1. Introducción

En ocasiones, las aplicaciones que desarrolles necesitarán **compartir información con otras aplicaciones**

A veces, una vez que esté finalizada y funcionando, queremos **programar una nueva aplicación** (y no necesariamente una aplicación web) reutilizando la lógica de negocio.

Una primera idea podría ser dar acceso a la base de datos en que se almacena. Pero esta generalmente **no es una buena idea**. Cuantas más aplicaciones utilicen los mismos datos, más posibilidades hay de que se generen **errores** en los mismos.

Además, existen otros **inconvenientes**:

- Si ya tienes una aplicación funcionando, ya has programado la lógica de negocio correspondiente, y ésta no se podrá **aprovechar** en otras aplicaciones si utilizan directamente la información almacenada en la base de datos.

- Si quieres poner la base de datos a disposición de terceros, éstos necesitarán conocer su estructura y, al dar acceso directo a los datos, será **complicado mantener el control sobre las modificaciones que se produzcan en los mismos**.

2. Servicios Web

Para facilitar esta tarea existen los **servicios web**.

Un **servicio web** es un método que permite que dos equipos intercambien información a través de una red informática.

Al utilizar servicios web, el servidor puede ofrecer **un punto de acceso** a la información que quiere compartir. De esta forma **controla y facilita el acceso** a la misma por parte de otras aplicaciones.

Los clientes del servicio, por su parte, no necesitan conocer la estructura interna de almacenamiento. En lugar de tener que programar un mecanismo para localizar la información, tienen un **punto de acceso directo** a la que les interesa.

2.1 Mecanismos y protocolos implicados

Los servicios web se crearon para permitir el intercambio de información sobre la base del protocolo **HTTP** (de ahí el término web).

En lugar de definir su propio protocolo para transportar las peticiones de información, utilizan HTTP para este fin.

La respuesta obtenida no será una página web, **sino la información que se solicitó**. De esta forma pueden funcionar sobre cualquier servidor web y, lo que es aún más importante, utilizando el **puerto 80** reservado para este protocolo.

Por tanto, cualquier ordenador que pueda consultar una página web, podrá también solicitar información de un servicio web.

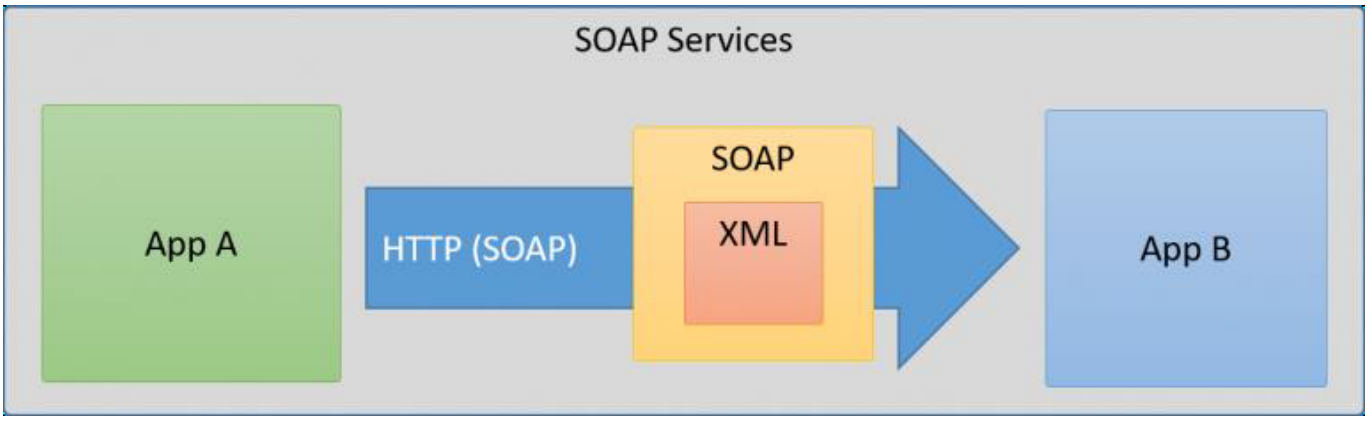
2.2 SOAP vs REST

SOAP y **REST** son dos mecanismos de intercambio de datos a través de Internet.

SOAP (Simple Object Access Protocol) y **REST** (Representational State Transfer) son dos enfoques arquitectónicos utilizados para facilitar la comunicación entre sistemas distribuidos en el desarrollo de aplicaciones web y servicios web.

SOAP

SOAP es un **protocolo basado en XML** diseñado para la transmisión de mensajes entre aplicaciones a través de protocolos de red como HTTP o SMTP.



Utiliza un formato de mensaje estructurado en XML, lo que lo hace altamente **legible** para humanos y máquinas.

SOAP define un conjunto de **reglas y estándares** para la creación y el procesamiento de mensajes, lo que garantiza una comunicación precisa y segura entre sistemas heterogéneos.

Sin embargo, su **estructura compleja** y su **sobrecarga de ancho de banda** pueden hacer que sea menos eficiente en comparación con alternativas más ligeras como REST.

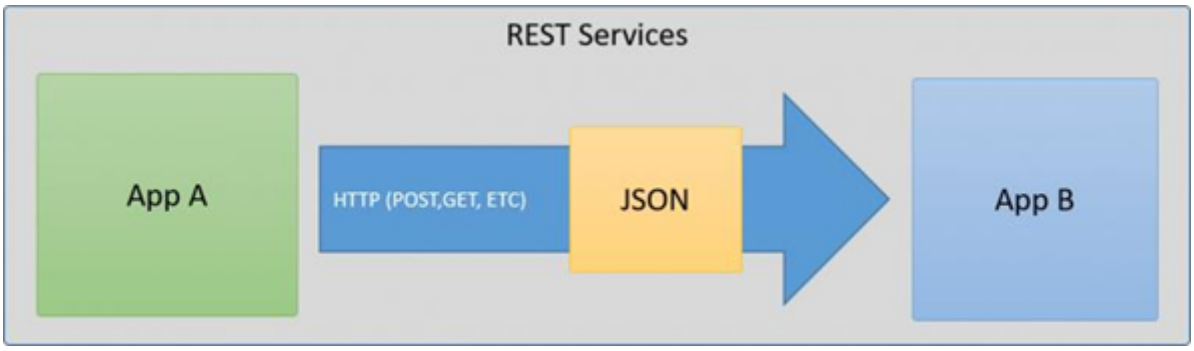
3. Servicios web REST

REST es una tecnología mucho más flexible que transporta datos por medio del protocolo HTTP.

Además permite utilizar los diversos métodos que proporciona HTTP para comunicarse, como lo son GET, POST, PUT, DELETE, PATCH.

Permite transmitir prácticamente **cualquier tipo de datos**, ya que el tipo de datos está definido por el Header Content-Type, lo que nos permite mandar, XML, JSON, binarios (imágenes o documentos), text, etc.

La gran mayoría transmite en JSON por un motivo muy importante: **JSON es interpretado de forma natural por JavaScript**



REST es **más liviano en peso** y **mucho más rápido** en su procesamiento que SOAP.

3.1 Cliente

Para obtener los datos del un servicio web REST se utiliza la **librería cURL**.

Es una biblioteca que permite conectarse y comunicarse con diferentes tipos de servidores y diferentes tipos de protocolos.

Un ejemplo de una petición GET podría ser la siguiente:

```
$url_servicio = "http://museo.laravel/api";
$curl = curl_init($url_servicio);
//establecemos el verbo http que queremos utilizar para la petición
curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "GET");
curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
$respuesta_curl = curl_exec($curl);
curl_close($curl);

$respuesta_decodificada = json_decode($respuesta_curl);
```

> Hoja08_ServiciosWeb_01

3.2 Servidor

La idea es generar una página que devuelva una respuesta en formato JSON.

Se crea un **controlador** específico y las **rutas** correspondientes.

Luego, en el controlador se hacen los **métodos** necesarios para realizar todas las operaciones del servicio web.

4. Crear servicios web en Laravel

Vamos a crear un API REST aprovechando el desarrollo hecho en el tema anterior.

Si fuese un **proyecto nuevo**, habría que crear los modelos, migraciones, relaciones y datos de prueba mediante factorías o seeders.

4.1 Rutas

Desde Laravel 11 hay que ejecutar un comando artisan para generar el fichero de rutas asociados a la API:

```
php artisan install:api
```

Esto nos creará el fichero **routes/api.php**

Las rutas para crear una API en Laravel se definen en el fichero **routes/api.php**

Un enfoque común es el uso del método **Route::apiResource()** para generar automáticamente rutas RESTful para un controlador.

Por ejemplo, si tenemos un controlador llamado ArticuloApiController, podemos usar:

```
Route::apiResource('articulos', ArticuloApiController::class);
```

Así generamos rutas CRUD estándar incluyendo los métodos index, store, show, update y destroy.

4.2 Controlador

Crearemos **controladores** específicos para nuestra API:

```
php artisan make:controller ArticuloApiController --api
```

Nos creará los métodos **index, store, show, update y destroy**

> Hoja08_ServiciosWeb_02

4.3 Resources

Los Resources en Laravel son herramientas para **dar forma a las respuestas JSON de tu API**.

Son especialmente útiles para personalizar la presentación de datos y mantener la consistencia en la salida de la API.

Para crear un **Resource**, utilizamos:

```
php artisan make:resource ArticuloResource
```

Esto generará un nuevo archivo en **App/Http/Resources** que podemos personalizar para presentar los datos de un artículo

Para crear un **Resource Collection**, podemos utilizar el siguiente comando:

```
php artisan make:resource ArticuloCollection
```

Un "**Resource**" en Laravel se utiliza para transformar **un solo modelo** en una respuesta JSON, mientras que una "**Resource Collection**" se emplea para transformar **colecciones** de modelos en respuestas JSON estructuradas

4.3.1 Transformación de datos

En el Resource, el método **toArray($request)** define cómo se deben transformar los datos del modelo Articulo antes de ser enviados como respuesta JSON.

Puedes personalizar qué campos incluir, renombrarlos y agregar información adicional.

```
// App/Http/Resources/ArticuloResource.php

public function toArray($request)
{
```

```
        return [
            'id' => $this->id,
            'titulo' => $this->titulo,
            'contenido' => $this->contenido,
            'fecha_publicacion' => $this->created_at->format('Y-m-d'),
            // Otros campos según sea necesario
        ];
    }
```

En el Resource Collection, el método **toArray($request)** define cómo se deben transformar los datos de una **colección** de modelos Articulo.

Puedes aplicar la misma lógica de transformación que en un Resource, pero ahora se aplica a toda la colección.

4.3.2 Uso en controladores

En el controlador ArticuloApiController, podemos utilizar el Resource al devolver una respuesta JSON.

Por ejemplo, en el método show para obtener un artículo:

```
// App/Http/Controllers/ArticuloApiController.php

public function show(Articulo $articulo)
{
    return new ArticuloResource($articulo);
}
```

Y el método index para obtener una colección de artículos:

```
public function index()
{
    $articulos = Articulo::all();
    return new ArticuloCollection($articulos);
}
```

4.3.3 Paginación

Laravel Resource Collections ofrecen soporte integrado para la **paginación** de resultados. Esto es útil cuando estamos manejando grandes conjuntos de datos y queremos dividir los resultados en páginas para mejorar el rendimiento y la usabilidad.

```
public function index()
{
    $articulos = Articulo::paginate(5);
    return new ArticuloCollection($articulos);
}
```

> Hoja08_ServiciosWeb_03

5. Insertar datos

La inserción de datos en una API REST desarrollada en Laravel implica el uso de las operaciones **POST** para enviar información al servidor.

5.1 Rutas

Ya habíamos definido las rutas utilizando **Route::apiResource** en el fichero **routes/api.php**

Podemos comprobar las rutas con el comando **php artisan route:list**

```
  GET|HEAD        api/articulos ..............  articulos.index
  POST            api/articulos ..............  articulos.store
  GET|HEAD        api/articulos/{articulo} ....  articulos.show
  PUT|PATCH       api/articulos/{articulo} ....  articulos.update
  DELETE          api/articulos/{articulo} ....  articulos.destroy
```

5.2 Controlador

En el controlador habrá que implementar el método **store** que manejará la inserción de datos.

Podemos utilizar el parámetro $request para acceder a los datos enviados o podemos personalizarlo como veremos más adelante.

```php
class ArticuloApiController extends Controller
{
    public function store(Request $request)
    {
        // Validación de los datos recibidos
        $request->validate([
            'titulo' => 'required|string|max:255',
            'contenido' => 'required|string',
            // Otros campos y reglas de validación según tus necesidades
        ]);

        // Crear un nuevo artículo con los datos recibidos
        return new ArticuloResource(Articulo::create($request->all()));
    }
}
```

5.3 Validación

Es crucial validar los datos antes de almacenarlos.

Laravel proporciona un sistema de validación potente.

En el ejemplo anterior, estamos utilizando el método **validate** en la solicitud para garantizar que los campos requeridos y otras reglas sean cumplidas.

Podemos personalizar las solicitudes (Request) para manejar la validación de datos.

Utilizaremos el comando:

```
php artisan make:request CreateArticuloRequest
```

Esto generará un nuevo archivo en el directorio **app/Http/Requests** llamado **CreateArticuloRequest.php**.

Personalizaremos las reglas de validación editando el fichero:

```
class CreateArticuloRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'titulo' => 'required|string|max:255',
            'contenido' => 'required|string',
            // Otras reglas de validación según tus necesidades
        ];
    }
}
```

El método **authorize()** en una clase de solicitud (Request) se utiliza para determinar si el usuario actual tiene el permiso necesario para realizar la acción asociada con la solicitud (en este caso crear un artículo)

Por último, actualizaremos el parámetro del método store del controlador:

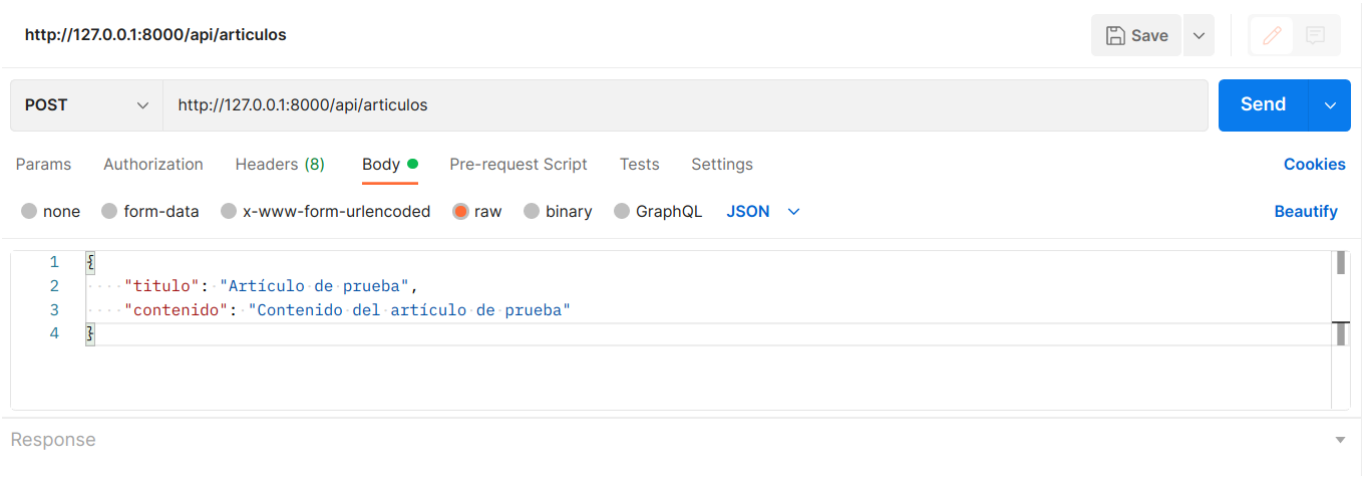
```
class ArticuloApiController extends Controller
{
    public function store(CreateArticuloRequest $request)
    {
        // Los datos ya han sido validados en la solicitud

        // Crear un nuevo artículo con los datos recibidos
        return new ArticuloResource(Articulo::create($request->all()));
    }
}
```

Al utilizar una solicitud personalizada, Laravel manejará automáticamente la validación **antes de que el método store** en el controlador sea ejecutado.

Si la validación falla, Laravel enviará una respuesta JSON que describe los errores de validación.

5.4 Postman

Usaremos Postman para probar nuestra API



- Asegúrate de que la solicitud sea de tipo **POST**.
- En la pestaña **Body**, selecciona el formato **raw**, con formato **JSON** para enviar los datos del nuevo artículo
- **Enviar** la solicitud y analizar la respuesta

> Hoja08_ServiciosWeb_04

6. Actualizar datos

Utilizaremos la ruta **PUT** generada en el fichero de rutas routes/api.php

El método del controlador que usaremos será el **update**. Este método recibe un Request, que podemos personalizar (como para insertar datos).

Utilizaremos el comando:

```
php artisan make:request UpdateArticuloRequest
```

Generará el fichero **UpdateArticuloRequest.php** en el directorio **app/Http/Requests**.

Dentro del controlador, en el método update:

```
class ArticuloApiController extends Controller
{
    public function update(UpdateArticuloRequest $request, Articulo
$articulo)
```

```
    {
        // Los datos ya han sido validados en la solicitud

        //Actualizamos el artículo
        $articulo->update($request->all());
        return new ArticuloResource($articulo);
    }
}
```

6.1 Validación

Para actualizar podemos utilizar los métodos PUT o PATCH.

- **PUT**: se espera que el cliente proporcione la representación completa del recurso, incluidos todos los campos. Si un campo no se proporciona, se considera nulo.
- **PATCH**: permite actualizar solo los campos específicos proporcionados, sin afectar los demás campos. Es útil cuando solo se quieren actualizar algunos campos del recurso.

```
class UpdateArticuloRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        $metodo = $this->method();
        if($metodo == 'PUT')
        {
            return [
                'titulo' => 'required|string|max:255',
                'contenido' => 'required|string',
                // Otras reglas de validación según tus necesidades
            ];
        }
        else
        {
            return [
                'titulo' => 'sometimes|string|max:255',
                'contenido' => 'sometimes|string',
                // Otras reglas de validación según tus necesidades
            ];
        }
    }
}
```

La regla **sometimes** indica que la validación solo se aplicará si el campo está presente en la solicitud

7. Autenticación

Para proteger nuestras rutas vamos a utilizar **Sanctum**. Podemos comprobarlo en el fichero *composer.json*, dentro de la sección *"require"*

En la validación para las API en Laravel, usando Sanctum, se **verifica** la identidad de un usuario utilizando un **sistema de tokens** y no basado en sesiones como en aplicaciones web tradicionales.

Cuando un usuario se autentica correctamente, se le proporciona un token de API único.

Este token es necesario para realizar solicitudes autenticadas a las rutas protegidas por el middleware de autenticación de Sanctum.

7.1 Generación de tokens

En primer lugar vamos a añadir el trait **HasApiTokens** en la clase **User**:

```
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens;
}
```

Luego, nn el fichero de rutas **routes/web.php** vamos a crear una ruta para poder generar un token.

```
Route::get('/profile/crearToken', [ProfileController::class, 'crearToken'])
    ->name('profile.crearToken')->middleware('auth');
```

Dentro del **controlador**, en el método **crearToken** borraremos todos los tokens anteriores y generaremos uno nuevo para el usuario autenticado.

```
    public function crearToken(Request $request)
    {
        $request->user()->tokens()->delete();
        $token = $request->user()->createToken('token-api');
        return ['token' => $token->plainTextToken];
    }
```

7.2 Envío de tokens en las solicitudes

Todas nuestras rutas de la API las protegeremos.

Para ello, en el fichero de rutas **routes/api.php** haremos:

```
Route::middleware("auth:sanctum")->group(function () {
    Route::apiResource('articulos', ArticuloApiController::class);
});
```

7.3 Revocación de tokens

Podemos borrar todos los tokens, el actual o uno en concreto:

```
// Revoke all tokens...
$user->tokens()->delete();

// Revoke the token that was used to authenticate the current request...
$request->user()->currentAccessToken()->delete();

// Revoke a specific token...
$user->tokens()->where('id', $tokenId)->delete();
```

7.4 Habilidades de tokens

Sanctum permite asignar "habilidades" (abilities)

Se puede pasar una serie de habilidades en modo texto como segundo argumento del método **createToken**:

```
$token = Auth::user()->createToken('token-api', ['crear-articulo']);
```

Para comprobar si tiene esa habilidad usamos el método **tokenCan**:

```
if ($user->tokenCan('crear-articulo')) {
    // ...
}
```

Sanctum también incluye dos middleware que se pueden usar para verificar que una solicitud entrante esté autenticada con un token al que se le ha otorgado una habilidad determinada.

Añadiremos los middleware al fichero **bootstrap/app.php**:

```
use Laravel\Sanctum\Http\Middleware\CheckAbilities;
use Laravel\Sanctum\Http\Middleware\CheckForAnyAbility;

->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
```

```
        'abilities' => CheckAbilities::class,
        'ability' => CheckForAnyAbility::class,
    ]);
})
```

El middleware **abilities** se puede asignar a una ruta para verificar que el token de la solicitud entrante tenga **todas** las habilidades enumeradas:

```
Route::get('/orders', function () {
    // Token tien "crear-articulo" y "editar-articulo" abilities...
})->middleware(['auth:sanctum', 'abilities:crear-articulo,editar-
articulo']);
```

El middleware **ability** se puede asignar a una ruta para verificar que el token de la solicitud entrante tenga **al menos una** de las capacidades enumeradas:

```
Route::get('/orders', function () {
    // Token tiene "crear-articulo" o "editar-articulo" ability...
})->middleware(['auth:sanctum', 'ability:crear-articulo,editar-articulo']);
```

> Hoja08_ServiciosWeb_06