

# UT4: Framework PHP SYMFONY



Symfony

Miguel Goyena

# ÍNDICE

1. **Symfony: ¿Que es?**
2. **Crear nuestro primer proyecto WEBAPP**
3. **Crear un CONTROLADOR**
4. **Crear un proyecto JSON REST API**
5. **SWAGGER: Documentación Open API**
6. **3 Pasos para la implementar REST API**
7. **Ejemplo de controlador GET: Path, Query**
8. **Ejemplo de controlador POST: Body binding y validaciones**
9. **Servicios/DI**
10. **Doctrine ORM**

# Symfony

## Que és

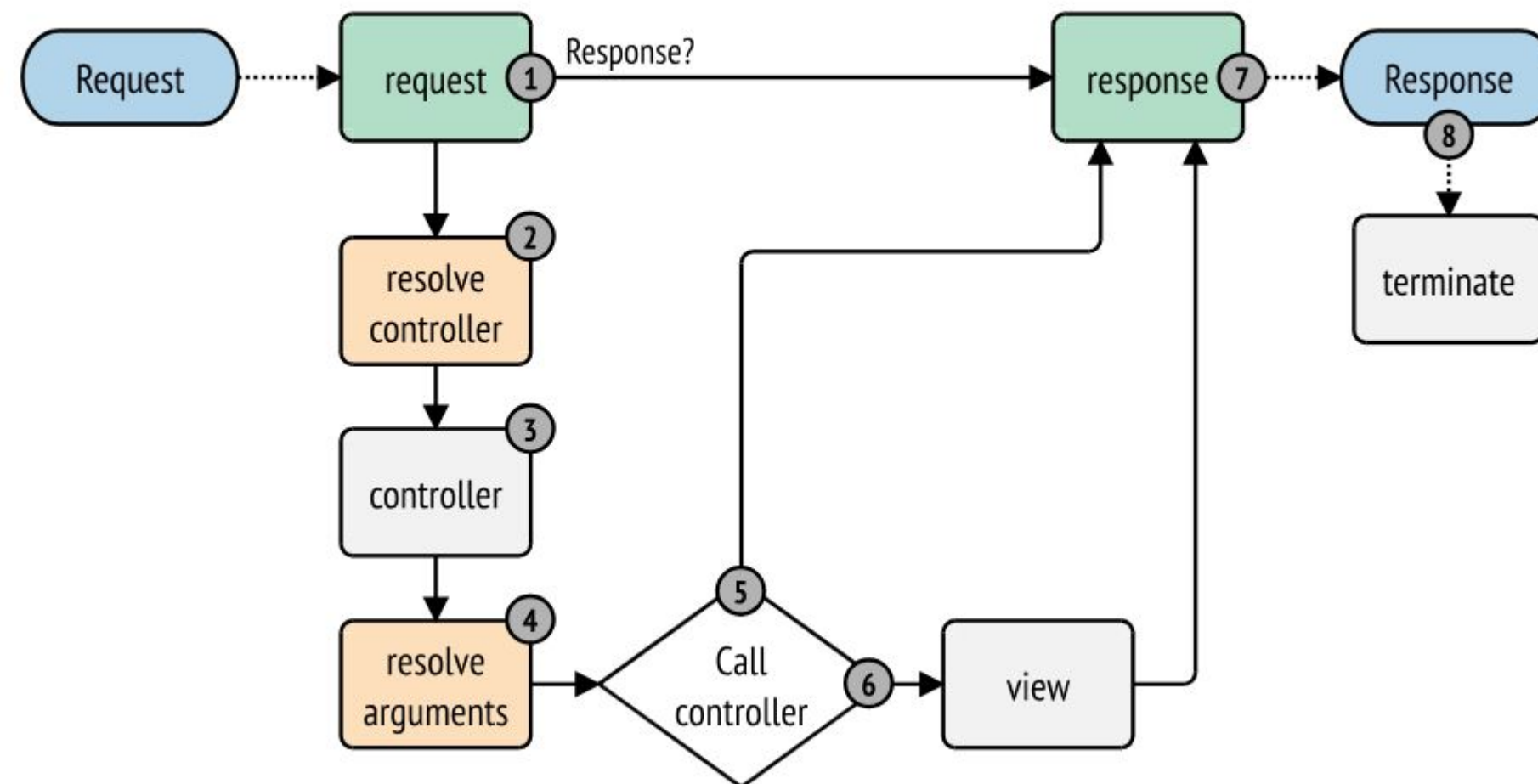
Symfony es un **framework** cuyo lenguaje de programación es **PHP** que contiene 3 propósitos:

- Darnos una **arquitectura basada en MVC** y una estructura de carpetas iniciales para que el desarrollo de una APP WEB sea algo mucho más ágil.
- Darnos un **FrontController** que enrute las peticiones HTTP a las funciones que nosotros creamos
- Darnos una serie de **librerías PHP** que nos hacen la vida más sencilla para hacer desarrollo WEB. Ejemplo un ORM o un sistema de plantillas TWIG para la Vista.

# Symfony FrontController

Este es el flujo de las peticiones que vienen a una APP Symfony:

EL **FrontController** es el que gestiona 1, 2, 4, 5 y 8. Por lo tanto nosotros solo tendremos que programar 3, 7 y View en el caso de APP con VIEW



# Symfony

## Librerías

Es un conjunto de librerías y utilidades para desarrollo WEB, como por ejemplo:

- Plantillas TWIG para desarrollo de la Vista. (Views)
- Doctrine/ORM para el acceso a la BBDD (DAO)
- Session, para la gestión de sesiones.
- Forms para crear formularios
- Validaciones de la petición HTTP
- Security (Authentication/Authorization)
- Services/DI
- Events & Messaging
- Scheduler
- Y más!!!!

# Symfony

## Arquitectura MVC y estructura de carpetas

Nos proponen 2 tipos de proyectos en Symfony, cada una de ellas tendrá una estructura de carpetas y unas librerías instaladas por defecto:

1. TIPO 1: **WEBAPP**. Contiene todo lo necesario para devolver contenido HTML en las respuestas a las peticiones HTTP. O sea contiene la V de MVC
2. TIPO 2: **HTTP Rest API**. Preparada para desarrollar una Rest API que devuelve formato JSON para las respuestas.

**Pero antes de explicar en más detalle vamos a empezar a desarrollar en Symfony!!!!**

# Crear nuestro primer proyecto WEBAPP

## Instalación

En este curso utilizaremos la **versión 7.1.x.** [Ver Manual Instalación](#)

Primero hay que cumplir una serie de requisitos para usar Symfony:

- Tener instalado PHP en versiones superiores a 8.2
- Tener instalado composer. Que es un el gestor de paquetes predominante en PHP.

Luego hay que instalar un programa CLI de Sistema Operativo

Al final estaremos preparados para utilizar symfony, chequearemos usando:

```
symfony check:requirements
```

# Crear nuestro primer proyecto WEBAPP

## Crear nuestro primer proyecto

Crearemos un proyecto SYMFONY de Tipo 1:

O sea con todo preparado para una APPWEB.

```
symfony new my_project_directory --version="7.1.*" --webapp
```

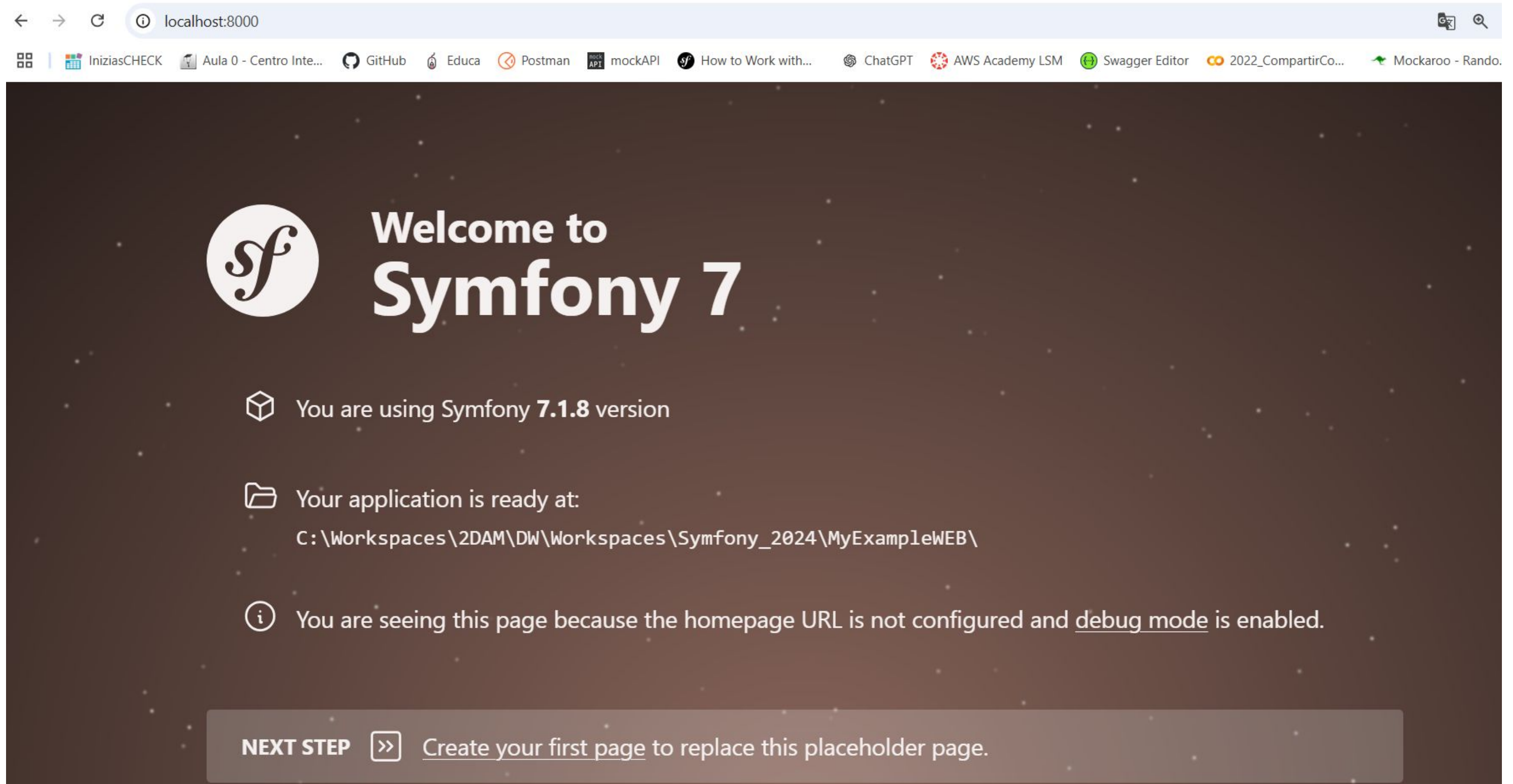
En cuanto se instala ya es posible levantar el servidor HTTP que nos sirve para desarrollar.

```
symfony server:start
```



# Crear nuestro primer proyecto WEBAPP

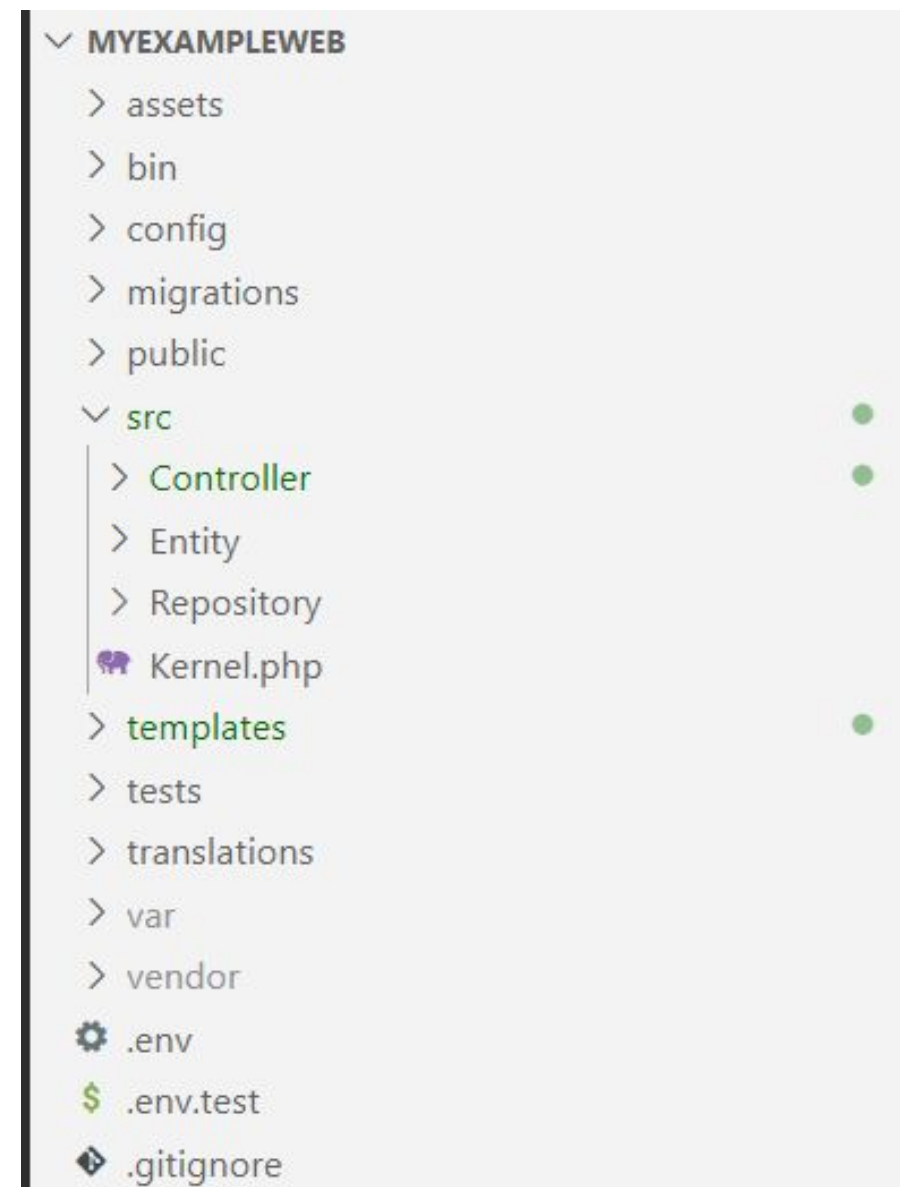
## Estamos listos!!!!: http://localhost:8000



# Crear nuestro primer proyecto WEBAPP

## Estructura de carpetas

Al crear una APP symfony nos propone una estructura de carpetas que nos ayudan a que nuestra APP tenga una puesta en marcha **rápida** y que sea **escalable**:

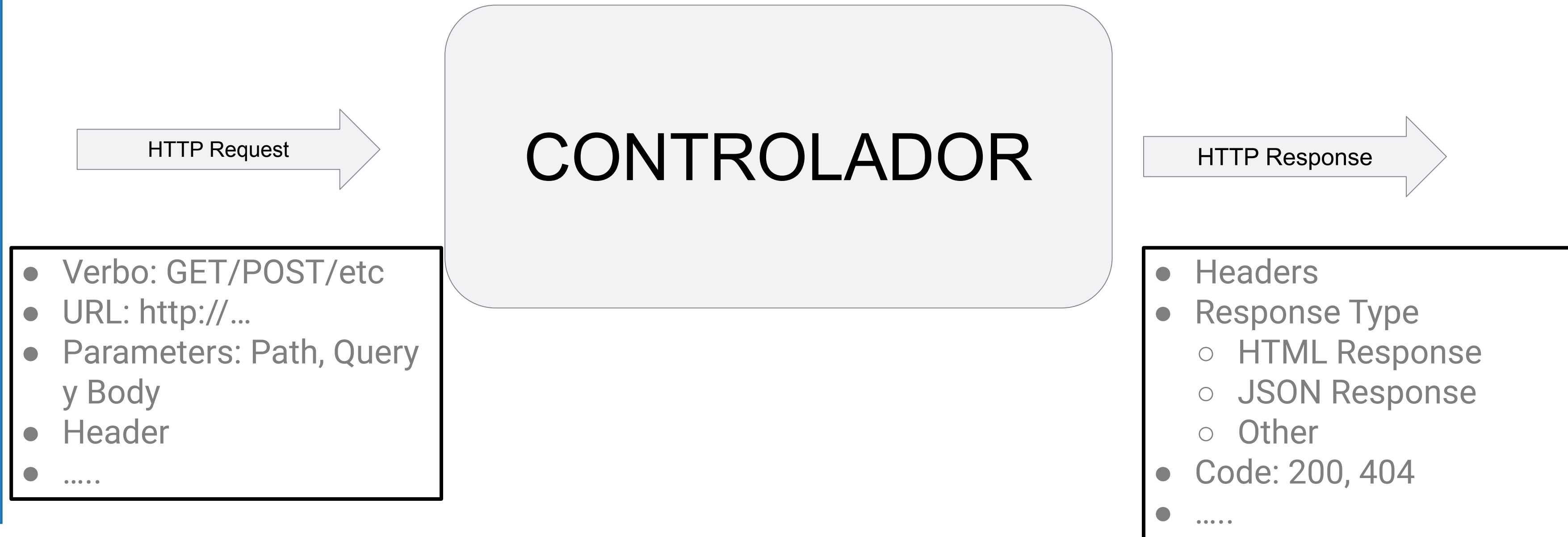


- **SRC**: Es la carpeta donde van a estar nuestras fuentes PHP
- **templates**: Para los templates de la Vista
- **assets**: Para los recursos públicos de la vista
- **config**: Toda la configuración tanto de la APP como de sus librerías
- **test**: Para los test unitarios
- **vendor**: la librerías utilizadas, usa composer como gestor de librerías
- **GIT**: Incluye un repositorio GIT
- **translation**: Traducciones de la vista
- **var**: Elementos que se borran como logs

# Crear un CONTROLADOR

## ¿Que es un controlador?

Un **controlador** es una **clase PHP** a la cual llegan los **request HTTP** y devuelve un **Response HTTP**



# Crear un CONTROLADOR

## Ejemplo código

Nos crearemos un controlador tipo Hola Mundo usando un comando CMD con bin/console para crear una estructura de controlador:

```
php bin/console make:controller HelloWorldController
```

```
class HelloWorldController extends AbstractController
{
    #[Route('/hello/world', name: 'app_hello_world')]
    public function index(): Response
    {
        return $this->render('hello_world/index.html.twig', [
            'controller_name' => 'HelloWorldController',
        ]);
    }
}
```

- Extiende de un controlador que tiene un montón de funcionalidades
- Route indica la URL al controlador
- Devuelve una Response en página HTML creada con twig, utilizando render

# Crear un proyecto JSON REST API

## Crear el proyecto

Crearemos un proyecto SYMFONY de Tipo 2:

O sea como punto de partida para la implementación de un HTTP API REST.

```
symfony new my_project_directory --version="7.1.*"
```

En cuanto se instala ya es posible levantar el servidor HTTP que nos sirve para desarrollar.

```
symfony server:start
```

# Crear un CONTROLADOR

## Ejemplo código

Nos crearemos un controlador tipo Hola Mundo usando un comando CMD con bin/console para crear una estructura de controlador:

```
php bin/console make:controller HelloWorldController
```

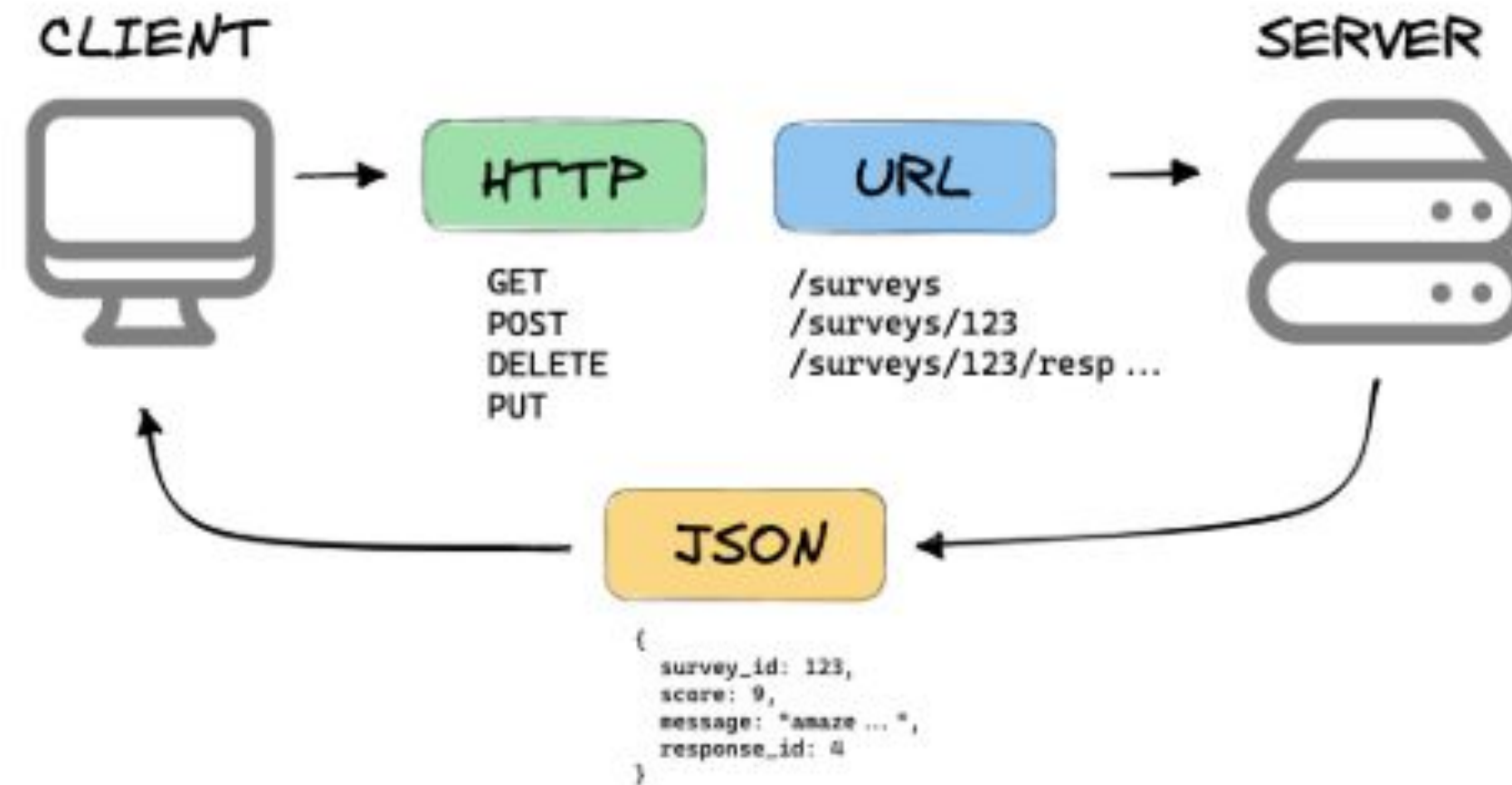
```
class HolaClaseController extends AbstractController
{
    #[Route('/hola/{clase}', name: 'app_hola_clase')]
    public function index($clase=""): JsonResponse
    {
        return $this->json([
            'message' => 'Welcome to your new controller!'.$clase,
            'path' => 'src/Controller/HolaClaseController.php',
        ]);
    }
}
```

- Route indica la URL al controlador
- Ejemplo de PathParameter: {clase}. Se convierte en un parámetro de index
- Devuelve un JsonResponse utilizando la función json



# Crear un CONTROLADOR A partir de AQUI

A partir de ahora siempre haremos APP de tipo 2.



**Pero, ¿Podemos especificar una rest api mediante algún standard?**

# SWAGGER: Documentación Open API

## Open API

**Open API** es una especificación de **cómo** se debe de definir un HTTP API REST:

¡¡¡Para que nos olvidemos de nada!!!

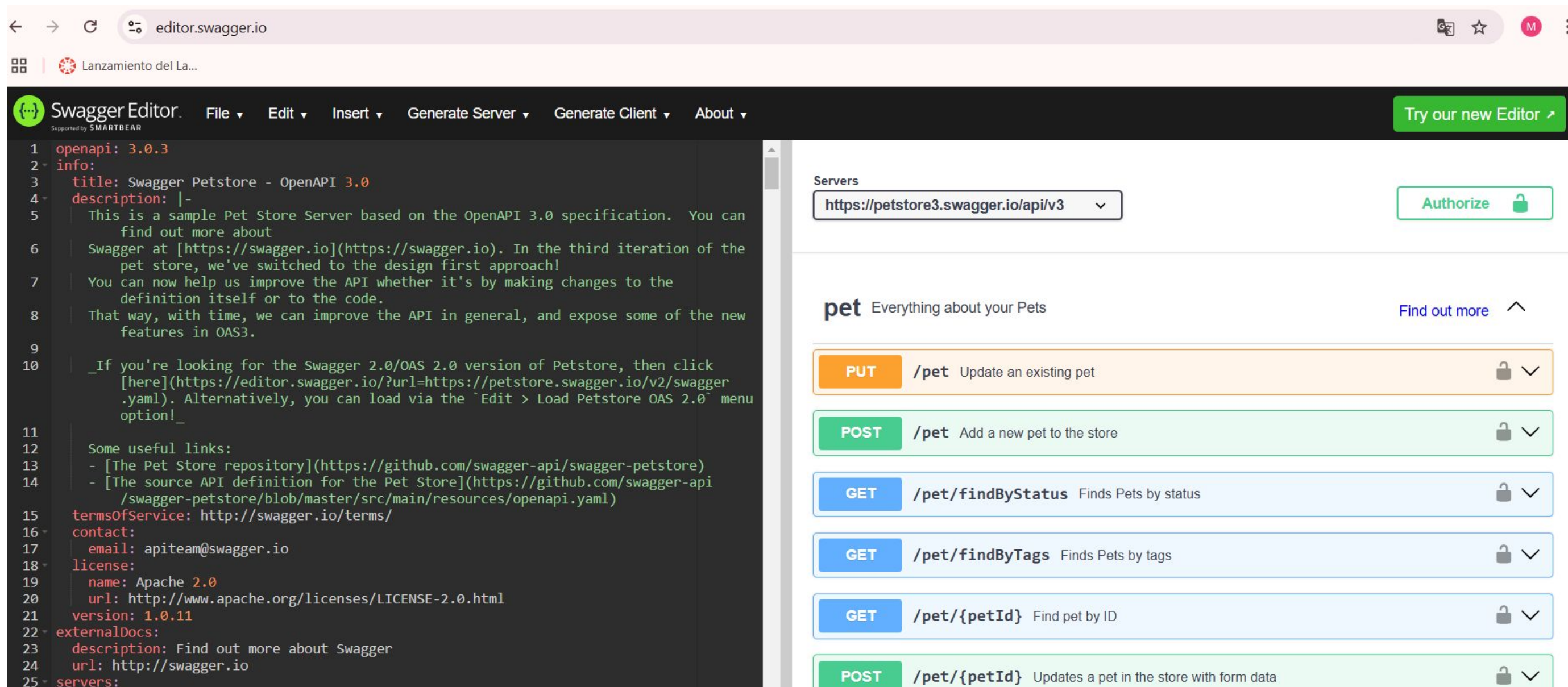
Se tienen que definir de un servidor:

- Las peticiones que se pueden realizar.
- Ejemplo de Request y Response
- Temas como la Autenticación y Autorización del API.



# SWAGGER: Documentación Open API

A partir de un fichero YAML, podemos definir. Y Swagger nos propone un editor convertir a una documentación más visual



The screenshot displays the Swagger Editor web application. The left pane shows a YAML file defining an OpenAPI 3.0 specification for a pet store. The right pane provides a visual representation of the API, including a list of endpoints with their methods, paths, and descriptions.

**YAML File Content:**

```
1 openapi: 3.0.3
2 info:
3   title: Swagger Petstore - OpenAPI 3.0
4   description: |-
5     This is a sample Pet Store Server based on the OpenAPI 3.0 specification. You can
6     find out more about
7     Swagger at [https://swagger.io](https://swagger.io). In the third iteration of the
8     pet store, we've switched to the design first approach!
9     You can now help us improve the API whether it's by making changes to the
10    definition itself or to the code.
11    That way, with time, we can improve the API in general, and expose some of the new
12    features in OAS3.
13
14    _If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click
15    [here](https://editor.swagger.io?url=https://petstore.swagger.io/v2/swagger
16    .yaml). Alternatively, you can load via the `Edit > Load Petstore OAS 2.0` menu
17    option!_
18
19    Some useful links:
20    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
21    - [The source API definition for the Pet Store](https://github.com/swagger-api
22    /swagger-petstore/blob/master/src/main/resources/openapi.yaml)
23  termsOfService: http://swagger.io/terms/
24  contact:
25    email: apiteam@swagger.io
26  license:
27    name: Apache 2.0
28    url: http://www.apache.org/licenses/LICENSE-2.0.html
29  version: 1.0.11
30  externalDocs:
31    description: Find out more about Swagger
32    url: http://swagger.io
33  servers:
```

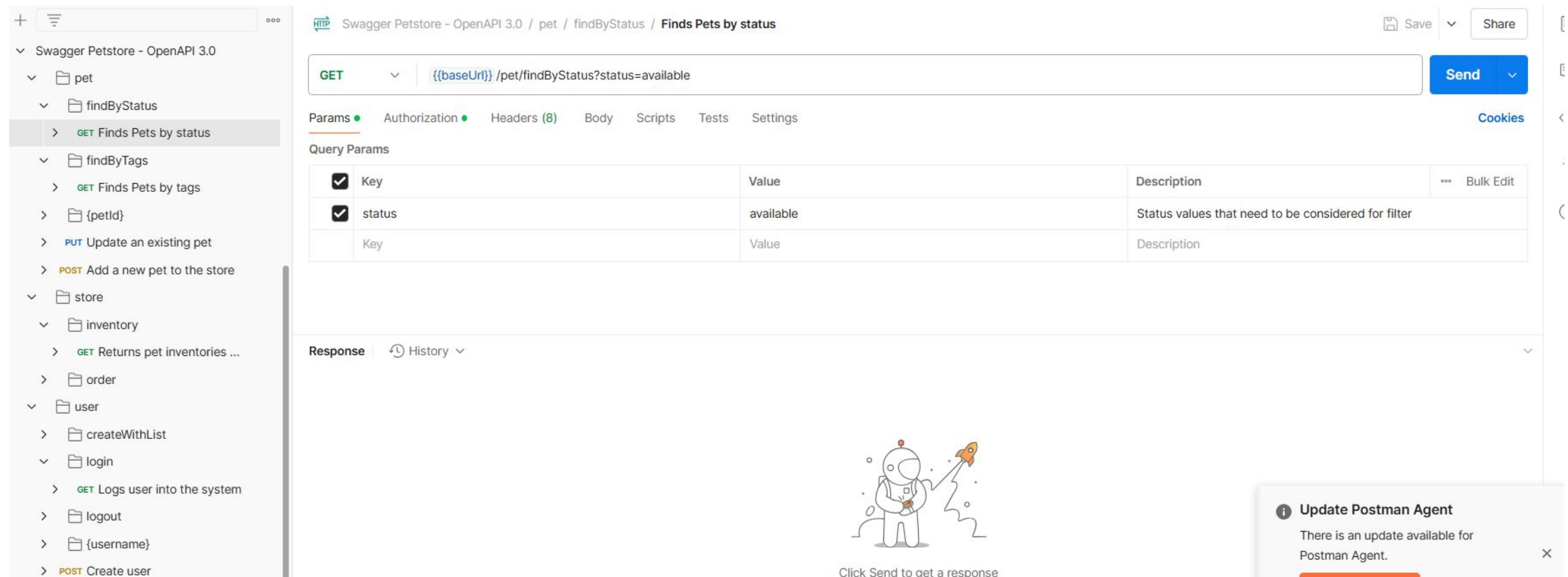
**Visual API Documentation:**

- Servers:** <https://petstore3.swagger.io/api/v3> (Authorize)
- pet** Everything about your Pets (Find out more)
- PUT /pet** Update an existing pet (Lock icon)
- POST /pet** Add a new pet to the store (Lock icon)
- GET /pet/findByStatus** Finds Pets by status (Lock icon)
- GET /pet/findByTags** Finds Pets by tags (Lock icon)
- GET /pet/{petId}** Find pet by ID (Lock icon)
- POST /pet/{petId}** Updates a pet in the store with form data (Lock icon)

# SWAGGER: Documentación Open API

## Postman

A partir de ese fichero YAML podemos con POSTMAN crear una colección para probar lo que vayamos implementando



Swagger Petstore - OpenAPI 3.0 / pet / findByStatus / Finds Pets by status


GET `{{baseUrl}}/pet/findByStatus?status=available` [Send](#) [Share](#)

Params • Authorization • Headers (8) Body Scripts Tests Settings [Cookies](#)

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	status	available	Status values that need to be considered for filter		
	Key	Value	Description		

Response [History](#)



Click Send to get a response

**Update Postman Agent**  
There is an update available for Postman Agent. [X](#)

## 3 Pasos para la implementar REST API

- Paso 1: Definir y Documentar con SWAGGER
- Paso 2: Implementar con SYMFONY
- Paso 3: Probar con POSTMAN

# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes GET sin params

Vamos a crear como ejemplo una API para gestionar Restaurantes de ElTenedor4V

1er Ejemplo /restaurants: Cómo conseguir toda la lista de restaurantes:

### GET SIN PARÉMETROS

Paso 1: Especificación SWAGGER.

```
- name: restaurants-types
description: Everything about the types of restaurantes
paths:
  /restaurantes:
    get:
      tags:
        - restaurants
      summary: Finds All Restaurants
      operationId: findAllRestaurants
      responses:
        '200':
          description: Returns OK in all restaurants
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Restaurantes'
        '400':
          description: Any problem in server
  /restaurants:
    get:
      tags:
        - restaurants
      summary: Finds All Restaurants
      operationId: findAllRestaurants
      parameters:
        - name: tipo
          in: query
          description: Tipos a filtrar
          required: false
          schema:
            type: string
      responses:
        '200':
          description: Return OK in all restaurants
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Restaurantes'
        '400':
          description: Any problem in server
  /restaurant-types:
```

#### restaurants

 Everything about restaurants  
**GET** /restaurants Finds All Restaurants  
**Parameters**  
No parameters  
**Responses**

Code	Description	Links
200	Returns OK in all restaurants Media type application/json Example Value <pre>{   "id": 10,   "name": "La Taggliatella",   "rest-type": {     "id": 10,     "name": "Oriental"   } }</pre>	No links
400	Any problem in server	No links



# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes GET sin params

### Paso 2: Implementación Symfony

Crearemos un fichero Controlador para Restaurants

```
php bin/console make:controller RestaurantsController
```

```
# [Route('/restaurantes', name: 'get_restaurantes')]
public function getRestaurantes(): JsonResponse
{
    return $this->json($this->restaurantes);
}
```

- Route con la URL
- El name solo es un nombre
- Devuelve un JsonResponse con la lista de restaurantes.
- Luego ya veremos cómo sacamos la lista de los restaurantes!!. Podría ser un array php ahora mismo

# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes GET sin params

### Paso 3: Pruebas con Postman

The screenshot shows a Postman interface for a GET request to `{{baseUrl}}/restaurantes`. The response is a 200 OK status with a JSON body containing two restaurant objects.

Query Params

Key	Value	Description
Key	Value	Description

Body

```
1 [
2   {
3     "id": 10,
4     "name": "La Taggliatella",
5     "rest-type": {
6       "id": 1,
7       "name": "Oriental"
8     }
9   },
10  {
11    "id": 11,
12    "name": "Ñam Sarasate",
13    "rest-type": {
14      "id": 2,
15      "name": "Italiano"
16    }
17  }
18 ]
```

- Petición GET a `/restaurants`
- baseUrl va a ser <http://localhost:8000>
- La respuesta es un 200 con un JSON bien formado

# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes: Get con Query Params

2º Ejemplo /restaurants?type=Italiano: Cómo conseguir toda la lista de restaurantes filtrado por tipo:

### GET CON QUERY PARAMS

#### Paso 1: Especificación SWAGGER.

```
5 description: Everything about the types of restaurantes
6 paths:
7   /restaurants:
8     get:
9       tags:
10        - restaurantes
11       summary: Finds ALL Restaurants
12       operationId: findAllRestaurants
13       parameters:
14        - name: tipo
15          in: query
16          description: Tipos a filtrar
17          required: false
18          schema:
19            type: string
20       responses:
21        '200':
22          description: Returns OK in all restaurants
23          content:
24            application/json:
25              schema:
26                type: array
27                items:
28                  $ref: '#/components/schemas/Restaurantes'
29        '400':
30          description: Any problem in server
31   /restaurant-types:
32     get:
33       tags:
34        - restaurantes
35       summary: Finds the types of restaurants available
36       operationId: findAllTypes
37       responses:
38        '200':
39          description: Returns OK in all the restaurant types
40          content:
41            application/json:
42              schema:
43                type: object
44                $ref: '#/components/schemas/RestaurantTypes'
45        '400':
46          description: Any problem in server
47   /restaurant-types/{id}:
48     get:
```

**restaurants** Everything about restaurants

**GET** /restaurants Finds ALL Restaurants [Try it out](#)

**Parameters**

Name	Description
tipo	Tipos a filtrar
string	<input type="text" value="tipo"/>
(query)	

**Responses**

Code	Description	Links
200	Returns OK in all restaurants	No links

Media type: **application/json**

Controls Accept header.

Example Value | Schema

```
{
  "id": 10,
  "name": "La Tagliatella",
  "rest-type": {
    "id": 10,
    "name": "Oriental"
  }
}
```

# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes: Get con Query Params

### Paso 2: Implementación Symfony

Reutilizamos el controlador antes creado

```
#[Route('/restaurants', name: 'get_restaurants')]

public function getRestaurants([MapQueryParameter] string
$tipo): JsonResponse
{
    // Buscamos por el tipo
    if ($tipo == "Italiano") {
        return $this->json($this->restaurantesItalianos);
    }
    else {
        return $this->json($this->restaurantes);
    }
}
```

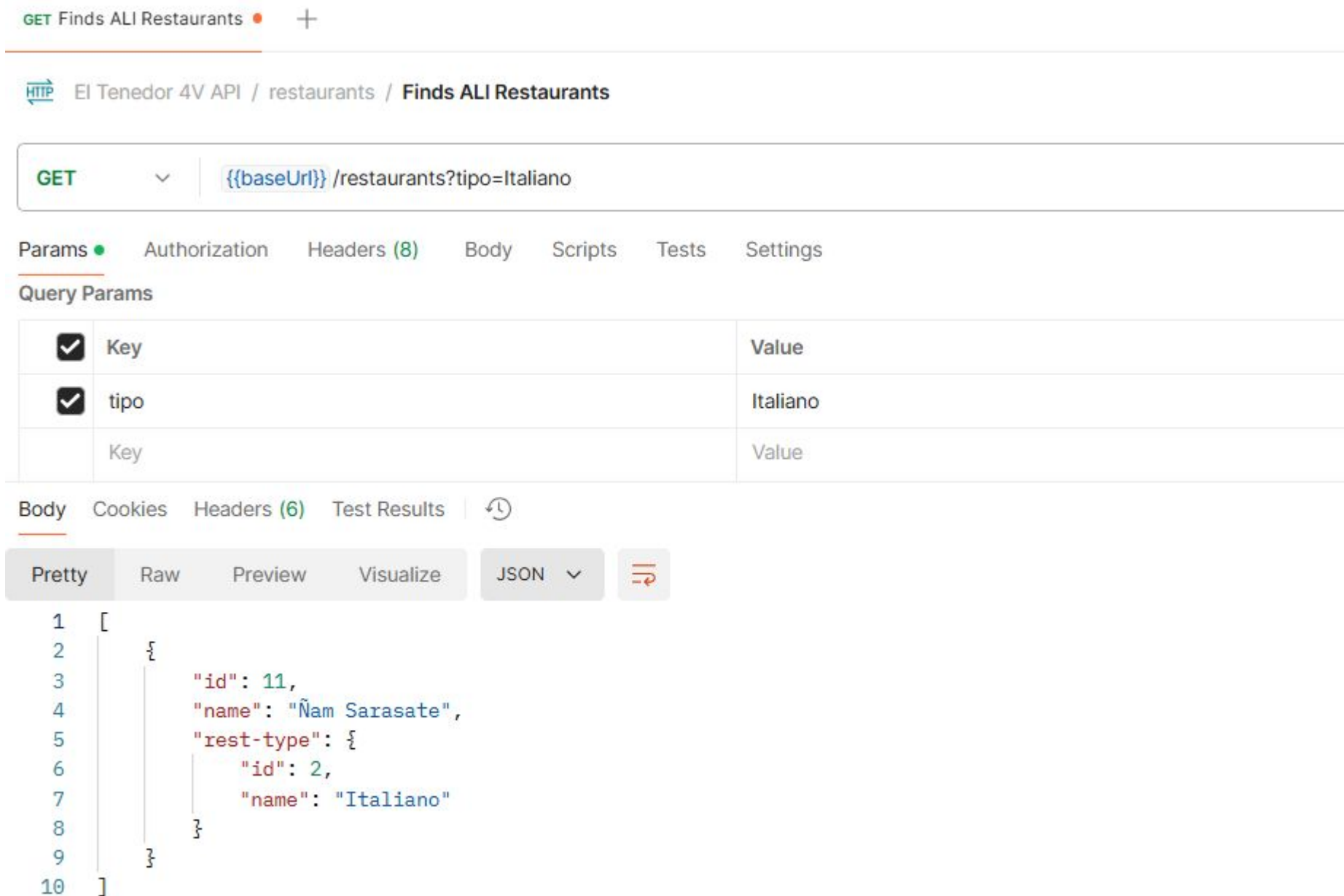
- Route con la URL
- MapQueryParameter por cada parametro
- Devuelve un JsonResponse con la lista de restaurantes.



# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes: Get con Query Params

### Paso 3: Pruebas con Postman



- Petición GET a /restaurants
- Se le pone como parametro de query con el ? delante de la URL
- La respuesta es un 200 con un JSON bien formado

# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes: Get con Path Params

3er Ejemplo /restaurants/{id}: Cómo conseguir la información de un restaurante

### GET CON PATHPARAMS

#### Paso 1: Especificación SWAGGER.

```
60-         content:
61-           application/json:
62-             schema:
63-               type: object
64-               $ref: '#/components/schemas/RestaurantTypes'
65-         '400':
66-           description: Any problem in server
67- /restaurant/{id}:
68-   get:
69-     tags:
70-       - restaurants
71-     summary: Finds a concrete restaurant
72-     operationId: findRest
73-     parameters:
74-       - name: id
75-         in: path
76-         description: El ID que quiero obtener
77-         required: true
78-         schema:
79-           type: integer
80-     responses:
81-       '200':
82-         description: Returns OK with a single restaurant
83-         content:
84-           application/json:
85-             schema:
86-               type: object
87-               $ref: '#/components/schemas/Restaurantes'
88-       '400':
89-         description: Any problem in server
90- /restaurant-types/{id}:
91-   get:
92-     tags:
93-       - restaurants
94-     summary: Finds a concrete restaurant type
95-     operationId: findByType
96-     parameters:
97-       - name: id
98-         in: path
99-         description: El ID que quiero obtener
100-        required: true
101-        schema:
102-          type: integer
103-     responses:
104-       '200':
105-         description: Returns OK with a single restaurant
106-         content:
```

GET /restaurant-types Finds the types of restaurants available

GET /restaurant/{id} Finds a concrete restaurant

Parameters

Try it out

Name	Description
id * required	El ID que quiero obtener
integer	id
(path)	

Responses

Code	Description	Links
200	Returns OK with a single restaurant	No links
400	Any problem in server	No links

Media type: application/json

Example Value | Schema

```
{
  "id": 10,
  "name": "La Tagliatella",
  "rest-type": {
    "id": 10,
    "name": "Oriental"
  }
}
```

# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes: Get con Query Params

### Paso 2: Implementación Symfony

Reutilizamos el controlador antes creado

```
# [Route('/restaurants/{id}', name: 'get_restaurants_by_id')]
public function getRestaurantsById(string $id): JsonResponse
{
    if ($id < sizeof($this->restaurantes)){
        return $this->json($this->restaurantes[$id]);
    }
    else{
        return $this->json(["error" => "No tengo el ID que me pides"], 400);
    }
}
```

- Route con la URL, entre {} el parámetro de URL, en este caso un id
- Un parámetro al método getRestaurantsById
- Devuelve un JsonResponse con el restaurante en concreto o con un JSON que da error, con código 400

# Ejemplo de controlador GET: Path, Query, Binding y Validaciones

## Ejemplo Restaurantes: Get con Query Params

### Paso 3: Pruebas con Postman

HTTP El Tenedor 4V API / restaurants / Finds One Restaurant

GET {{baseUrl}}/restaurants/0

Params Authorization Headers (8) Body Scripts Tests Settings

Query Params

Key	Value
Key	Value

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 10,
3   "name": "La Taggliatella",
4   "rest-type": {
5     "id": 1,
6     "name": "Oriental"
7   }
8 }
```

- Petición GET a /restaurants/0
- La respuesta es un 200 con un JSON bien formado.
- Pero podría haber sido un 400 con el error en concreto

Body Cookies Headers (6) Test Results

400 Bad Request

Pretty Raw Preview Visualize JSON

```
1 {
2   "error": "No tengo el ID que me pides"
3 }
```

# Ejemplo POST

## Definición

También un controlador puede recibir peticiones de tipo POST PUT o DELETE

- **POST:** Nuevo
- **PUT:** Actualización
- **DELETE:** Borrado

Este tipo de verbos se suelen utilizar con datos BODY, o sea se envía un JSON como parámetro.



# Ejemplo de controlador POST

## Ejemplo Restaurantes

4º Ejemplo POST /restaurants: Cómo introducir un nuevo restaurante

## POST

### Paso 1: Especificación SWAGGER.

The image shows the Swagger Editor interface with the OpenAPI specification for a POST endpoint. The left pane displays the raw YAML code, and the right pane shows the rendered UI.

**YAML Specification (Left Pane):**

```
21 summary: Finds All Restaurants
22 operationId: findAllRestaurants
23 parameters:
24   - name: tipo
25     in: query
26     description: Tipos a filtrar
27     required: false
28     schema:
29       type: string
30 responses:
31   '200':
32     description: Returns OK in all restaurants
33     content:
34       application/json:
35         schema:
36           type: array
37           items:
38             $ref: '#/components/schemas/Restaurantes'
39   '400':
40     description: Any problem in server
41 post:
42   tags:
43     - restaurants
44   summary: Add a new restaurant
45   description: Add a new restaurant
46   operationId: addRestaurant
47   requestBody:
48     description: Create a new Restaurant
49     content:
50       application/json:
51         schema:
52           $ref: '#/components/schemas/RestaurantesNew'
53     required: true
54   responses:
55     '200':
56       description: Successful operation
57       content:
58         application/json:
59           schema:
60             $ref: '#/components/schemas/Restaurantes'
61     '400':
62       description: Invalid input
63     '422':
64       description: Validation exception
65 /restaurant-types:
66 get:
67   tags:
68     - restaurants
69   summary: Finds the types of restaurants available
70   operationId: findAllTypes
71   responses:
72     '200':
73       description: Returns OK in all the restaurant types
74       content:
75         application/json:
76           schema:
77             type: object
78             $ref: '#/components/schemas/RestaurantTypes'
```

**Rendered UI (Right Pane):**

**POST /restaurants** Add a new restaurant

Add a new restaurant

**Parameters** Try it out

No parameters

**Request body** required application/json

Create a new Restaurant

Example Value Schema

```
{
  "id": 10,
  "name": "La Tagliatella",
  "rest-type": 23
}
```

**Responses**

Code	Description	Links
200	Successful operation	No links
400	Invalid input	No links

# Ejemplo de controlador POST

## Ejemplo Restaurantes

### Paso 2: Implementación Symfony

### Reutilizamos el controlador antes creado

```
#[Route('/restaurants', name: 'post_restaurants', methods: ['POST'])]
public function newRestaurants(Request $request): JsonResponse
{
    // Recuperamos del request el Body
    $jsonBody = $request->getContent(); // Obtiene el cuerpo como texto
    $data = json_decode($jsonBody, true); // Lo decodifica a un array asociativo

    // Manejo de errores si el JSON no es válido
    if (json_last_error() !== JSON_ERROR_NONE) {
        return $this->json(['error' => 'JSON inválido'], 400);
    }

    // Inserto el objeto
    array_push($this->restaurantes, $data);

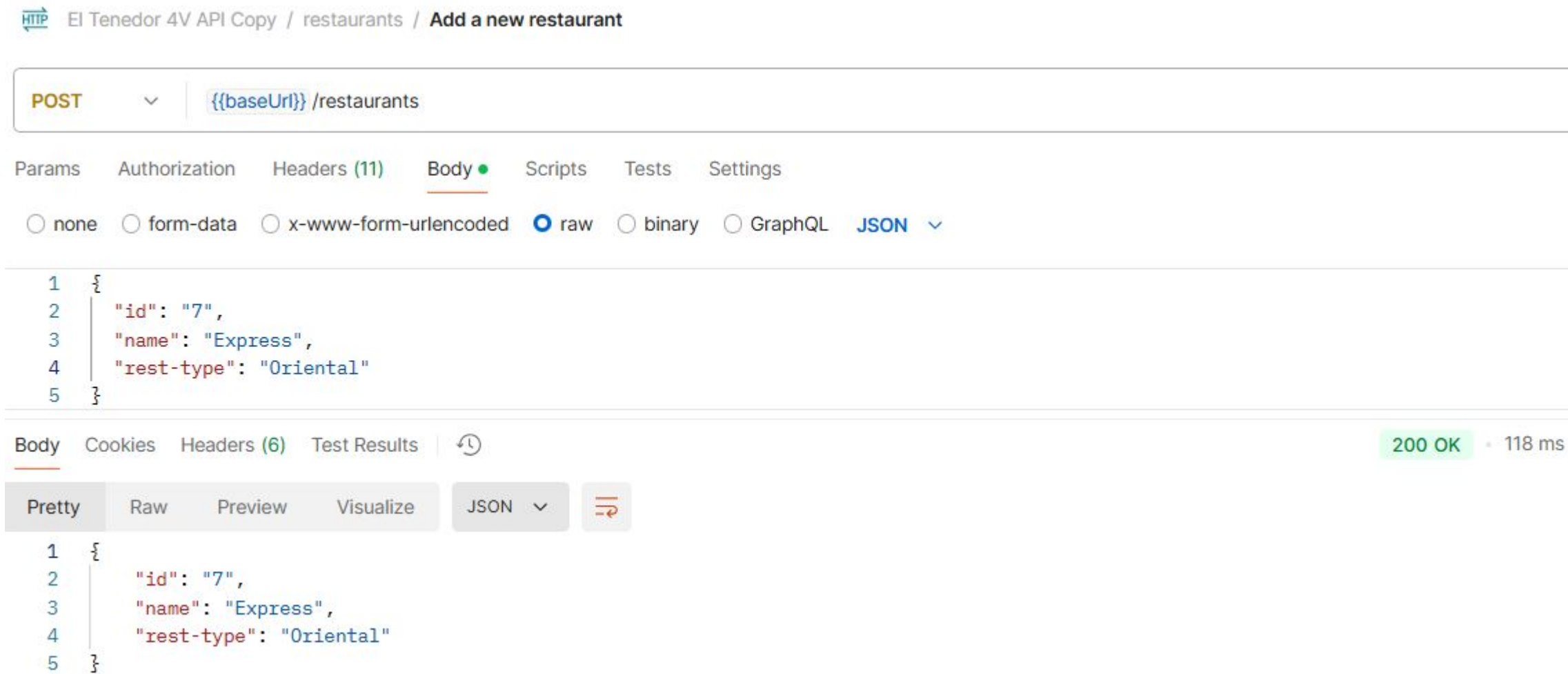
    //Contesto
    return $this->json($this->restaurantes[sizeof($this->restaurantes)-1]);
}
```

- Route con la URL, METHOD, indicamos que es un POST, revisar entonces el GET
- Recupero todo el Request, y dentro el Content es lo que llamamos Body
- Hacemos lo que tengamos que hacer y devolvemos un JSON Response

# Ejemplo de controlador POST

## Ejemplo Restaurantes

### Paso 3: Pruebas con Postman



- Petición POST /restaurants
- Se le pasa un Body con el JSON de entrada
- La respuesta es un 200 con un JSON bien formado, que es el resultamos que queremos



# Ejemplo de controlador POST

## Mapping y validaciones

Es una buena praxis crear un objeto de modelo para las peticiones POST, así tenemos el JSON ya en un objeto manipulable.

### Paso 1: Nos creamos un modelo para nuestros nuevos restaurantes

```
namespace App\Model;

class RestaurantNewDTO
{
    public function __construct(
        public int $id,
        public string $name,
        public int $resType) {}
}
```

- En el constructor pondremos nuestros atributos del modelo
- Crearemos una carpeta de modelos
- Es para la entrada de los datos por lo tanto resType es un entero

# Ejemplo de controlador POST

## Mapping y validaciones

Paso 2: Hacemos que el controlador utilice ese objeto, pero primero hay que bajarse el serializador utilizando el composer

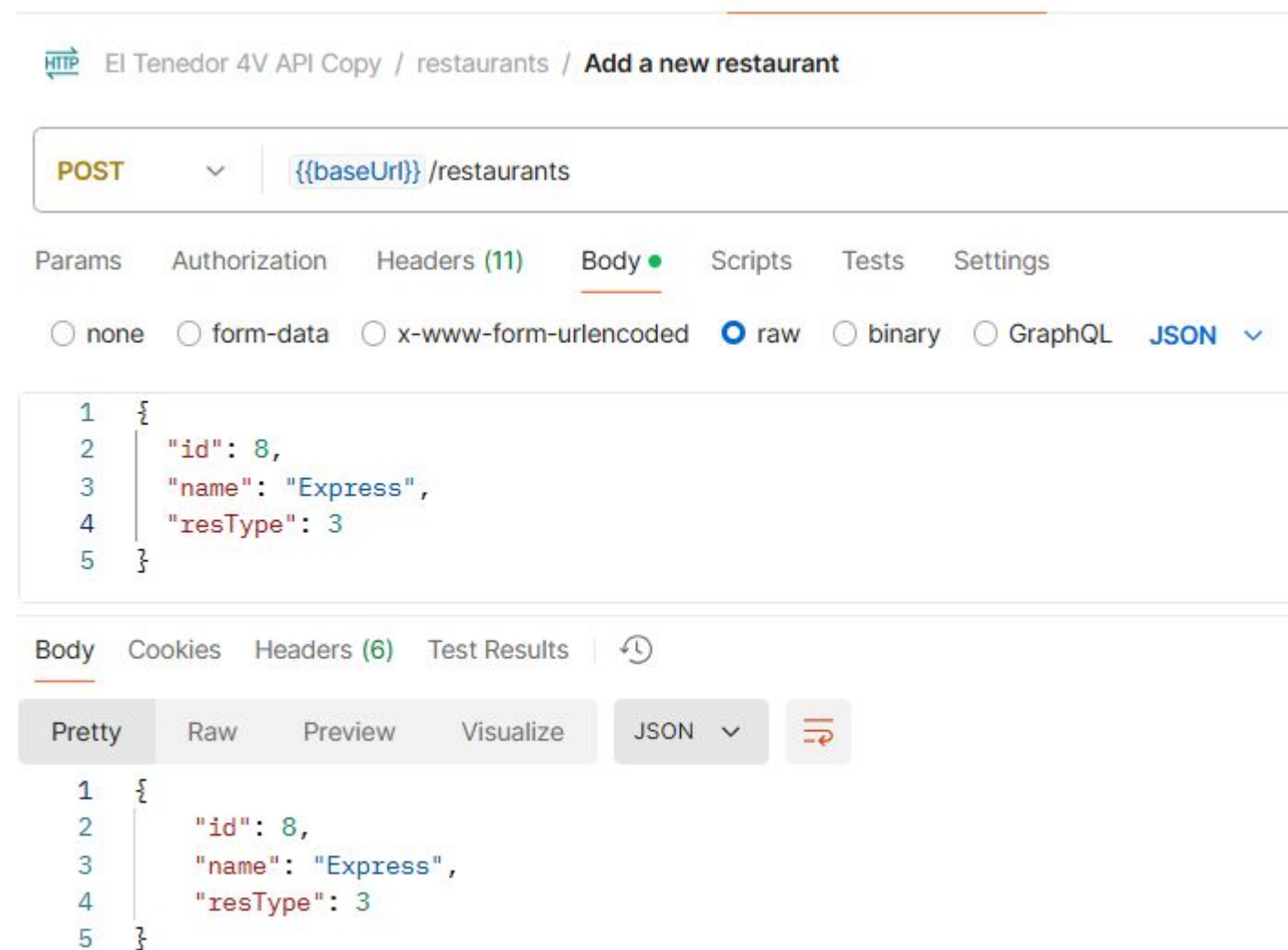
```
composer require symfony/serializer-pack
```

```
#[Route('/restaurants', name: 'post_restaurants', methods: ['POST'],  
format: 'json')]  
  
public function newRestaurants(#[MapRequestPayload] RestauranteNewDTO  
$restaurantNewDto): JsonResponse  
{  
    // Inserto el objeto  
    array_push($this->restaurantes, $restaurantNewDto);  
  
    //Contesto  
    return  
    $this->json($this->restaurantes[sizeof($this->restaurantes)-1]);  
}
```

- MapRequestPayload, indica que hay que utilizar un objeto RestauranteNew a mapear
- Por lo tanto el resto ya es mucho más sencillo
- Hemos puesto que el format es JSON y así damos los errores en

# Ejemplo de controlador POST Mapping y validaciones

Paso 3: Para probarlo hemos tenido que hacer algún que otro cambio en el POSTMAN



- res-type pasa a resType, para que cumpla la especificación de nombrado de PHP
- He cambiado el tipo y el id para que sean enteros
- El resultado es idéntico

# Ejemplo de controlador POST

## Mapping y validaciones

Paso 4: Vamos a ponerle Validaciones al objeto!!!. Primero tenemos que meter el paquete de validadores con composer

```
composer require symfony/validator
```

```
class RestauranteNewDTO
{
    public function __construct(
        #[Assert\NotBlank]
        public int $id,
        #[Assert\NotBlank(message:"El nombre es obligatorio")]
        public string $name,
        #[Assert\NotBlank]
        public int $resType) {}
}
```

- Se pueden poner un montón de Asserts diferentes, incluso los nuestros propios
- Se pueden poner propiedades como un mensaje en concreto
- No os olvideis de importar

# Ejemplo de controlador POST

## Mapping y validaciones

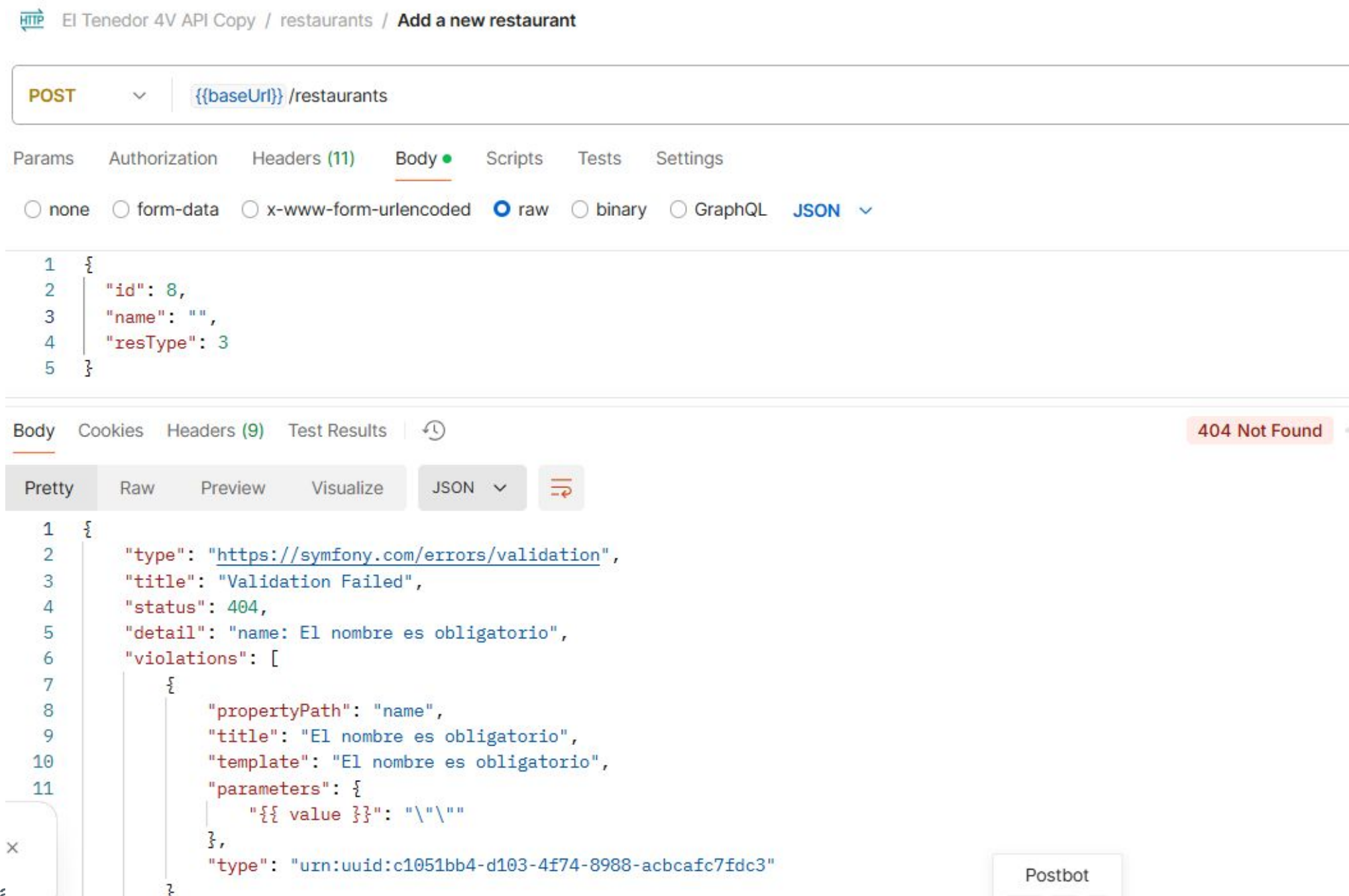
Paso 5: En el controlador podemos manejar las validaciones o simplemente podemos dejar el trabajo a Symfony

```
public function newRestaurants ([MapRequestPayload(
    acceptFormat: 'json',
    validationFailedStatusCode: Response::HTTP_NOT_FOUND
)] RestauranteNewDTO $restauranNewtDto): JsonResponse
```

- Podemos ponerle el formato de salida o decidir qué código de respuesta

# Ejemplo de controlador POST Mapping y validaciones

## Paso 6: Veremos como queda en POSTMAN



- Vemos que devuelve un error en JSON
- Que también devuelve un 404
- Recuerda que podríamos definir nuestros propios errores en el controlador

## Ejemplo de controlador POST Mapping y validaciones

- Hay muchas más posibilidades en los validadores y es todo un mundo!!!. Incluso se pueden crear los propios servicios de validación
- Se puede utilizar también en los QueryParameters del GET



## Ejemplo de controlador POST

### Mapping y validaciones: **Ejercicio**

Podemos crear un modelo como RestaurantDTO, y así utilizarlo para todas las respuestas GET de la API.

También podemos crear un modelo QueryRestaurantsDTO con los campos para las búsquedas de los restaurantes, como por ejemplo por Tipo



## Servicios/DI

### ¿Que son?

Los servicios son las clases PHP que existen en tu proyecto symfony que ofrecen una funcionalidad específica y no son Controladores.

El propio symfony ya viene con un montón de servicios dentro, como el servicio LoggerInterface. Para obtener la lista de servicios disponibles:

```
php bin/console debug:autowiring
```

# Servicios/DI

## LoggerInterface

El servicio LoggerInterface sirve para escribir logs en las APPS Symfony.

```
public function __construct(public LoggerInterface $logger)
{
    #[Route('/restaurants', name: 'get_restaurants', methods:['GET'])]
    public function getRestaurants(#[MapQueryParameter] string $tipo):
    JsonResponse
    {
        $this->logger->info("Quiero los restaurantes del tipo: ".$tipo);
        // Buscamos por el tipo
        if ($tipo == "Italiano") {
            return $this->json($this->restaurantesItalianos);
        }
        else{
            return $this->json($this->restaurantes);
        }
    }
}
```

- Para recuperarlo solo hay que ponerlo como parámetro del constructor o de una función en concreto
- Luego se utiliza el objeto: \$logger

# Servicios/DI

## Podemos crearlos

Podemos crear nuestro propios servicios, poniéndolos en la carpeta services. Por ejemplo vamos a crear un servicio de Restaurantes

```
namespace App\Service;

use App\Model\RestauranteNewDTO;

class RestaurantsService
{
    .....

class RestaurantsController extends AbstractController
{

    public function __construct(private LoggerInterface $logger, private
    RestaurantsService $restaurantsService)

    {}
```

- Lo creamos en la carpeta de Service y se autocarga!!!

Servicios/DI

Podemos crearlos: EJERCICIO

## **EJERCICIO:**

Pasaremos toda la lógica de los restaurantes del controlador al servicio.

Luego en el controlador lo utilizamos al invocarlo en el constructor!!

# Doctrine ORM

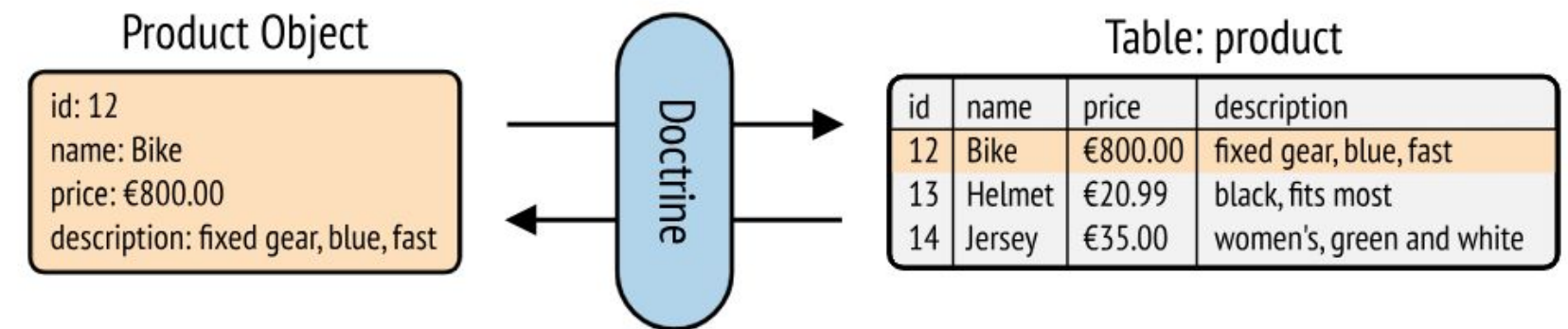
## ¿Que es?

Cuando nuestras APP symfony necesitan acceso a una BBDD, podemos utilizar Doctrine para que nos haga de pasarela de modelos relaciones (tablas) a modelos OO (clases).

Sirve para cualquier motor de BBDD

Es un productor de DAO!!!!.

Primero hay que descargarnos doctrine y su maker utilizando composer



```
composer require symfony/orm-pack
composer require --dev symfony/maker-bundle
```

# Doctrine ORM

## ¿Cómo configurarlo?

Paso 1: Tenemos que configurar la BBDD a la que pegar.

Nos habrá creado en .env con esta configuración

```
# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
DATABASE_URL="mysql://root@127.0.0.1:3306/eltenedor4v?serverVersion=10.4.
32-MariaDB&charset=utf8mb4"
# DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=16&charset=utf8"
```

Aquí configuramos nuestra BBDD: usuario, password, puerto, etc...

**OJO, que la versión de la BBDD en .env me ha dado problemas y he tenido que utilizar la que tengo en mi XAMP (select @@version)**



# Doctrine ORM

## ¿Cómo configurarlo?

Paso 2: Crearemos la BBDD para nuestro proyecto

```
php bin/console doctrine:database:create
```

# Doctrine ORM

## Creamos Entidades

Una **Entidad** es un modelo que vamos a llevar a la BBDD, por lo tanto se convertirá posiblemente en una tabla de BBDD.

En nuestro caso sería **Restaurant**, o sea nuestros restaurantes.

```
php bin/console make:entity
```

Class name of the entity to create or update (e.g. VictoriousPizza):

> Restaurant

created: src/Entity/Restaurant.php

created: src/Repository/RestaurantRepository.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

.....

# Doctrine ORM

## Creamos Entidades

No hace falta crear el ID, porque se da por supuesto que hay un id por cada entidad. Se crea la clase Restaurant(Modelo) y el Respository (DAO)

```
<?php
```

```
namespace App\Entity;

use App\Repository\RestaurantRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: RestaurantRepository::class)]
class Restaurant
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $name = null;

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

```
<?php
```

```
namespace App\Repository;

use App\Entity\Restaurant;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

/**
 * @extends ServiceEntityRepository<Restaurant>
 */
class RestaurantRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Restaurant::class);
    }

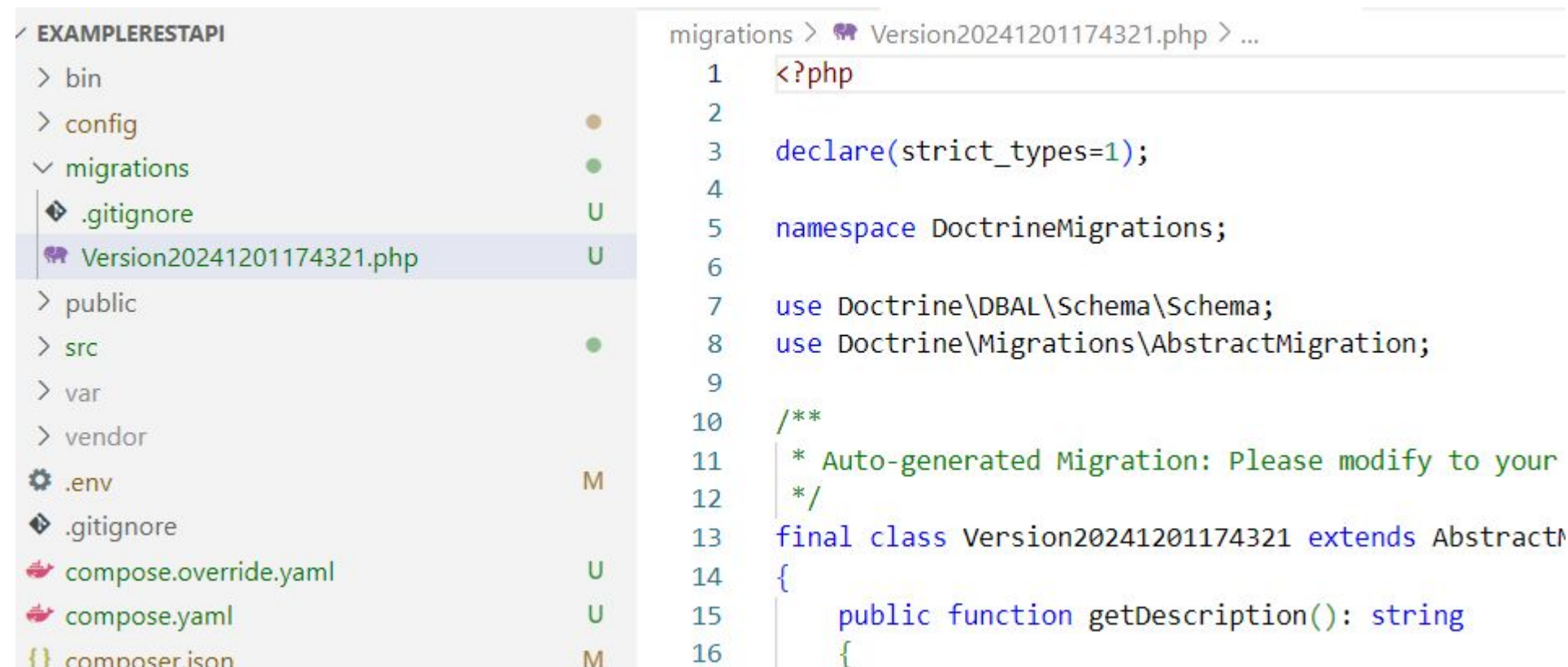
    /**
     * @return Restaurant[] Returns an array of Restaurant objects
     */
    public function findByExampleField($value): array
    {
        return $this->createQueryBuilder('r')
            ->andWhere('r.exampleField = :val')
            ->getQuery()
            ->getResult();
    }
}
```

# Doctrine ORM

## Creamos los scripts de BBDD

A partir de la Entidad creada, se pueden crear automáticamente los script de BBDD

```
php bin/console make:migration
```



The screenshot shows a code editor with a file explorer on the left and a migration script being created in the migrations directory.

**File Explorer (Left):**

- EXAMPLERESTAPI
  - bin
  - config
  - migrations
    - .gitignore
    - Version20241201174321.php
  - public
  - src
  - var
  - vendor
  - .env
  - .gitignore
  - compose.override.yaml
  - compose.yaml
  - composer.json

**Migration Script (Right):**

```
migrations > Version20241201174321.php > ...
1  <?php
2
3  declare(strict_types=1);
4
5  namespace DoctrineMigrations;
6
7  use Doctrine\DBAL\Schema\Schema;
8  use Doctrine\Migrations\AbstractMigration;
9
10 /**
11  * Auto-generated Migration: Please modify to your
12  */
13 final class Version20241201174321 extends AbstractMigration
14 {
15     public function getDescription(): string
16     {
```

- Crea en la carpeta migration un script con la fecha actual para las actualizaciones de las nuevas entidades
- Por lo tanto tendremos versionado en los cambios de BBDD



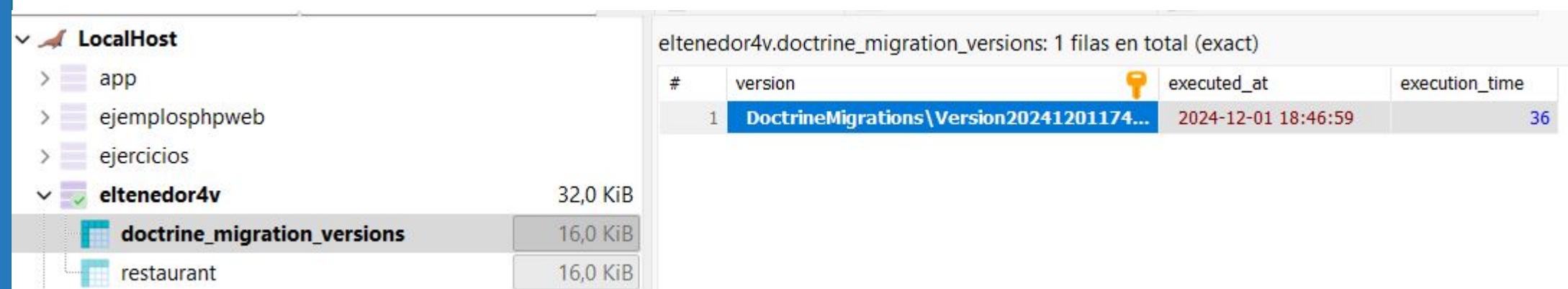
# Doctrine ORM

## Ejecutamos el cambio en BBDD

Ahora podemos ejecutar los cambios sobre la BBDD

```
php bin/console doctrine:migrations:migrate
```

- Nos ha creado la tabla Restaurant
- Y también nos ha generado una tabla doctrine\_migrations\_versions que mantiene nuestro historial de cambios



The screenshot shows a database interface with a left sidebar and a main table view. The sidebar lists 'LocalHost' with folders 'app', 'ejemplosphpweb', 'ejercicios', and 'eltenedor4v'. Under 'eltenedor4v', there are three tables: 'doctrine\_migrations\_versions' (16,0 KiB), 'restaurant' (16,0 KiB), and 'ejercicios' (32,0 KiB). The main view displays the 'doctrine\_migrations\_versions' table with the title 'eltenedor4v.doctrine\_migrations\_versions: 1 filas en total (exact)'. The table has four columns: '#', 'version', 'executed\_at', and 'execution\_time'. There is one row with the following data: '# 1', 'version DoctrineMigrations\Version20241201174...', 'executed\_at 2024-12-01 18:46:59', and 'execution\_time 36'.

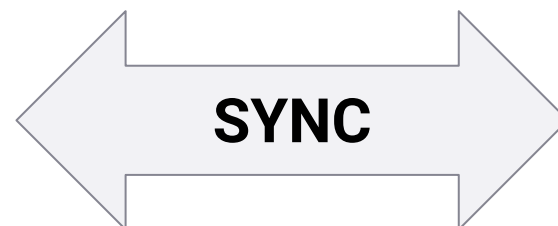
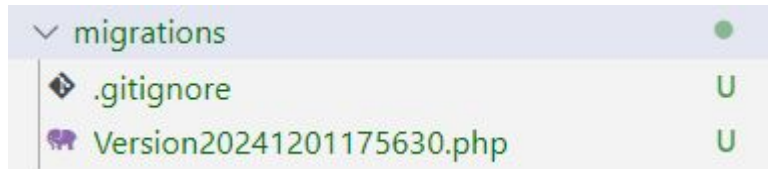
#	version	executed_at	execution_time
1	DoctrineMigrations\Version20241201174...	2024-12-01 18:46:59	36

# Doctrine ORM

## Ejecutamos el cambio en BBDD

OJO: Siempre tenemos que tener sincronizados la carpeta migrations con la tabla de BBDD doctrine\_migrations\_version. Se pueden dar ciertas situaciones:

- Borramos la tabla de BBDD, entonces si volvemos a ejecutar el migrations\_migrate entonces ejecutará todos los scripts
- Borramos la carpeta de scripts. Entonces perdemos el historial de cambios en nuestro proyecto y es también necesario borrar las tablas de BBDD para que funcione todo de nuevo correcto. Empezaremos de nuevo





# Doctrine ORM

## Cómo utilizamos los repositorios: READ

Ahora vamos a utilizar los repositorios con nuestro servicio: RestaurantService

```
public function __construct(private EntityManagerInterface  
$entityManager) {}  
  
public function getAllRestaurants(): array  
{  
    return  
    $this->entityManager->getRepository(Restaurant::class)->findAll();  
}
```

- En el constructor inyectamos el EntityManagerInterface
- En el método que queremos llamamos al repositorio para ejecutar los métodos que queremos

# Doctrine ORM

## Cómo utilizamos los repositorios: CREATE

Ahora crearemos un nuevo restaurante

```
public function addRestaurant(RestauranteNewDTO $newRestaurant):  
    RestauranteNewDTO  
    {  
        // Creamos la entidad restaurante  
        $newRestaurantEntity = new Restaurant();  
        $newRestaurantEntity->setName($newRestaurant->name);  
  
        // Le dices a Doctrine que quieres persistir el objeto,, todavia no  
        // hace nada  
        $this->entityManager->persist($newRestaurantEntity);  
  
        // Aqui es donde confirmas, asi tienes el concepto de transaccion!!!!  
        $this->entityManager->flush();  
  
        // Fijate que se ha cambiado la entidad con el ID nuevo  
        $newRestaurant->id = $newRestaurantEntity->getId();  
        return $newRestaurant;  
    }
```

- Creamos la entidad
- Le decimos al EntityManager que haga el cambio
- Vemos que se ha modificado la Entidad con el ID dado por BBDD

## Doctrine ORM

# Cómo utilizamos los repositorios: CREATE

### EJERCICIO:

Vamos a tener que crear la entidad TipoRestaurantes con el nombre del tipo

Vamos a tener que llevarlo hasta la BBDD, de forma incremental con respecto al cambio anterior.

Cambiaremos el Servicio RestaurantesService para poder llamar a la entidad y al repositorio

## Doctrine ORM

### Relaciones entre Entidades (ManyToOne)

Por ejemplo ahora vamos a crear una relación entre tipos de restaurantes y restaurantes.

Paso 1: Creamos un campo en la entidad Restaurant de relación con la entidad anterior.

```
php bin/console doctrine:migrations:migrate
```

## Doctrine ORM

### Relaciones entre Entidades (ManyToOne)

Paso 2: Creamos un campo en la entidad Restaurant de relación con la entidad anterior.

- Nos pedirá que añadamos un nuevo campo
- Le diremos que es de tipo Relational
- Le diremos que está relacionado con la entidad RestaurantType
- Le diremos que es una relación ManyToOne
- Nos dirá si queremos también modificar la entidad RestaurantType para meter la lista de Restaurantes.

```
php bin/console doctrine:migrations:migrate
```

## Doctrine ORM

### Relaciones entre Entidades (ManyToOne)

Paso 3: Volvemos a hacer los cambios en BBDD

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



## Doctrine ORM

### Proximos pasos

Ahora podríamos modificar el servicio RestaurantService para que utilice los repositorios y la entidades. (**Ejercicio**)

Otras cosas que sería bueno explorar es ver una **relación de tipo N->M** donde en BBDD se pone una 3ª tabla con atributos.

Imaginar el supuesto en que eltenedor4V tiene usuarios y esos usuarios pueden realizar reservas sobre los restaurantes. Eso es una relación N-M entre usuarios y restaurantes, que llamaremos reservas. Y a su vez tendrá la fecha como atributo de la reserva y el número de comensales.

**¿CÓMO LO HACEMOS? (Ejercicio)**

**muchas gracias**