

# Symfony

---

*Desarrollo de aplicaciones MVC en el servidor con un  
framework PHP*

## 8. Desarrollo de servicios REST con Symfony

Ignacio Iborra Baeza  
Julio Martínez Lucas

# Índice de contenidos

<b>Symfony.....</b>	<b>1</b>
1.Introducción. Servicios REST.....	3
1.1.El protocolo HTTP.....	3
1.2.Los servicios REST.....	3
1.3.El formato JSON.....	4
2.Construyendo una API REST básica.....	5
2.1.Instalando los bundles necesarios.....	5
2.2.Definiendo los servicios.....	6
2.2.1.Listado de todos los elementos (GET /).....	6
2.2.2.Listado de un elemento concreto (GET /id).....	7
2.2.3.Inserción de película (POST).....	8
2.2.4.Borrado de elementos (DELETE /id).....	9
2.2.5.Modificación de elementos (UPDATE /id).....	10
3.Introducción a Postman.....	11
3.1.Descarga, instalación y primeros pasos.....	11
3.2.Añadir peticiones simples: GET.....	13
3.3.Añadir peticiones POST.....	15
3.4.Añadir peticiones PUT o DELETE.....	16
3.5.Exportar/Importar colecciones.....	16
4.Otras opciones adicionales.....	18
4.1.Configuración de CORS.....	18
4.2.Validación de datos.....	18
4.3.Autenticación basada en tokens.....	20
4.3.1.Introducción a JWT.....	20
4.3.2.Creación de la entidad Usuario para validarse.....	21
4.3.3.Generación de certificados.....	21
4.3.4.Configuración en el archivo .env.....	21
4.3.5.Configuración de config/packages/lexik_authentication.yaml.....	22
4.3.6.Configuración de config/packages/security.yaml.....	22
4.3.7.El controlador de login.....	23
4.3.8.Probando la autenticación.....	24
4.3.9.Probando la autorización.....	24
5.Ejercicios.....	26
5.1.Ejercicio 1.....	26
5.2.Ejercicio 2.....	27
5.3.Ejercicio 3 (opcional).....	27
5.4.Ejercicio 4 (opcional).....	28

# 1. Introducción. Servicios REST

---

En esta última sesión del curso veremos cómo emplear Symfony como proveedor de servicios REST, pero para ello, debemos tener claros ciertos conceptos previos. Para empezar, a estas alturas todos tendremos claro que cualquier aplicación web se basa en una arquitectura cliente-servidor, donde un servidor queda a la espera de conexiones de clientes, y los clientes se conectan a los servidores para solicitar ciertos recursos.

## 1.1. El protocolo HTTP

---

Estas comunicaciones entre cliente y servidor se realizan mediante el protocolo **HTTP** (o **HTTPS**, en el caso de comunicaciones seguras). En ambos casos, cliente y servidor se envían cierta información estándar, en cada mensaje:

- En cuanto a los clientes, envían al servidor los datos del recurso que solicitan, junto con cierta información adicional, como por ejemplo las cabeceras de petición (información relativa al tipo de cliente o navegador, contenido que acepta, etc), y parámetros adicionales llamados normalmente *datos del formulario*.
- Por lo que respecta a los servidores, aceptan estas peticiones, las procesan y envían de vuelta algunos datos relevantes, como un código de estado (indicando si la petición pudo ser atendida satisfactoriamente o no), cabeceras de respuesta (indicando el tipo de contenido enviado, tamaño, idioma, etc), y el recurso solicitado propiamente dicho, si todo ha ido correctamente.

## 1.2. Los servicios REST

---

En esta sesión del tema veremos cómo aplicar lo aprendido hasta ahora para desarrollar un servidor sencillo que proporcione una API REST a los clientes que se conecten. REST son las siglas de *REpresentational State Transfer*, y designa un estilo de arquitectura de aplicaciones distribuidas basada en HTTP. En un sistema REST, identificamos cada recurso a solicitar con una URI (identificador uniforme de recurso), y definimos un conjunto delimitado de comandos o métodos a realizar, que típicamente son:

- GET: para obtener resultados de algún tipo (listados completos o filtrados por alguna condición)
- POST: para realizar inserciones o añadir elementos en un conjunto de datos
- PUT: para realizar modificaciones o actualizaciones del conjunto de datos
- DELETE: para realizar borrados del conjunto de datos
- Existen otros tipos de comandos o métodos, como por ejemplo PATCH (similar a PUT, pero para cambios parciales), HEAD (para consultar sólo el encabezado de la respuesta obtenida), etc. Nos centraremos de momento en los cuatro métodos principales anteriores

Por lo tanto, identificando el recurso a solicitar y el comando a aplicarle, el servidor que ofrece esta API REST proporciona una respuesta a esa petición. Esta respuesta

típicamente viene dada por un mensaje en formato JSON o XML (aunque éste cada vez está más en desuso). Esto permite que las aplicaciones puedan extenderse a distintas plataformas, y acceder a los mismos servicios desde una aplicación Angular, o una aplicación de escritorio .NET, o una aplicación móvil en Android, por poner varios ejemplos.

Veremos cómo podemos identificar los diferentes tipos de comandos de nuestra API, y las URLs de los recursos a solicitar, para luego dar una respuesta en formato JSON ante cada petición.

### 1.3. El formato JSON

---

JSON son las siglas de *JavaScript Object Notation*, una sintaxis propia de Javascript para poder representar objetos como cadenas de texto, y poder así serializar y enviar información de objetos a través de flujos de datos (archivos de texto, comunicaciones cliente-servidor, etc).

Un objeto Javascript se define mediante una serie de propiedades y valores. Por ejemplo, los datos de una persona (como nombre y edad) podríamos almacenarlos así:

```
let persona = {  
  nombre: "Nacho",  
  edad: 39  
};
```

Este mismo objeto, convertido a JSON, formaría una cadena de texto con este contenido:

```
{"nombre":"Nacho","edad":39}
```

Del mismo modo, si tenemos una colección (vector) de objetos como ésta:

```
let personas = [  
  { nombre: "Nacho", edad: 39 },  
  { nombre: "Mario", edad: 4 },  
  { nombre: "Laura", edad: 2 },  
  { nombre: "Nora", edad: 10 }  
]
```

Transformada a JSON sigue la misma sintaxis, pero entre corchetes:

```
[{"nombre":"Nacho","edad":39}, {"nombre":"Mario","edad":4},  
 {"nombre":"Laura","edad":2}, {"nombre":"Nora","edad":10}]
```

## 2. Construyendo una API REST básica

---

Veamos ahora qué pasos dar para construir una API REST que dé soporte a las operaciones básicas sobre una o varias entidades: consultas (GET), inserciones (POST), modificaciones (PUT) y borrados (DELETE).

Como no vamos a necesitar toda la funcionalidad de una aplicación web completa, es mejor que los proyectos de este tipo los creamos a partir del *skeleton* básico, en lugar del *website-skeleton*. En nuestro caso, nos basaremos en el proyecto *symfony.bundles* de ejemplo de la sesión anterior, y trabajaremos con la entidad *Pelicula* que ya tenemos creada.

### 2.1. Instalando los bundles necesarios

---

Como paso previo, deberíamos tener instalado el bundle de **Doctrine**, junto con alguno adicional (*maker*) para poder crear entidades y conectar con la base de datos correspondiente. Como vamos a basarnos en el ejemplo de la sesión anterior, ya tenemos todo esto instalado, pero si creamos un proyecto nuevo, debemos tener presente que necesitaríamos incorporar estos bundles (los recordamos):

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

Además de lo anterior, vamos a hacer uso de un bundle específico para desarrollar APIs REST, llamado **FOSRestBundle**. Está creado por el equipo de desarrollo *Friends Of Symfony*, responsable de varios bundles populares para este framework. Además, este bundle requiere de otro adicional, que se va a encargar de serializar/deserializar los datos que se envíen cliente y servidor, empleando el formato JSON (aunque se puede elegir otro formato, como XML o HTML, pero nos centraremos en JSON). Dicho bundle se llama **JMSSerializerBundle**.

Resumiendo, estos son los comandos que necesitaremos (y en este orden):

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

El primer comando lo necesitaremos porque *JMSSerializerBundle* es un bundle *contribuido*, palabra que significa poco más o menos que no ha sido validado o verificado por el equipo de desarrollo de Symfony, y se acepta como contribución. Sin embargo, debemos especificar en nuestro proyecto que permitimos la instalación de paquetes contribuidos. Posiblemente este comando ya lo habremos escrito antes (es necesario para permitir la reescritura de URLs, por ejemplo), así que no será necesario. Pero lo incluimos en este tutorial por si acaso.

## 2.2. Definiendo los servicios

---

Ahora que ya tenemos instalado lo necesario para empezar a definir los servicios, vayamos a lo importante. Crearemos una nueva clase (para no interferir con lo que ya tenemos), donde definiremos los servicios básicos sobre la entidad *Película*. Llamaremos a esta clase *PelículaRestController*, y la añadimos en la carpeta *src/Controller*:

```
<?php

namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use FOS\RestBundle\Controller\Annotations as Rest;
use FOS\RestBundle\Controller\FOSRestController;
use App\Entity\Película;

/**
 * @Route("/películas/api")
 */
class PelículaRestController extends FOSRestController
{
}

?>
```

La clase tiene una anotación *@Route*, que implica que cualquier ruta que indiquemos dentro va a tener ese prefijo (en este caso, todas las rutas de los métodos internos tendrán el prefijo */películas/api*).

### 2.2.1. Listado de todos los elementos (GET /)

Vamos a añadir un método a nuestra clase anterior para que devuelva, en formato JSON, todas las películas de la base de datos. El código del método es el siguiente:

```
/**
 * @Rest\Get("/", name="lista_peliculas")
 */
public function lista_peliculas()
{
    $serializer = $this->get('jms_serializer');

    $repositorio = $this->getDoctrine()->getRepository(Película::class);
    $peliculas = $repositorio->findAll();

    if (count($peliculas) > 0)
    {
        $respuesta = [
            'ok' => true,
            'peliculas' => $peliculas
        ];
    }
}
```

```

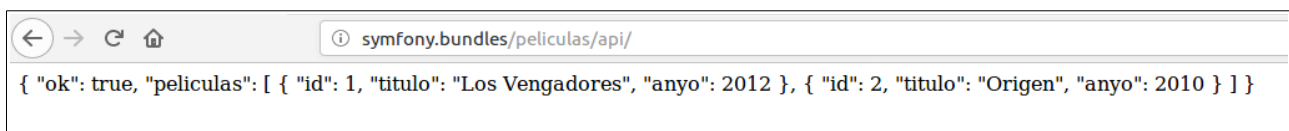
    ];
} else {
    $respuesta = [
        'ok' => false,
        'error' => 'No se han encontrado películas'
    ];
}
return new Response($serializer->serialize($respuesta, "json"));
}

```

Analizamos algunos aspectos importantes que no hemos visto antes:

- El método *lista\_películas* tiene una anotación *@Rest*, que es similar a *@Route* pero permite especificar el comando que se va a atender (GET, en este caso), y la ruta asociada (hemos indicado la raíz /, lo que significa que se atienden peticiones GET a */películas/api/*, que es la ruta base de la clase).
- Dentro del método, hacemos lo siguiente:
  - Obtenemos el serializador
  - Obtenemos el listado de todas las películas (esto ya lo sabemos hacer de sesiones previas)
  - En función de si el listado tiene datos o no, construimos una respuesta a enviar al cliente. Si hay datos, enviamos un atributo *ok* a *true*, y un atributo *películas* con el array de películas obtenido. Si no hay datos, enviamos el atributo *ok* a *false* y el mensaje de error correspondiente (no se han encontrado películas).
  - Finalmente, emitimos una respuesta (*Response*) con el array de atributos que hemos construido, serializado a formato JSON.

Si abrimos un navegador y accedemos a *symfony.bundles/peliculaREST* obtendremos esto:



Lo que vemos es la información de respuesta en formato JSON.

### 2.2.2. Listado de un elemento concreto (GET /id)

Vamos ahora a definir un segundo método que buscará una película por su *id*, que recibirá como parte de la URI, tal y como hemos hecho en sesiones previas con los contactos. El método sería el siguiente:

```

/**
 * @Rest\Get("/{id}", name="busca_pelicula")
 */
public function busca_pelicula($id)
{
    $serializer = $this->get('jms_serializer');
    $repositorio = $this->getDoctrine()->getRepository(Pelicula::class);
}

```

```

$pelicula = $repositorio->find($id);

if ($pelicula)
{
    $respuesta = [
        'ok' => true,
        'pelicula' => $pelicula
    ];
} else {
    $respuesta = [
        'ok' => false,
        'error' => 'Película no encontrada'
    ];
}
return new Response($serializer->serialize($respuesta, "json"));
}

```

La diferencia principal con el método anterior es que empleamos el método *find* para buscar por el *id* de la película, y en el resultado JSON ya no devolvemos un array de películas, sino una sola película, en el atributo *pelicula*, si todo ha ido bien.

Podemos probar el funcionamiento de este servicio también desde el navegador, accediendo a la URL *symfony.bundles/peliculas/api/1*, o con cualquier código de película que tengamos disponible en la base de datos. Si el código no es correcto, veremos el mensaje de error en lugar de los datos de la película.



### 2.2.3. Inserción de película (POST)

Vamos a definir ahora una nueva función para gestionar la inserción de películas. En este caso, el comando a tratar es POST, y los datos llegarán fuera de la URI, en el cuerpo de la petición. El método podría ser algo así:

```

/**
 * @Rest\Post("/", name="nueva_pelicula")
 */
public function nueva_pelicula(Request $request)
{
    $serializer = $this->get('jms_serializer');
    $pelicula = new Pelicula();
    $pelicula->setTitulo($request->get('titulo'));
    $pelicula->setAnyo($request->get('anyo'));
    $entityManager = $this->getDoctrine()->getManager();
    $entityManager->persist($pelicula);
    $entityManager->flush();

    $respuesta = [
        'ok' => true,

```



```

        'pelicula' => $pelicula
    ];

    return new Response($serializer->serialize($respuesta, "json"));
}

```

Necesitaremos incluir una instrucción *use* al principio para poder trabajar con el objeto *Request*, que es el que emplearemos para extraer los datos de la petición (el título y año de la película). Con ellos, creamos el objeto *Pelicula*, lo insertamos en la base de datos y lo devolvemos en la respuesta.

Para poder probar este servicio correctamente, y los de borrado y modificación que veremos ahora, necesitaremos alguna herramienta que simule una petición POST. En breve explicaremos una de ellas, llamada Postman.

## 2.2.4. Borrado de elementos (DELETE /id)

El método de borrado recibirá también como parámetro el *id* del elemento a borrar. El código sería así:

```

/**
 * @Rest\Delete("/{id}", name="borra_pelicula")
 */
public function borra_pelicula($id)
{
    $serializer = $this->get('jms_serializer');
    $entityManager = $this->getDoctrine()->getManager();
    $repositorio = $this->getDoctrine()->getRepository(Pelicula::class);
    $pelicula = $repositorio->find($id);
    if ($pelicula)
    {
        $entityManager->remove($pelicula);
        $entityManager->flush();
        $respuesta = [
            'ok' => true,
            'pelicula' => $pelicula
        ];
    } else {
        $respuesta = [
            'ok' => false,
            'error' => 'Película no encontrada'
        ];
    }
    return new Response($serializer->serialize($respuesta, "json"));
}

```

Básicamente, utilizando *Doctrine*, buscamos el elemento a borrar y lo eliminamos. En el caso de no encontrarlo, devolvemos un mensaje de error en la respuesta.

### 2.2.5. Modificación de elementos (UPDATE /id)

La modificación de elementos es similar a la inserción, pero recibiremos como parámetro en la URI el *id* del elemento a modificar.

```
/**
 * @Rest\Put("/{id}", name="modifica_pelicula")
 */
public function modifica_pelicula($id, Request $request)
{
    $serializer = $this->get('jms_serializer');
    $entityManager = $this->getDoctrine()->getManager();
    $repositorio = $this->getDoctrine()->getRepository(Pelicula::class);
    $pelicula = $repositorio->find($id);

    if ($pelicula)
    {
        $pelicula->setTitulo($request->get('titulo'));
        $pelicula->setAnyo($request->get('anyo'));
        $entityManager->flush();
        $respuesta = [
            'ok' => true,
            'pelicula' => $pelicula
        ];
    } else {
        $respuesta = [
            'ok' => false,
            'error' => 'Pelicula no encontrada'
        ];
    }
    return new Response($serializer->serialize($respuesta, "json"));
}
```

Nuevamente, si encontramos la película, actualizamos sus datos con los que recibimos en la petición (como al insertarla nueva), y después actualizamos los cambios.

En este punto, puedes realizar el [Ejercicio 1](#) de los propuestos al final de la sesión.

## 3. Introducción a Postman

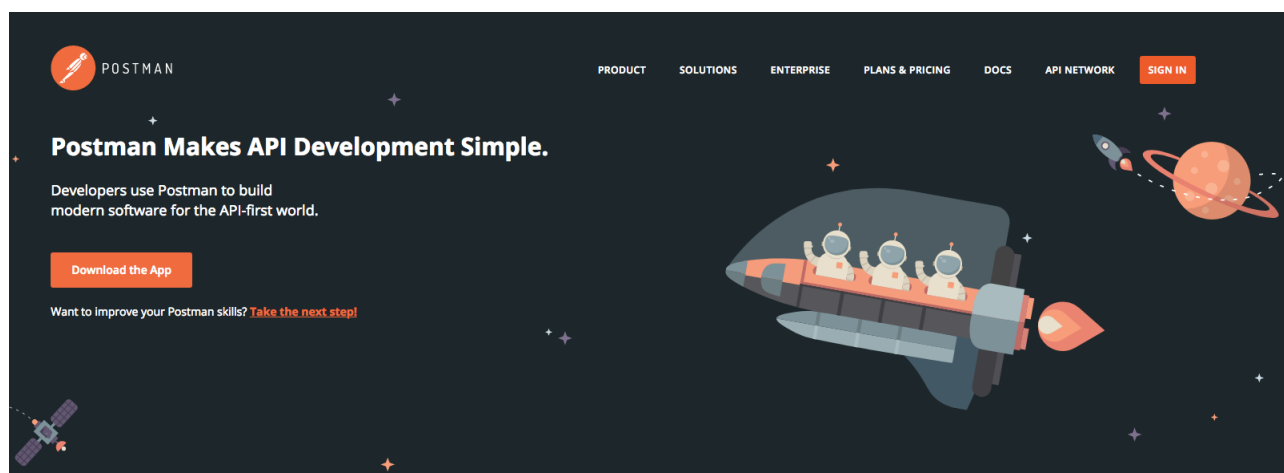
Ya hemos visto que probar unos servicios de listado (GET) es sencillo a través de un navegador. Incluso probar un servicio de inserción (POST) podría hacerse a través de un formulario HTML, pero los servicios de modificación (PUT) o borrado (DELETE) exigen de otras herramientas para poder ser probados. Una de las más útiles en este sentido es Postman.



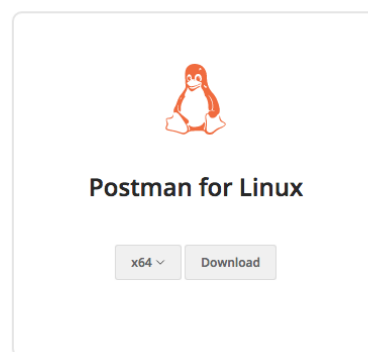
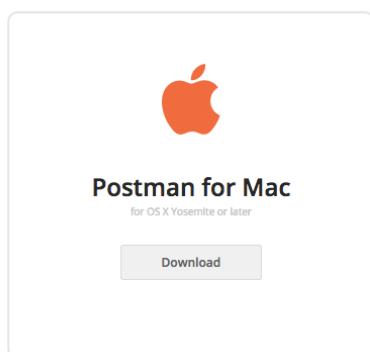
**Postman** es una aplicación gratuita y multiplataforma que permite enviar todo tipo de peticiones a un servidor determinado, y examinar la respuesta que éste produce. De esta forma, podemos comprobar que los servicios ofrecen la información adecuada antes de ser usados por una aplicación cliente real.

### 3.1. Descarga, instalación y primeros pasos

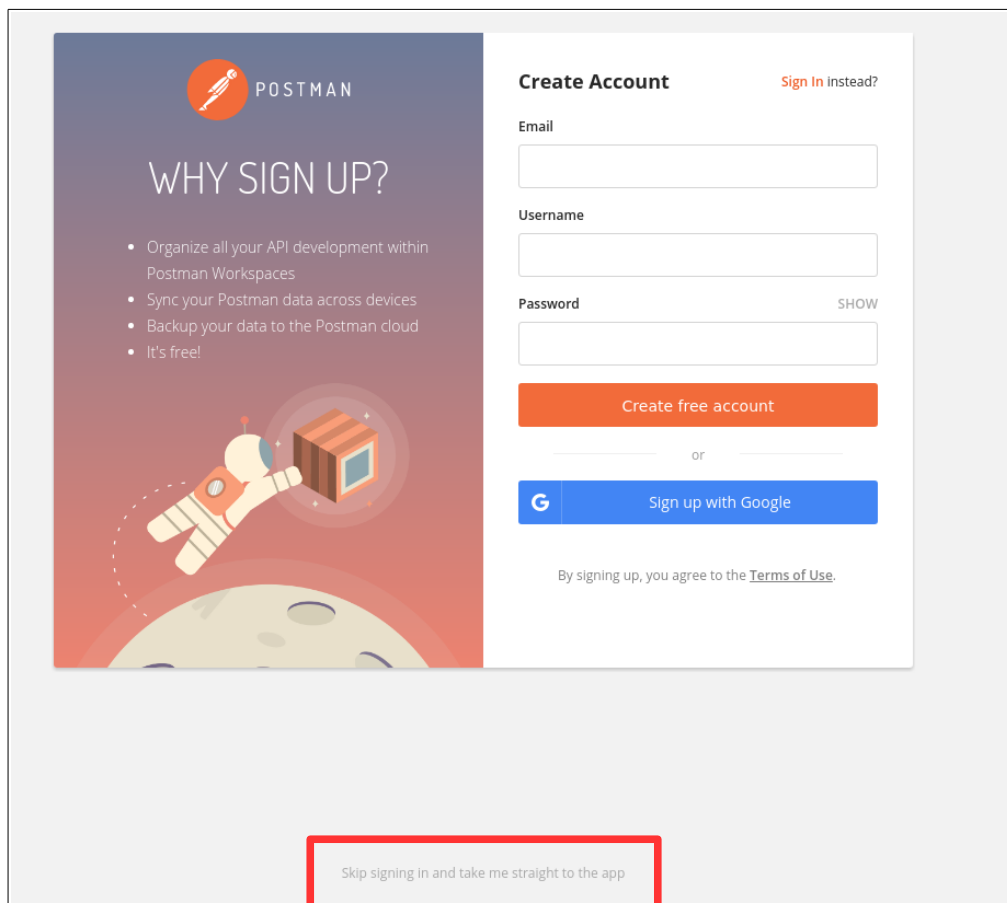
Para descargar e instalar Postman, debemos ir a su [web oficial](#), y hacer clic en el botón de descargar (*Download the app*), y después elegir la versión concreta para nuestro sistema operativo.



Choose your platform:

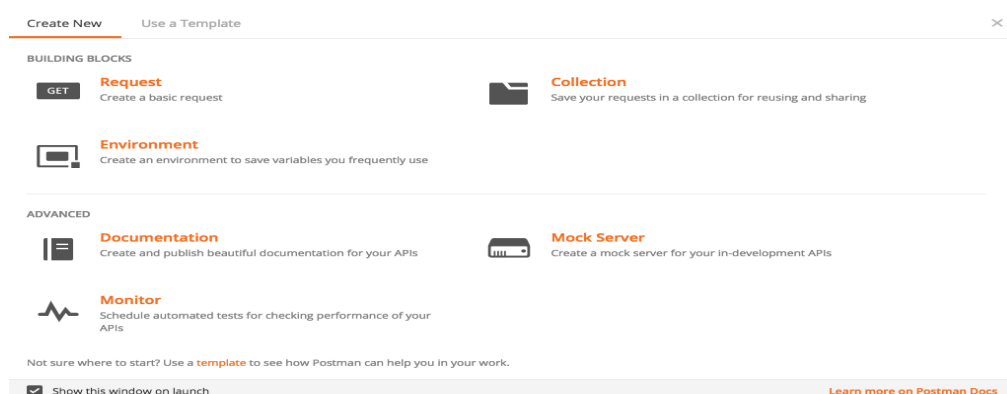


Obtendremos un archivo comprimido portable, que podemos descomprimir y ejecutar directamente. Nada más iniciar, nos preguntará si queremos registrarnos, e incluso asociar Postman a una cuenta Google. Esto tiene las ventajas de poder almacenar en la cuenta las diferentes pruebas que hagamos, para luego poderlas utilizar en otros equipos, pero no es un paso obligatorio, y podemos omitirlo, si queremos, yendo al enlace de la parte inferior de la ventana.



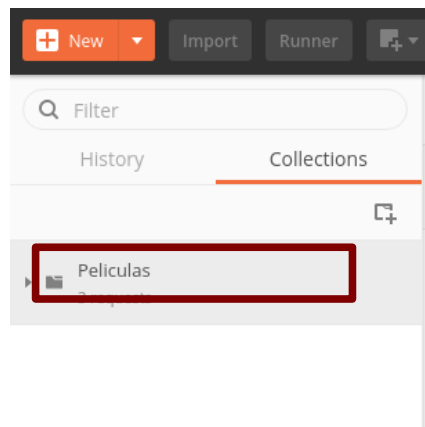
The image shows the Postman 'WHY SIGN UP?' screen. On the left, there's a graphic with the Postman logo and the text 'WHY SIGN UP?' followed by a list of benefits: 'Organize all your API development within Postman Workspaces', 'Sync your Postman data across devices', 'Backup your data to the Postman cloud', and 'It's free!'. On the right, there's a 'Create Account' form with fields for 'Email', 'Username', and 'Password' (with a 'SHOW' toggle). Below the form is a 'Create free account' button. There's also a 'Sign up with Google' button. At the bottom, there's a link to 'Terms of Use'. A red box highlights a button at the bottom center that says 'Skip signing in and take me straight to the app'.

Tras esta pantalla, veremos un diálogo para crear peticiones simples o colecciones de peticiones (conjuntos de pruebas para una aplicación). Lo que haremos habitualmente será esto último.

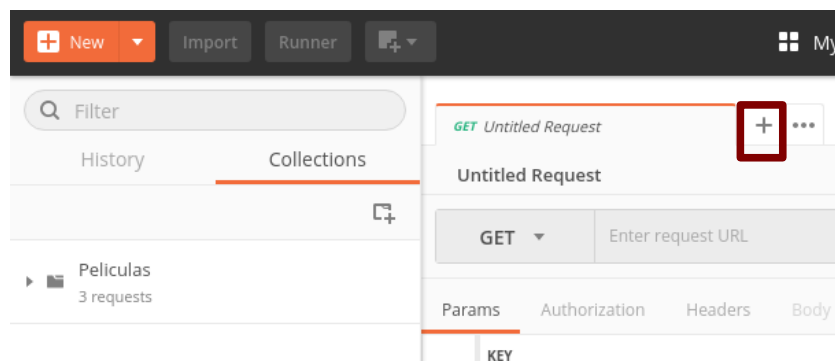


The image shows the 'Create New' dialog in Postman. It has two tabs: 'Create New' and 'Use a Template'. Under 'BUILDING BLOCKS', there are four options: 'Request' (Create a basic request), 'Collection' (Save your requests in a collection for reusing and sharing), 'Environment' (Create an environment to save variables you frequently use), and 'Mock Server' (Create a mock server for your in-development APIs). Under 'ADVANCED', there are two options: 'Documentation' (Create and publish beautiful documentation for your APIs) and 'Monitor' (Schedule automated tests for checking performance of your APIs). At the bottom, there's a checkbox 'Show this window on launch' and a link 'Learn more on Postman Docs'.

Si elegimos crear una colección, le deberemos asociar un nombre (por ejemplo, "Contactos", "Libros", o cualquier nombre asociado a la aplicación que estemos haciendo) y guardarla. Entonces podremos ver la colección en el panel izquierdo de Postman. Para empezar, vamos a crear una colección para gestionar nuestras Películas:

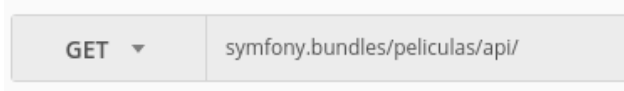


Desde el botón "New" de color naranja en la esquina superior izquierda podemos crear nuevas peticiones (también nuevas colecciones) y asociarlas a una colección. Existe una forma alternativa (quizá más cómoda) de crear esas peticiones, a través del panel de pestañas, añadiendo nuevas:



### 3.2. Añadir peticiones simples: GET

Para añadir una petición, habitualmente elegiremos el tipo de comando bajo las pestañas (GET, POST, PUT, DELETE) y la URL asociada a dicho comando. Por ejemplo, esta sería la petición para obtener el listado de todas las películas:



Entonces, podemos hacer clic en el botón "Save" en la parte derecha, y guardar la petición para poderla reutilizar. Al guardarla, nos pedirá que le asignemos un nombre (por ejemplo, "GET peliculas" en este caso), y la colección en la que se almacenará (nuestra colección de "Películas").

SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).

[Learn more about creating collections](#)

Request name

GET peliculas

Request description (Optional)

Adding a description makes your docs better

Descriptions support Markdown

Select a collection or folder to save to:

Search for a collection or folder

Peliculas

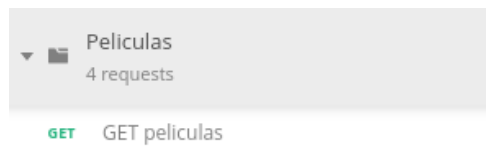
+ Create Folder

POST POST pelicula

Cancel

Save to Peliculas

Después, podremos ver la prueba asociada a la colección, en el panel izquierdo:



Si seleccionamos esta prueba y pulsamos en el botón azul de "Send" (parte superior derecha), podemos ver la respuesta emitida por el servidor en el panel inferior de respuesta:

New Import Runner

My Workspace Invite

Filter

History Collections

Peliculas 5 requests

GET GET peliculas

GET GET pelicula

POST POST pelicula

PUT PUT pelicula

DELETE DELETE pelicula

Tareas 5 requests

GET GET tareas

GET GET tarea

POST POST tarea

PUT PUT tarea

DELETE DELETE tarea

GET GET tareas GET GET tarea POST POST tarea PUT PUT tarea DELETE DELETE tarea GET GET pel X

No Environment

GET GET peliculas

symfony.bundles/peliculas/api/

Send Save

Params Authorization Headers Body Pre-request Script Tests

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (8) Test Results

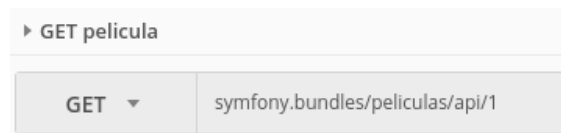
Status: 200 OK Time: 424 ms Size: 686 B Save Download

Pretty Raw Preview HTML

```

1 {
2   "ok": true,
3   "peliculas": [
4     {
5       "id": 1,
6       "titulo": "Los Vengadores",
7       "anyo": 2012
8     },
9     {
10      "id": 2,
11      "titulo": "Shutter Island",
12      "anyo": 2006
13     },
14     {
15      "id": 4,
16      "titulo": "La vida de Brian",
17      "anyo": 1980
18     }
19   ]
20 }
```

Siguiendo estos mismos pasos, podemos también crear una nueva petición para obtener una película a partir de su *id*, por GET, pasándole el *id* en la URI:



Bastaría con reemplazar el *id* de la URI por el que queramos consultar realmente. Si probamos esta petición, obtendremos la respuesta correspondiente:

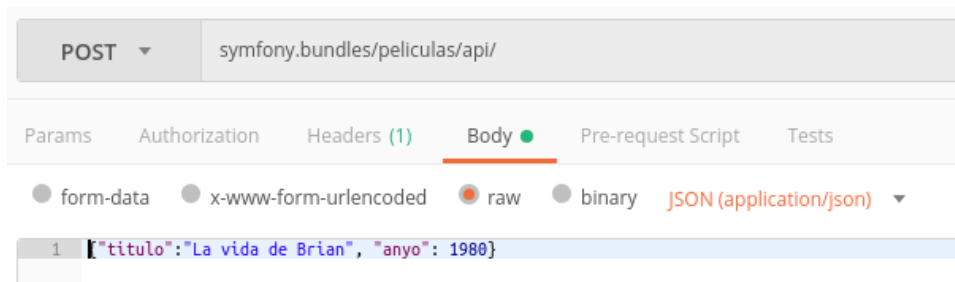


### 3.3. Añadir peticiones POST

Las peticiones POST difieren de las peticiones GET en que se envía cierta información en el cuerpo de la petición. Esta información normalmente son los datos que se quieren añadir en el servidor. ¿Cómo podemos hacer esto con Postman?

En primer lugar, creamos una nueva petición, elegimos el comando POST y definimos la URI (siguiendo con nuestro ejemplo, la URI será *symfony.bundles/peliculas/api/*). Entonces, hacemos clic en la pestaña *Body*, bajo la URL, y establecemos el tipo como *raw* para que nos deje escribirlo sin restricciones. También conviene cambiar la propiedad *Text* para que sea *application/json*, y que así el servidor recoja el tipo de dato adecuado: se añadirá automáticamente una cabecera de petición (*Header*) que especificará que el tipo de contenido que se va a enviar son datos JSON.

Después, en el cuadro de texto bajo estas opciones, especificamos el objeto JSON que queremos enviar para insertar:

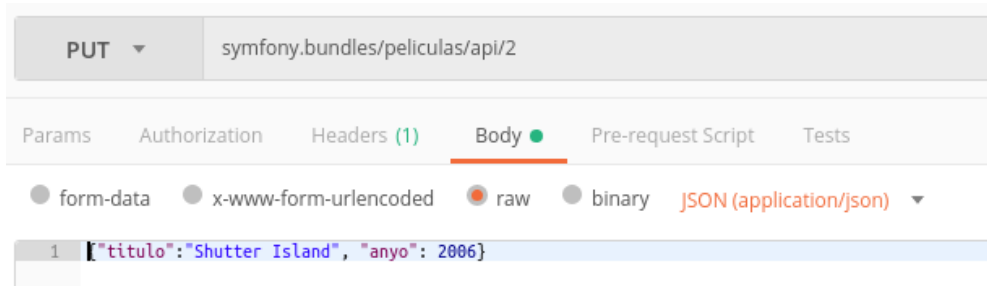


Si enviamos esta petición, obtendremos el resultado de la inserción:

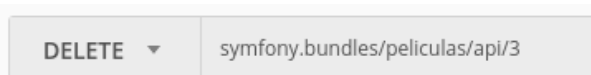


### 3.4. Añadir peticiones PUT o DELETE

En el caso de peticiones PUT, procederemos de forma similar a las peticiones POST vistas antes: debemos elegir el comando (PUT en este caso), la URI, y completar el cuerpo de la petición con los datos que queramos modificar del contacto. En este caso, además, el *id* del elemento a modificar lo enviaremos también en la propia URI:

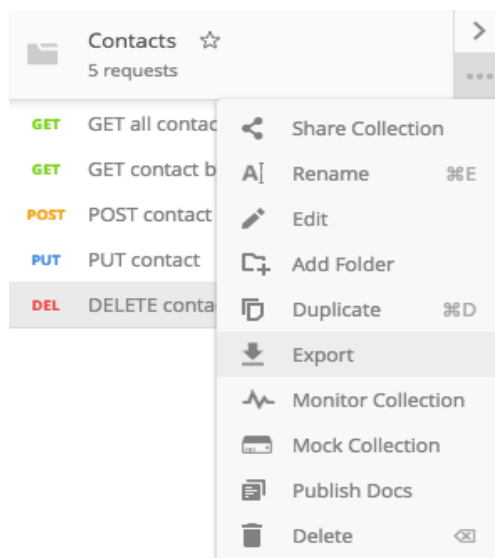


Para peticiones DELETE, la mecánica es similar a la de GET para obtener la ficha de un elemento por su *id*, cambiando el comando GET por DELETE, y sin necesidad de establecer nada en el cuerpo de la petición:



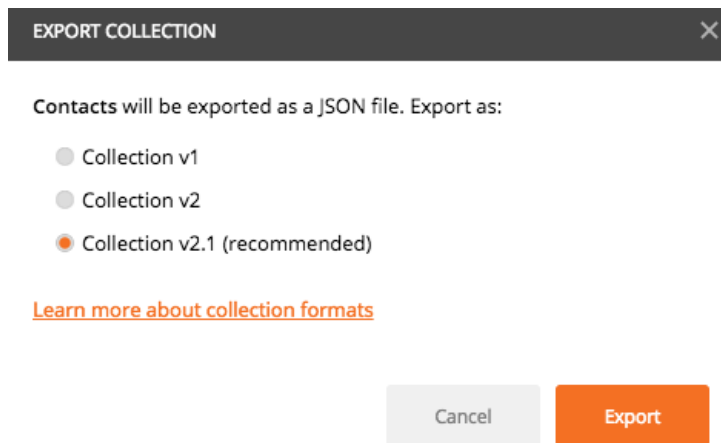
### 3.5. Exportar/Importar colecciones

Podemos exportar e importar nuestras colecciones en Postman, de forma que podemos llevarlas de un equipo a otro. Para **exportar** una colección, hacemos clic en el botón de puntos suspensivos (...) que hay junto a ella en el panel izquierdo, y elegimos *Export*.



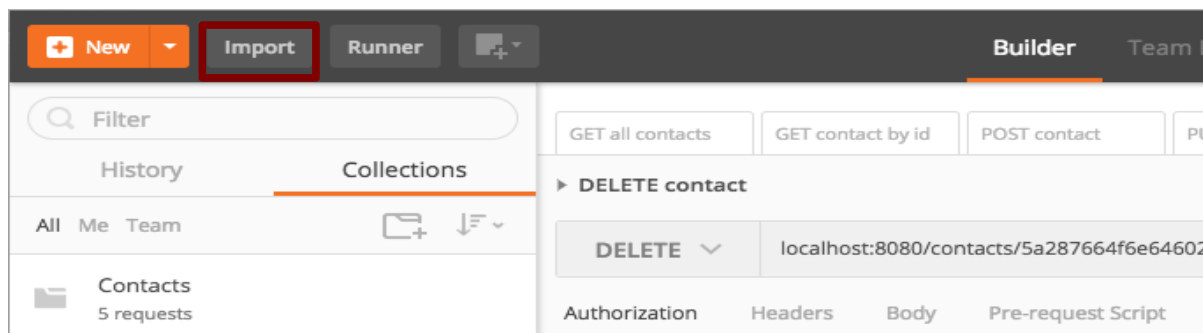
Nos preguntará para qué versión de Postman queremos exportar (normalmente la recomendada es la mejor opción):





Se creará un nuevo archivo Postman en la ubicación que elijamos.

Si queremos **importar** una colección previamente exportada, podemos hacer clic en el botón *Import* de la esquina superior izquierda en la ventana principal:



Entonces, elegimos el archivo Postman con la colección y aparecerá en el panel izquierdo tras la importación.

En este punto, puedes realizar el [Ejercicio 2](#) del final de la sesión.

## 4. Otras opciones adicionales

---

Ahora que ya hemos visto cómo construir una API REST básica utilizando algunos bundles auxiliares, veamos algunas otras opciones que podemos incorporar a este proceso para hacerlo algo más completo.

### 4.1. Configuración de CORS

---

CORS son las siglas de *Cross-Origin Resource Sharing*, una tecnología que permite que dos aplicaciones de dominios diferentes puedan intercambiarse información. Es una tecnología que está muy vinculada a aplicaciones basadas en servicios REST, ya que éstos pueden ser accesibles desde diferentes plataformas cliente. Imaginemos que tenemos una aplicación web en *www.midominio.com*, y queremos acceder desde ella a nuestra API recién creada, por ejemplo, a la ruta *symfony.bundles/peliculas/api/1* para obtener los datos de una película.

Al pertenecer a dos dominios diferentes, esta petición automáticamente queda sin efecto, y el cliente no será capaz de recibir los datos que el servidor envía, por motivos de seguridad (acceso de un dominio a otro diferente).

Para evitar este problema, y permitir que diferentes fuentes o dominios puedan acceder a nuestra API REST, debemos habilitar CORS en nuestra aplicación Symfony. Una de las opciones que tenemos es utilizar un bundle llamado [NelmioCorsBundle](#). Los pasos son sencillos: en primer lugar instalamos el bundle así:

```
composer require cors
```

Después, podemos editar el archivo de configuración *config/packages/nelmio\_cors.yaml* y ajustarlo a nuestro gusto. En la subsección *paths* están las rutas para las que se habilita CORS. Si, por ejemplo, queremos habilitarlo sólo para la subruta */peliculas/apis/*, deberíamos definir únicamente el path *^/peliculas/apis/*.

### 4.2. Validación de datos

---

Podemos incorporar el bundle *validator* de Symfony, que ya hemos empleado para validar los datos que provienen de un formulario, y utilizarlo para validar que los datos que recibe un servicio (POST o PUT) son correctos antes de realizar las correspondientes inserciones o modificaciones. Para ello, en primer lugar incluimos el bundle en el proyecto:

```
composer require symfony/validator
```

A continuación, editamos el archivo de configuración *config/packages/framework.yaml* y añadimos esta línea, para permitir la validación mediante anotaciones:

```
framework:
    ...
    validation: { enable_annotations: true }
```

Después, definimos las reglas de validación en la(s) entidad(es) correspondiente(s). Por ejemplo, si queremos que el título de la película no esté vacío, podemos hacer algo así en nuestra entidad *Pelicula* (indicamos en negrita las dos nuevas líneas a añadir):

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

/**
 * @ORM\Entity(repositoryClass="App\Repository\PeliculaRepository")
 */
class Pelicula
{
    ...
    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank()
     */
    private $titulo;
    ...
}
```

Finalmente, cuando vayamos a utilizar el servicio POST o PUT, para inserciones o modificaciones respectivamente, basta con validar el objeto construido antes de realizar la operación. Si no hay errores, se inserta/modifica, y si los hay, se devuelve un mensaje de error. Así sería en el caso de la inserción:

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

...
/**
 * @Rest\Post("/", name="nueva_pelicula")
 */
public function nueva_pelicula(Request $request, ValidatorInterface $validator)
{
    $serializer = $this->get('jms_serializer');
    $pelicula = new Pelicula();
    $pelicula->setTitulo($request->get('titulo'));
    $pelicula->setAnyo($request->get('anyo'));

    $errores = $validator->validate($pelicula);

    if (count($errores) == 0)
    {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($pelicula);
        $entityManager->flush();

        $respuesta = [
            'ok' => true,
            'pelicula' => $pelicula
        ];
    } else {
        $respuesta = [

```

```

        'ok' => false,
        'error' => 'Los datos no son correctos'
    ];
}

return new Response($serializer->serialize($respuesta, "json"));
}

```

Notar que utilizamos un servicio llamado *ValidatorInterface* para validar los datos. La validación devuelve un array de errores encontrados que, si está vacío, indica que no hay errores, y podemos proceder con la inserción.

En este punto, puedes intentar realizar el [Ejercicio 3](#) del final de la sesión, que es de carácter optativo.

### 4.3. Autenticación basada en tokens

---

Los mecanismos de autenticación tradicional en aplicaciones web están basados en sesiones: el usuario envía sus credenciales a través de algún formulario, el servidor lo valida y almacena en la sesión los datos del usuario logueado, para que, mientras no caduque la sesión o la cierre el usuario, pueda seguir accediendo sin tener que volver a autenticarse.

Sin embargo, este tipo de autenticación tiene la limitación de ser exclusiva para aplicaciones web, es decir, para clientes web que se conecten a servidores web. Si quisiéramos adaptar la aplicación a móvil, o a una versión de escritorio, no podríamos seguir empleando este mecanismo.

Para superar este escollo, podemos utilizar la autenticación basada en tokens. Ésta es una autenticación “sin estado” (*stateless*), lo que significa que no se almacena nada entre cliente y servidor para seguir accediendo autenticados. Lo que se hace es lo siguiente:

1. El cliente envía al servidor sus credenciales (usuario y password)
2. El servidor las valida, y si son correctas, genera una cadena cifrada llamada *token*, que contiene la validación del usuario, además de cierta información adicional que podamos querer añadir (como el login del usuario, por ejemplo). Este token se envía de vuelta al usuario como respuesta a su autenticación.
3. A partir de este punto, cada vez que el cliente quiera autenticarse contra el servidor para solicitar un recurso, basta con que envíe el token que el servidor le proporcionó. El servidor se encargará de verificarlo para comprobar que es correcto, y darle acceso o denegárselo.

Al igual que las sesiones, los tokens también pueden tener una caducidad, que se indica dentro del propio token. Si, pasado ese tiempo, el servidor recibe el token, lo descartará como inválido (caducado), y el cliente volverá a no estar autenticado.

#### 4.3.1. Introducción a JWT

JWT (*JSON Web Token*) es un estándar abierto que permite el envío de tokens de forma segura. Los tokens se firman con una clave, y contienen la información necesaria del

usuario autenticado (normalmente su login es suficiente), para que no se tenga que volver a consultar quién es, mientras el token no caduque.

Para poder trabajar con JWT en Symfony, podemos emplear (entre otros) el bundle *lexik/jwt-authentication-bundle*, que se instala de este modo:

```
composer require jwt-auth
```

Además, necesitaremos añadir el bundle de seguridad de Symfony:

```
composer require symfony/security
```

### 4.3.2. Creación de la entidad *Usuario* para validarse

Emplearemos para validarnos una entidad *Usuario* como la que empleamos en la sesión 6 para la aplicación *symfony.contactos*. Podemos copiarla y pegarla en nuestro proyecto *symfony.bundles*, en la carpeta *src/Entity* junto a la entidad *Pelicula*. También copia y pega el archivo *UsuarioRepository* de la carpeta *src/Repository* en la carpeta correspondiente del proyecto *symfony.bundles*. Cuando tengamos estos dos archivos copiados, ejecutamos los dos comandos para actualizar la base de datos:

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

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

Insertaremos también algún usuario de prueba en la tabla, con el password encriptado en formato *bcrypt* de 12 vueltas, como hicimos en la sesión 6 (puedes copiar alguno de esos usuarios también, si lo prefieres).

### 4.3.3. Generación de certificados

Para poder codificar los tokens, es necesario generar unos certificados. Generaremos uno privado para generar el token cuando el usuario se valide, y uno público para poderlo validar cuando el usuario lo envíe.

Para ello, ejecutamos estos comandos desde la carpeta raíz del proyecto. Cuando nos lo pida, elegiremos como *passphrase* la palabra *symfony* (es sólo una palabra o frase que utilizar para cifrar el contenido, elegimos esa por ejemplo):

```
mkdir config/jwt
```

```
openssl genrsa -out config/jwt/private.pem -aes256 4096
```

```
openssl rsa -pubout -in config/jwt/private.pem -out config/jwt/public.pem
```

### 4.3.4. Configuración en el archivo *.env*

Debemos editar también el archivo *.env* y añadir estas líneas:

```
JWT_PRIVATE_KEY_PATH=config/jwt/private.pem
```

```
JWT_PUBLIC_KEY_PATH=config/jwt/public.pem
```

```
JWT_PASSPHRASE=symfony
```

```
JWT_TOKENTTL=3600
```

El atributo *JWT\_PASSPHRASE* deberá coincidir con el que indicamos al generar los certificados en el paso anterior (*symfony*, en nuestro caso). El atributo *JWT\_TOKENTTL*

es el tiempo de vida o caducidad del token, en segundos. En este caso, le damos un tiempo de validez de una hora.

#### 4.3.5. Configuración de *config/packages/lexik\_authentication.yaml*

Este archivo quedará de este modo, en el que indicamos dónde están generadas la clave privada y pública, la palabra de cifrado y el tiempo de vida:

```
lexik_jwt_authentication:
    private_key_path: '%kernel.project_dir%/%env(JWT_PRIVATE_KEY_PATH)%'
    public_key_path:  '%kernel.project_dir%/%env(JWT_PUBLIC_KEY_PATH)%'
    pass_phrase:      '%env(JWT_PASSPHRASE)%'
    token_ttl:         '%env(JWT_TOKENTTL)%'
```

#### 4.3.6. Configuración de *config/packages/security.yaml*

El archivo principal de seguridad *config/packages/security.yaml* deberá contener estos atributos para la autenticación por token:

```
security:

    encoders:
        App\Entity\Usuario:
            algorithm: bcrypt
            cost: 12

    providers:
        api_user_provider:
            entity:
                class: App\Entity\Usuario
                property: login

    firewalls:
        login:
            pattern: ^/películas/api/login
            stateless: true
            anonymous: true
            provider: api_user_provider
            form_login:
                check_path: /películas/api/login
                success_handler: lexik_jwt_authentication.handler.authentication_success
                failure_handler: lexik_jwt_authentication.handler.authentication_failure
                require_previous_session: false
                username_parameter: username
                password_parameter: password

    api:
```

```

    pattern: ^/peliculas/api
    stateless: true
    anonymous: false
    guard:
        authenticators:
            - lexik_jwt_authentication.jwt_token_authenticator

access_control:
    - { path: ^/peliculas/api/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/peliculas/api, roles: IS_AUTHENTICATED_FULLY }

```

Lo que hemos definido en este archivo es:

- En la sección *encoders* definimos cómo estarán codificados los passwords. En nuestro caso usaremos *bcrypt* de 12 vueltas, como en la sesión 6.
- En la sección *providers* especificamos de dónde obtendremos los usuarios. Emplearemos la entidad *Usuario*, cuyo login viene almacenado en la propiedad *login*
- En la sección *firewalls* especificamos las regiones protegidas de la aplicación:
  - La sección */peliculas/api/login* no está protegida (acceso anónimo abierto). Al acceder a ella se activará el controlador mapeado con la ruta */peliculas/api/login*, y automáticamente tomará un parámetro llamado *username* y otro llamado *password* de la petición, y los validará contra el *provider* indicado. Si todo es correcto, se utilizará el *success handler* indicado (que será el encargado de generar el token). Si algo falla, se empleará el *failure handler* indicado (que enviará una respuesta de no autorización).
  - El resto de la sección */peliculas/api* está protegida mediante el *jwt\_authenticator*, con lo que deberemos enviar un token previamente obtenido para entrar.
- Finalmente, en la sección de *access\_control* indicamos que no es necesario estar autenticado para acceder al login, y sí para el resto de URLs de */peliculas/api*.

#### 4.3.7. El controlador de login

El controlador de login básicamente consiste en mapear la URI */peliculas/api/login* con una función vacía, ya que el propio bundle se encarga de todo (validar credenciales y generar el token, o validar el token que llega del cliente, según el caso). Por lo tanto, podemos añadir este método en nuestra clase *PeliculaRESTController*:

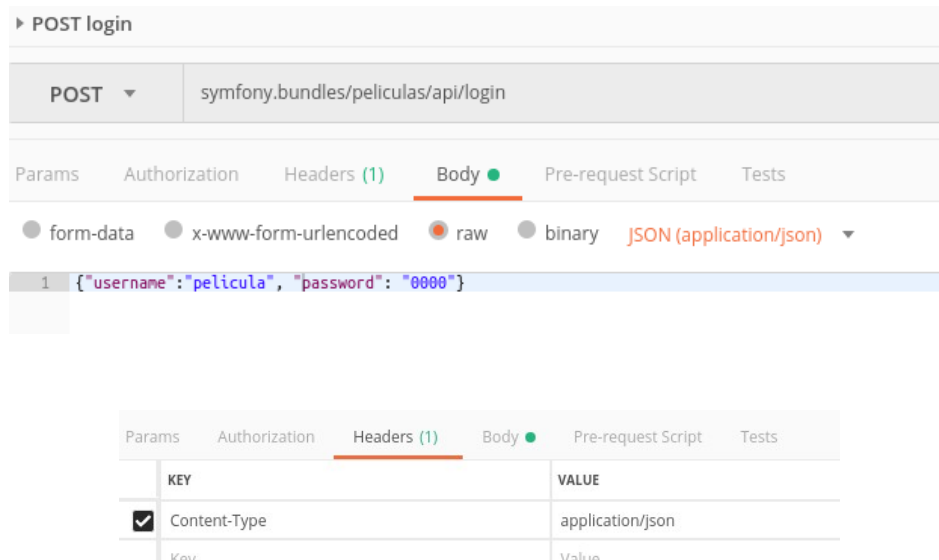
```

/**
 * @Rest\Post("/login", name="login")
 */
public function login() {}

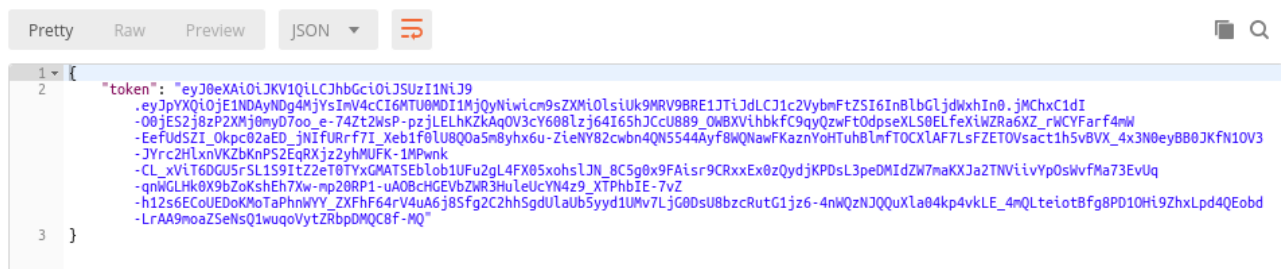
```

### 4.3.8. Probando la autenticación

Para probar que la autenticación funciona, creamos una nueva petición POST en Postman a la URI `/peliculas/api/login`, y le pasamos en el cuerpo de la petición el usuario (*username*) y la contraseña (*password*). En este ejemplo, suponemos que el usuario es *pelicula* y la contraseña (sin encriptar) es *0000*. Debemos añadir también una cabecera (*Header*) con el atributo *Content-Type* establecido a *application/json*.

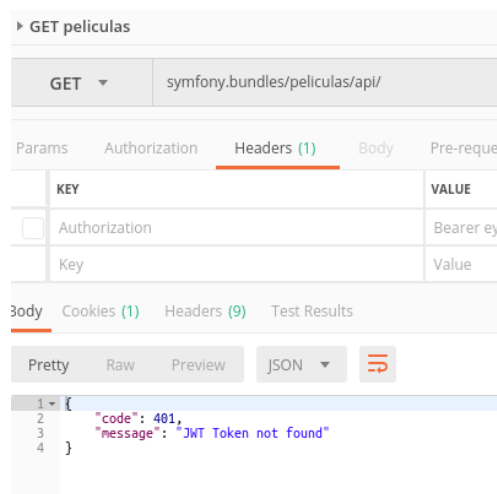


Si todo va correctamente, recibiremos como respuesta un token:



### 4.3.9. Probando la autorización

Ahora, vamos a probar a obtener un listado de películas. Si lanzamos la petición en Postman sin ningún tipo de autorización, recibiremos este mensaje de vuelta:





Debemos añadir una cabecera *Authorization* cuyo valor sea el prefijo “Bearer “ (incluyendo el espacio final) seguido del token que nos ha enviado el servidor al autenticarnos:

The screenshot shows a REST client interface for a GET request to `symfony.bundles/peliculas/api/`. The **Headers** tab is active, showing a table with one header: **Authorization** with a checked checkbox, a key of `Authorization`, and a value of `Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ...`. Below the headers, the **Body** tab is active, displaying a JSON response in pretty-printed format. The response status is **200 OK** with a time of **889 ms** and a size of **70**.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ...	
Key	Value	Description

Status: 200 OK Time: 889 ms Size: 70

```
i 1 {
  2   "ok": true,
  3   "peliculas": [
  4     {
  5       "id": 1,
  6       "titulo": "Los Vengadores",
  7       "anyo": 2012
  8     },
  9     {
 10       "id": 2,
 11       "titulo": "Shutter Island",
 12       "anyo": 2006
 13     },
 14     {
 15       "id": 4,
 16       "titulo": "La vida de Brian",
 17       "anyo": 1980
 18     }
 19   ]
 20 }
```

Con esto, sí obtendremos el listado de películas. Deberemos proceder de la misma forma (enviando el token en la cabecera *Authorization*) para poder emplear el resto de peticiones.

Si quieres probar a hacerlo por ti mismo, puedes realizar el [Ejercicio 4](#) del final de los apuntes, de carácter opcional.

## 5. Ejercicios

---

### 5.1. Ejercicio 1

---

Basándote en el ejemplo de las películas visto en los apartados anteriores, construye ahora tú una API REST sobre la aplicación *symfony.tareas* que iniciamos en la sesión anterior. Ya tendrás instalados los bundles de *Doctrine* de la sesión previa, pero ahora deberás instalar *JMSSerializerBundle* y *FOSRestBundle*, como se explica en esta sesión (apartado 2.1).

Después, crea una clase llamada *TareaRestController* en la carpeta *src/Controller*, y define en ella el código necesario para definir estos servicios. Todos ellos deberán partir de la URI base */tareas/api*:

- Servicio GET a la subruta */*, que devolverá un listado de tareas si todo ha ido bien, o un mensaje de error si no hay tareas que mostrar.
- Servicio GET a la subruta */id*, que devolverá los datos de la tarea con el *id* indicado.
- Servicio POST a la subruta */*, que insertará la tarea que le llega en el cuerpo de la petición, devolviendo la tarea insertada. El formato de la fecha lo puedes elegir tú mismo, y puedes emplear el método *DateTime::createFromFormat* para crear la fecha a partir de ese formato.
- Servicio DELETE a la subruta */id*, que eliminará la tarea con el *id* indicado, devolviendo la tarea eliminada.
- Servicio PUT a la subruta */id*, que modificará los datos de la tarea con el *id* indicado. Recibirá en el cuerpo de la petición todos los datos de la tarea (descripción, fecha y prioridad) y los actualizará sobre la tarea encontrada. Se devolverá la propia tarea modificada.

Todos los servicios devolverán una estructura JSON compuesta por los mismos datos que en el ejemplo de las películas:

- Un atributo *ok*, que será verdadero si la operación ha sido correcta, y falso si no. Por el momento, las operaciones de GET se considerarán incorrectas si no se encuentran resultados, la de POST será siempre correcta, y las de DELETE y PUT serán incorrectas si no se encuentra el elemento a borrar o modificar, respectivamente.
- En el caso de que *ok* sea verdadero (la operación sea correcta), se adjuntará como segundo atributo el elemento afectado (el listado encontrado, o el elemento insertado/borrado/modificado).
- En el caso de que *ok* sea falso (operación incorrecta), se adjuntará un segundo atributo *error* con el mensaje de error correspondiente.

## 5.2. Ejercicio 2

---

Crea una colección en Postman llamada *Tareas*, y define dentro estas cinco peticiones:

- **GET tareas**, que enviará una petición GET a *symfony.tareas/tarea/api/* para obtener todas las tareas
- **GET tarea**, que enviará una petición GET a *symfony.tareas/tarea/api/1*, para obtener los datos de la tarea indicada (en este caso la 1, pero puedes poner el *id* que prefieras para probar).
- **POST tarea**, que enviará una petición POST a *symfony.tareas/tarea/api/*, y en el cuerpo (*body*) de la petición, enviará los datos de una tarea. Por ejemplo (el formato de fecha puede variar dependiendo del que hayas elegido tú):  

```
{"descripcion": "Preparar examen diciembre", "fecha": "10/12/2018", "prioridad": "BAJA"}
```
- **DELETE tarea**, que enviará una petición DELETE a *symfony.tareas/tarea/api/2*, para eliminar la tarea indicada (en este caso la 2, pero puedes poner el *id* que prefieras para probar)
- **PUT tarea**, que enviará una petición PUT a *symfony.tareas/tarea/api/3*, para modificar los datos de la tarea indicada (en este caso la 3, pero puedes poner el *id* que prefieras para probar). Los datos de la tarea (descripción, fecha y prioridad) se enviarán en el cuerpo de la petición. Por ejemplo (el formato de fecha, nuevamente, puede variar dependiendo del que hayas elegido):  

```
{"descripcion": "Acabar sesión 8", "fecha": "28/11/2018", "prioridad": "ALTA"}
```

Cuando tengas todas las peticiones hechas y comprobadas, expórtalas a un archivo llamado *tareas.postman\_collection.json*, que deberás adjuntar como entrega de este ejercicio.

## 5.3. Ejercicio 3 (opcional)

---

Modifica la inserción de tareas del Ejercicio 1 para que se valide que la descripción, la fecha y la prioridad no estén en blanco, y que la prioridad tenga valores entre 1 y 3 (inclusive). Si la tarea no es válida, se devolverá un mensaje de error indicándolo, en lugar de la tarea insertada.

**NOTA:** para indicar que la prioridad esté entre dos valores dados, puedes utilizar las aserciones *GreaterThanOrEqual* y *LessThanOrEqual*, tal y como se explica [aquí](#).

Crea una nueva petición en Postman llamada *POST incorrecto* que intente enviar los datos de una tarea con la prioridad o la descripción vacías (cadena vacía), o con una prioridad no válida (por ejemplo, 5), y comprueba que devuelve un resultado de error. Actualiza el archivo exportado *tareas.postman\_collection.json* del ejercicio anterior con esta nueva petición.

## 5.4. Ejercicio 4 (opcional)

---

Sigue los pasos indicados en el subapartado 4.3 para añadir seguridad basada en tokens en la aplicación de ejemplo *symfony.bundles* (la misma en la que se basa ese apartado). Prueba a autenticarte con el usuario que hayas creado, obtener el token y emplearlo para obtener el listado de películas o cualquier otra ruta.