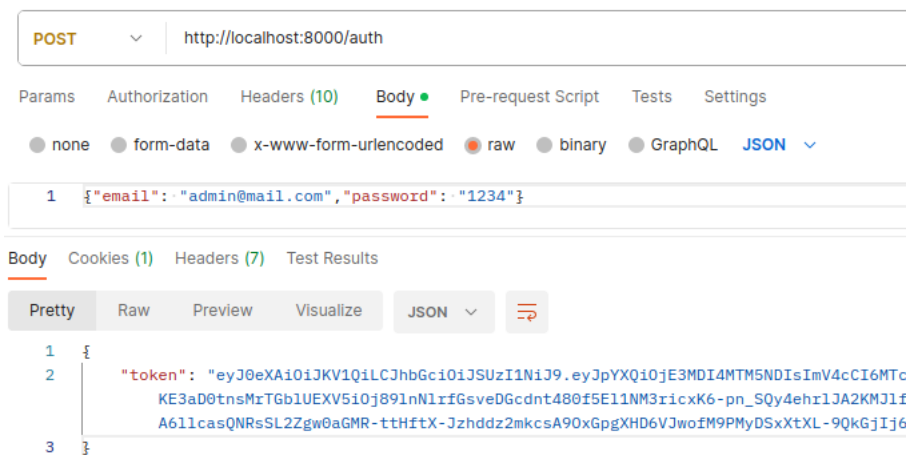
As you can see, we are requiring the `ROLE_USER` for all the operations with `security: "is_granted('ROLE_USER')"` for the whole class, and the `ROLE_ADMIN` for only the PUT, POST and DELETE operations (`new Post(security: "is_granted('ROLE_ADMIN')")`, etc).

More information about security in API Platform: <https://api-platform.com/docs/core/security/>



## 4.7 Usage

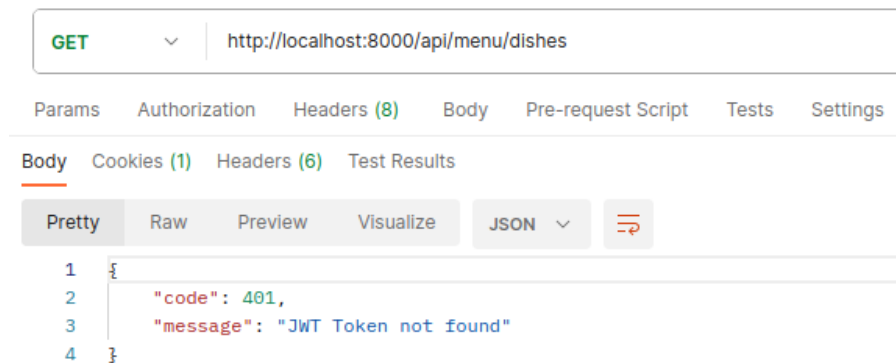
The first step is to authenticate the user using its credentials. You can test getting the token with Postman. Send a POST request to the `/auth` route and a json string in the body with valid username and password:



**Figure 24:** Getting token

Copy the value of “token”.

Now, make a get request without the token. You will get *JWT Token not found* message.:



**Figure 25:** Unauthenticated GET request

To do a successful request, we must add an **Authorization** header which value will be “**Bearer**” (with an ending space) followed by the token value:

GET http://localhost:8000/api/menu/dishes

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Headers 7 hidden

Key	Value
<input checked="" type="checkbox"/> Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpYXQiOiJlE3MDI4MTMzMj...
Key	Value

Body Cookies (1) Headers (13) Test Results

Pretty Raw Preview Visualize JSON ↕

```
1 {
2   "@context": "/api/contexts/Dish",
3   "@id": "/api/menu/dishes",
4   "@type": "hydra:Collection",
5   "hydra:totalItems": 7,
6   "hydra:member": [
7     {
8       "@id": "/api/menu/food/1",
9       "@type": "Dish",
10      "id": 1,
11      "name": "Pepperoni pizza",
12      "ingredients": "Mozzarella, pepperoni, etc",
13      "price": "10.50",
14      "category": {
15        "@id": "/api/categories/1",
16        "@type": "Category",
17        "name": "Pizzas",
18        "description": "Delicious Italian pizzas"
19      }
20    },
21  ]
22 }
```

**Figure 26:** Authenticated GET request

That's all! Try to do the rest of the requests with the token. The POST, PUT and DELETE operations only will work with an admin user, while a normal registered user can use only the GET operations. Remember that the token will last only for 1 hour.

You can get all the code from the [GitHub repository](#).

[Full API Platform documentation](#)