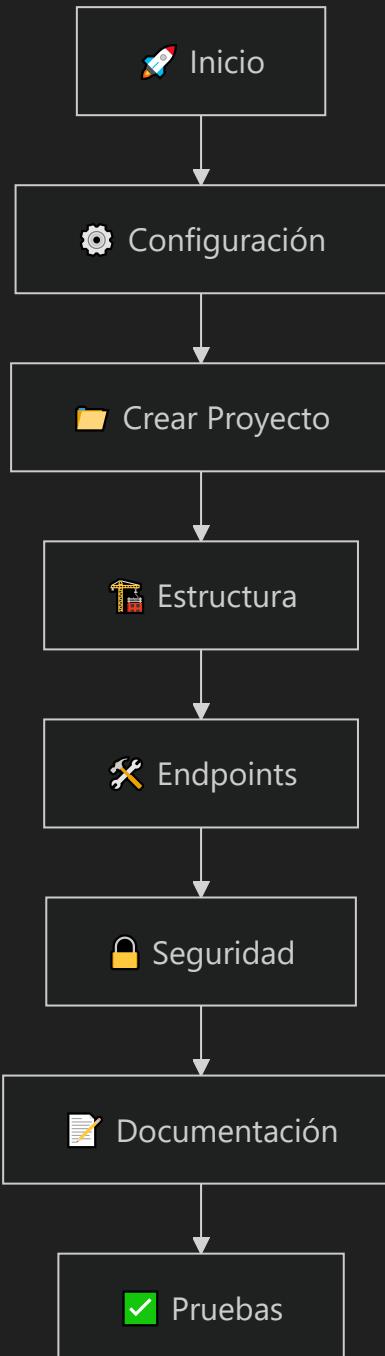


# Desarrollo de API con PHP y Laravel

@Maria Luz Camacho Parra

⚙ Este tutorial está diseñado para enseñarte los fundamentos del desarrollo backend para construir una Api, utilizando PHP y el framework Laravel. A lo largo del curso, explorarás desde la instalación y configuración del entorno de desarrollo hasta la creación de APIs y gestión de bases de datos.

```
graph TD; A["🚀 Inicio"] --> B["⚙️ Configuración"]; B --> C["📁 Crear Proyecto"]; C --> D["🏗️ Estructura"]; D --> E["🔧 Endpoints"]; E --> F["🔒 Seguridad"]; F --> G["📝 Documentación"]; G --> H["✅ Pruebas"];
```



## Introducción a PHP y Laravel: Instalación y Configuración

## ¿Qué es PHP?

PHP (Hypertext Preprocessor) es un lenguaje de programación de código abierto ampliamente utilizado para el desarrollo web del lado del servidor. Desde su creación en 1994, PHP ha evolucionado significativamente y es conocido por su facilidad de uso, flexibilidad y amplia comunidad de desarrolladores. PHP permite crear aplicaciones web dinámicas y es compatible con diversas bases de datos, incluyendo MySQL, PostgreSQL y SQLite. Su sintaxis influenciada por C y Perl facilita la integración con HTML, lo que lo convierte en una elección popular para el desarrollo web.

## ¿Qué es Laravel?

Laravel es un framework de desarrollo web escrito en PHP que fue creado por Taylor Otwell en 2011. Está diseñado para facilitar el desarrollo de aplicaciones web complejas a través de su enfoque en la elegancia, la simplicidad y la eficiencia. Laravel implementa patrones de diseño como MVC (Modelo-Vista-Controlador), lo que permite una separación clara de la lógica del negocio y la presentación. Entre sus características más destacadas se encuentran la gestión de rutas, el ORM Eloquent para la manipulación de bases de datos, la migración de esquemas y un sistema de plantillas denominado Blade.

## Requisitos previos para la instalación

Antes de proceder con la instalación de PHP y Laravel, es fundamental asegurarse de que se cuentan con los siguientes requisitos:

- Servidor Web:** Para el desarrollo local, se puede usar servidores como Apache o Nginx.
- PHP:** Laravel requiere PHP 7.3 o superior. Para esta práctica usaremos la versión 8.2 como mínimo.
- Composer:** Es el gestor de dependencias para PHP que facilita la instalación de Laravel y sus paquetes.
- NOTA:** Para la práctica instalaremos XAMPP <https://www.apachefriends.org/download.html>

## Instalación de XAMPP en Windows

## 1. Descargar XAMPP

- Ve al sitio oficial de Apache Friends:  
<https://www.apachefriends.org/es/index.html>.
- Descarga la versión adecuada para tu sistema operativo (Windows 32/64 bits).

## 2. Ejecutar el instalador

- Una vez descargado el archivo .exe, haz doble clic para ejecutarlo.
- Si aparece una advertencia de "Control de cuentas de usuario (UAC)", da clic en "Sí" para continuar.

## 3. Seleccionar componentes

- Marca los componentes que deseas instalar.
- Se recomienda instalar **Apache, MySQL, PHP y phpMyAdmin** (los demás son opcionales).
- Haz clic en "Siguiente".

## 4. Elegir carpeta de instalación

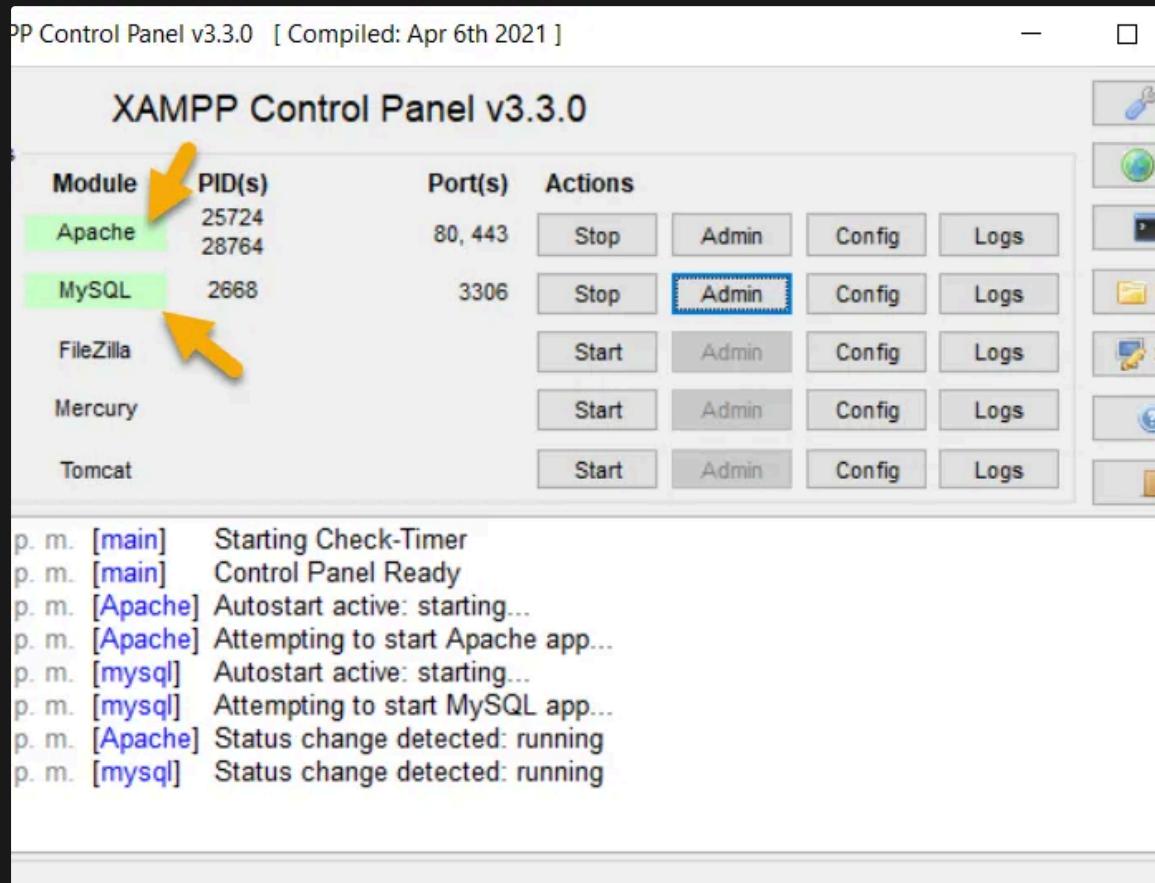
- Se recomienda instalar en C:\xampp para evitar problemas con permisos.
- Haz clic en "Siguiente".

## 5. Iniciar instalación

- Espera a que termine el proceso.
- Cuando finalice, puedes seleccionar la opción para ejecutar XAMPP y hacer clic en "Finalizar".

## 6. Iniciar los servicios

- Abre el "Panel de control de XAMPP".
- Haz clic en "Start" en Apache y MySQL.
- Si todo funciona bien, ambos cambiarán a color verde.



## 7. Probar la instalación

- Abre tu navegador y escribe `http://localhost/`.
- Si ves la página de XAMPP, la instalación fue exitosa.

## Instalación de XAMPP en macOS

### Instalación de XAMPP en macOS

#### 1. Descargar XAMPP

- Ve a <https://www.apachefriends.org/es/index.html>.
- Descarga la versión para macOS.

#### 2. Ejecutar el instalador

- Abre el archivo .dmg descargado y sigue las instrucciones de instalación.
- Arrastra el ícono de XAMPP a la carpeta "Aplicaciones".

### 3. Iniciar XAMPP

- Abre XAMPP desde "Aplicaciones".
- En el panel de control, inicia **Apache** y **MySQL**.

### 4. Probar la instalación

- Abre un navegador y escribe `http://localhost/`.
- Si ves la página de XAMPP, todo está funcionando correctamente.



## Instalación de Composer

Composer es esencial para manejar las dependencias de Laravel. Los pasos para instalar Composer son:

### En Windows

1. **Descargar el instalador:** Visitar el [sitio oficial de Composer](#) y descargar el instalador.
2. **Ejecutar el instalador:** Seguir las instrucciones en pantalla y asegurarse de que PHP esté en el PATH.

### En macOS y Ubuntu/Linux

## 1. Ejecutar un comando en la terminal:

```
curl -sS https://getcomposer.org/installer | php
```

## 2. Mover Composer a una ubicación global:

```
sudo mv composer.phar /usr/local/bin/composer
```

Para verificar que Composer se instaló correctamente, abre una terminal y ejecuta el comando 'composer'. Esto debería mostrar la lista de comandos disponibles. Una vez verificada la instalación de Composer, podemos proceder con la configuración inicial del proyecto Laravel.

```
poser
-----
/ \
/ \ V \ `--\ V --\ V --\ V --\ V --\ V --\
/ \ / \ / \ / \ / \ / \ / \ ( ) / \ / \
/ \ / \ / \ / \ . / \ / \ / \ / \
/ \
poser version 2.8.6 2025-02-25 13:03:50

ge:
command [options] [arguments]
```

## Instalación de Laravel

Una vez que PHP y Composer están instalados, se puede proceder a instalar Laravel:

**Instalar Laravel de forma global:** Ejecutar el siguiente comando en la línea de comandos:

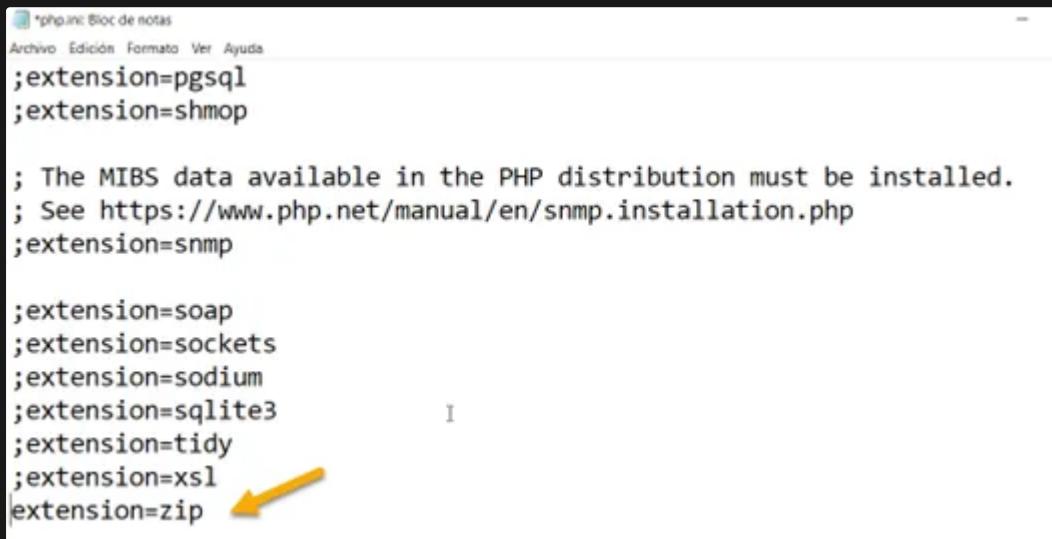
```
composer global require laravel/installer
```

Consultar Documentación <https://laravel.com/docs/12.x#installing-php>

### POSIBLES ERROR AL INSTALAR

<https://www.youtube.com/watch?v=GWHF3W5JTV0>

Failed to download laravel/laravel from dist: The zip extension and unzip/7z commands are both missing, skipping. The php.ini used by your command-line PHP is: C:\xampp\php\php.ini Now trying to download from source



```
php.ini: Bloc de notas
Archivo Edición Formato Ver Ayuda
;extension=pgsql
;extension=shmop

; The MIBS data available in the PHP distribution must be installed.
; See https://www.php.net/manual/en/snmp.installation.php
;extension=snmp

;extension=soap
;extension=sockets
;extension=sodium
;extension=sqlite3
;extension=tidy
;extension=xsl
extension=zip
```

## 1. Configuración Inicial del proyecto

- Crear una carpeta para incluir el proyecto** (En este ejemplo estamos en Transferencia)
- Crear un nuevo proyecto Laravel:** Ejecutar el siguiente comando en la línea de comandos:

```
composer create-project laravel/laravel laravel-api
```

Una vez creado el proyecto, accede al directorio del mismo mediante el comando:

```
cd laravel-api
```

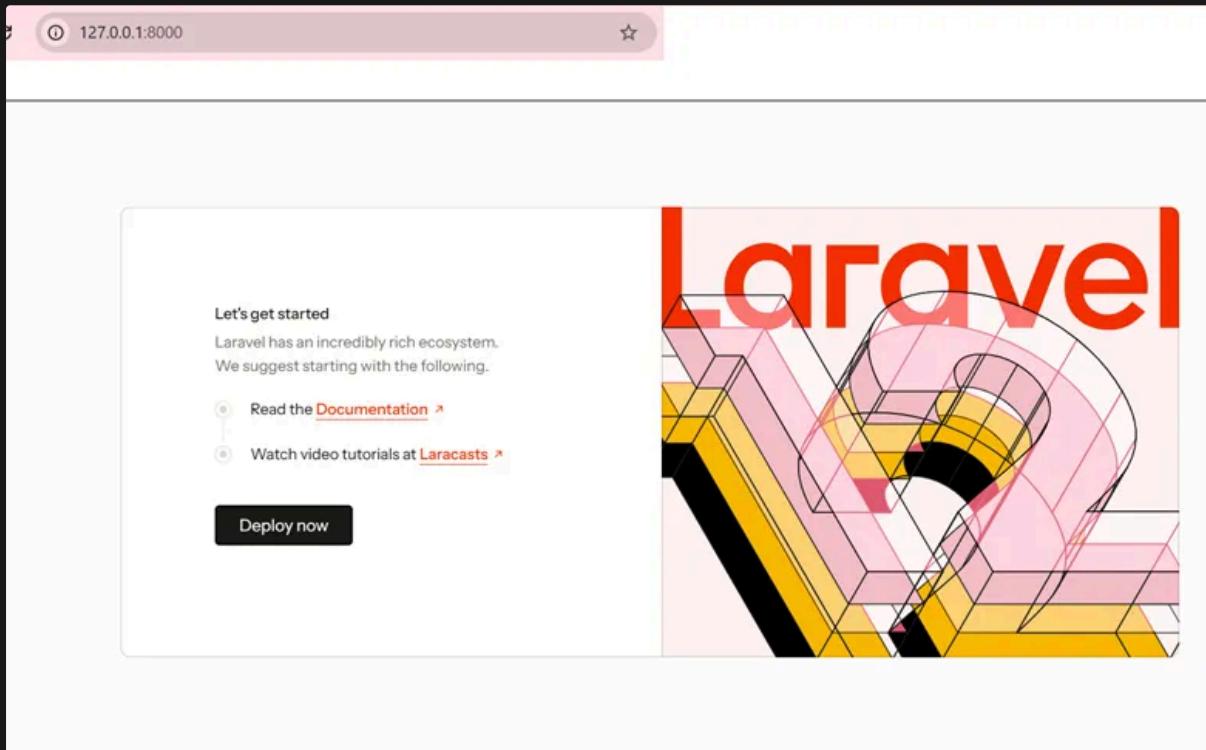
Luego, abra la carpeta en su editor de código del proyecto y en la terminal ejecute el siguiente comando:

```
php artisan serve
```

Para configurar un servidor local para desarrollar aplicaciones Laravel, se puede usar `php artisan serve`.

- Iniciar la aplicación en el navegador, recuerde tener activos los servicios de XAMPP.

Posteriormente, si todo salió bien, iniciar un servidor de desarrollo que permite acceder a la aplicación en `http://localhost:8000`.



- Instalar PHP (versión 8.1 o superior)
- Instalar Composer
- Instalar Laravel vía Composer

## 2. Creación de archivos para la API

### Construcción de API RESTful con Laravel

La creación de una API RESTful con Laravel es un proceso que permite a los desarrolladores construir aplicaciones web robustas y escalables. A continuación, se detallan los conceptos clave y pasos necesarios para implementar una API RESTful utilizando este popular framework de PHP.

## Qué es una API RESTful

Una API RESTful (Representational State Transfer) es un conjunto de convenciones que permite la comunicación entre sistemas mediante el uso de HTTP. REST se basa en recursos, que se representan en formato JSON o XML. Las acciones sobre estos recursos se realizan mediante los métodos HTTP: GET, POST, PUT, PATCH y DELETE.

En Laravel se cuenta un paquete `install: api`, el cual permite crear un archivo de rutas API e instale Laravel Sanctum

- Crear las rutas con el siguiente comando

```
php artisan install:api
```

- Validar como se visualiza en el terminal del editor

```

: \PROYFLUTTER\TRANSFERENCIA> cd .\laravel-api\
: \PROYFLUTTER\TRANSFERENCIA\laravel-api> php artisan install:all
  composer.json has been updated
  Using composer update laravel/sanctum
  Using composer repositories with package information
  Getting dependencies
  File operations: 1 install, 0 updates, 0 removals
  Locking laravel/sanctum (v4.0.8)
  Using lock file
  Calling dependencies from lock file (including require-dev)
  Package operations: 1 install, 0 updates, 0 removals
  Downloading laravel/sanctum (v4.0.8)
  Installing laravel/sanctum (v4.0.8): Extracting archive
  Generating optimized autoload files
  Illuminate\Foundation\ComposerScripts::postAutoloadDump
  php artisan package:discover --ansi

```

**INFO** Discovering packages.

Crea una Base de datos en Sqlite, digitar Yes

```

[INFO] No publishable resources for tag [laravel-assets].
[INFO] No security vulnerability advisories found.
[INFO] Published API routes file.

One new database migration has been published. Would you like to run all pending database migrations? (yes/no) [yes]:
> yes

[INFO] Running migrations.

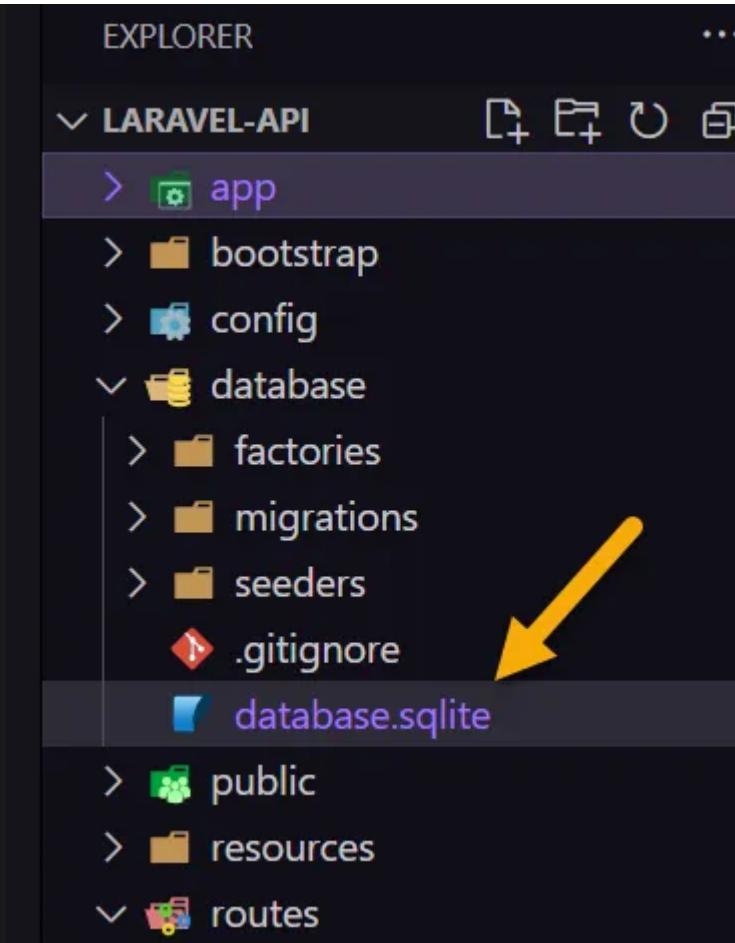
2025_03_17_201301_create_personal_access_tokens_table ..... 16.51ms DONE

[INFO] API scaffolding installed. Please add the [Laravel\Sanctum\HasApiTokens] trait to your User model.

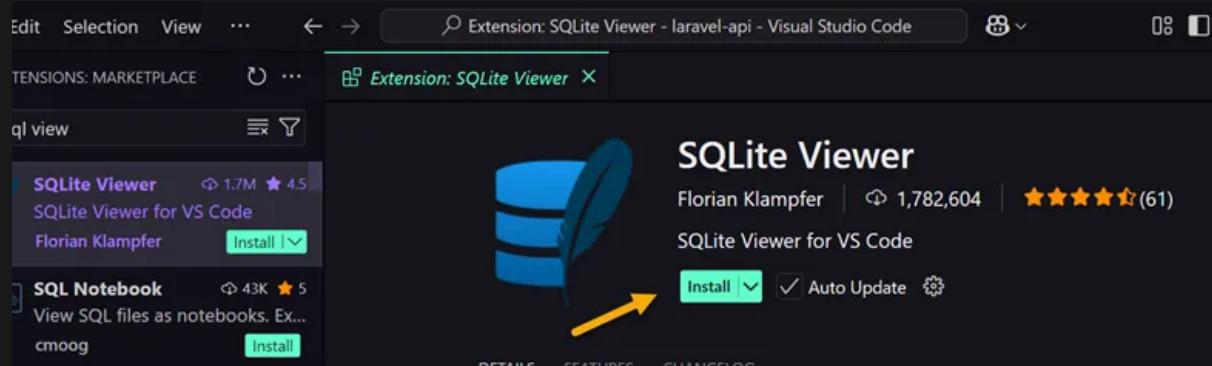
> PS F:\PROYFLUTTER\TRANSFERENCIA\laravel-api> []

```

Crea una base de datos en sqlite



💡 Puede instalar una extensión para visualizar el contenido de la base de datos SQLITE, es SQLite Viewer



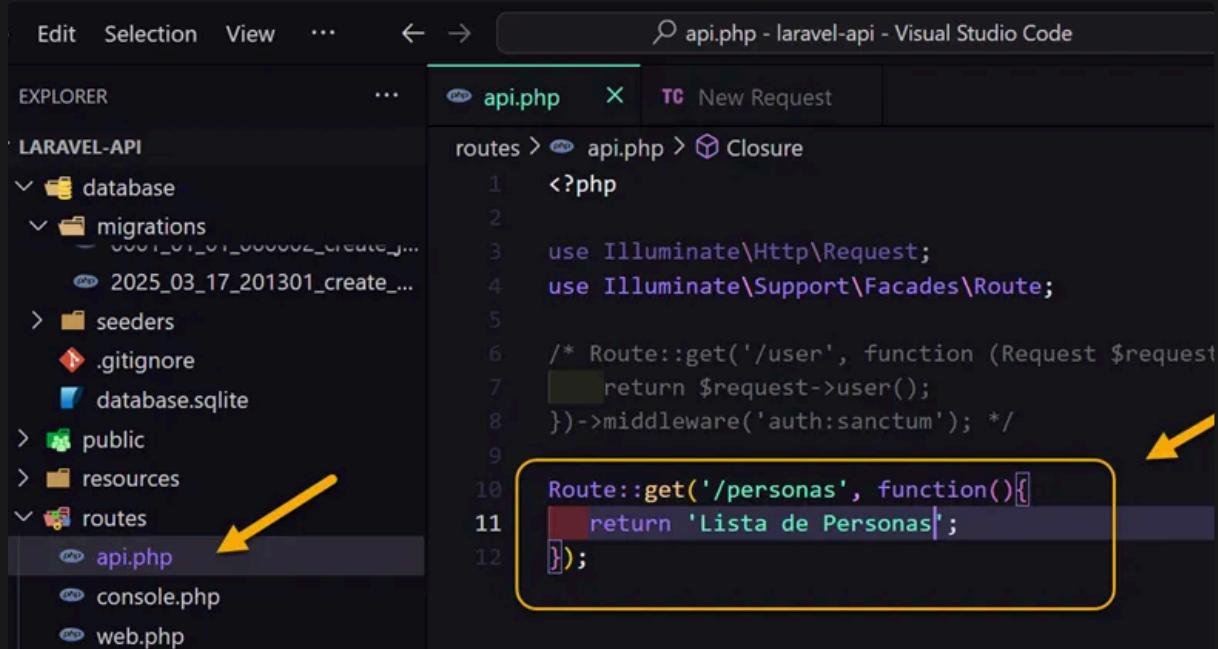
### 3. Crea rutas en el archivo api.php

Organiza tu proyecto siguiendo la estructura de Laravel:

- app/Http/Controllers/- Controladores
- app/Models - Modelos
- database/migrations - Migraciones
- routes/api.php - Rutas API

Vamos a ver las rutas en el archivo api.php

- Crear la primera ruta en el archivo api.php



```
routes > api.php > Closure
1  <?php
2
3  use Illuminate\Http\Request;
4  use Illuminate\Support\Facades\Route;
5
6  /* Route::get('/user', function (Request $request) {
7      return $request->user();
8  })->middleware('auth:sanctum'); */
9
10 Route::get('/personas', function(){
11     return 'Lista de Personas';
12 });


```

En el navegador realizamos las pruebas

- Probar en el navegador

<http://127.0.0.1:8000/api/personas>



- Creamos las rutas para los métodos HTTP

```

Edit Selection View Go Run ... ← → api.php - laravel-api - Visual Studio Code
EXPLORER ...
LARAVEL-API
✓ database
✓ migrations
  2025_03_17_201301_create...
  migrations
  2025_03_17_201301_create...
✓ seeders
  .gitignore
  database.sqlite
> public
> resources
✓ routes
  api.php
  console.php
  web.php
> storage
> tests
> vendor
  .editorconfig
  .env
  .env.example
  .gitattributes
  .gitignore
  artisan
  composer.json
routes > api.php > ...
4  use Illuminate\Support\Facades\Route;
5
6  /* Route::get('/user', function (Request $request) {
7      return $request->user();
8  })->middleware('auth:sanctum'); */
9
10 Route::get('/personas', function(){
11     return 'Lista de Personas';
12 });
13
14 Route::get('/personas/{id}', function(){
15     return 'Lista de una Persona';
16 });
17
18 Route::post('/personas', function(){
19     return 'Creando datos de Persona';
20 });
21
22 Route::put('/personas/{id}', function(){
23     return 'actualizando Persona';
24 });
25
26 Route::delete('/personas/{id}', function(){
27     return 'Eliminando datos de una Personas';
28 });

```

- Instalar Postman para realizar Pruebas, crear una colección
- Probar en Postman para los Métodos GET / POST / PUT / DELETE

POST http://127.0.0.1:8000/api/personas

200 OK • 124 ms • 253 B

## 4. Crear la migración de la tabla persona

Vamos a la terminal cancelamos control + c y vamos a realizar la migración creando una tabla personas

- Ejecutar el comando para crear la migración de la tabla persona

```
php artisan make:migration create_persona_table
```

- Configurar la estructura de la tabla personas en el archivo de migración

Genera un archivo de migración, donde va agregar los campos de la tabla Persona. Se deja \$table->id(); y \$table->timestamps(); que son propios del ORM de Laravel.

```
2025_03_20_182632_create_persona_table.php
```

```
database > migrations > 2025_03_20_182632_create_persona_table.php > class > up
```

```

8  {
9      /**
10     * Run the migrations.
11     */
12    public function up(): void
13    {
14        Schema::create('persona', function (Blueprint $table) {
15            $table->id();
16            $table->string('identificacion');
17            $table->string('nombre');
18            $table->string('apellido');
19            $table->string('email');
20            $table->string('telefono');
21            $table->string('direccion');
22            $table->timestamps();
23        });
24    }
25 }
```

- Ejecutar la migración para crear la tabla en la base de datos

```
php artisan migrate
```

Puede ver la tabla creada en la base de datos con la extensión SQL Vlewer

The screenshot shows the SQLite Viewer interface. On the left, a sidebar lists project files like app, bootstrap, cache, app.php, providers.php, config, database, factories, migrations, and various migration files. In the center, the database browser shows 'database > database.sqlite'. The 'TABLES' section lists several tables, with 'personas' highlighted by a yellow arrow. The table itself has columns: id, identifica..., nombre, and apellido. Another yellow arrow points to the column headers. At the bottom, a terminal window shows the command 'PS F:\PROYFLUTTER\TRANSFERENCIA\laravel-api> php artisan migrate' being run, with a yellow arrow pointing to it. The output shows 'INFO Running migrations.' and '2025\_03\_19\_214435\_create\_personas\_table .....'. The status bar at the bottom indicates 'PROBLEMS DEBUG CONSOLE TERMINAL PORTS'.

## 4. Creación del Modelo persona

Creemos modelos Eloquent para cada tabla de su base de datos. Estos modelos le ayudarán a interactuar con la base de datos de forma orientada a objetos.

- Crear modelo para la tabla Persona

```
php artisan make:model Persona
```

Genera el archivo del modelo en la carpeta de Models

The screenshot shows the Visual Studio Code interface for a Laravel project named 'laravel-api'. The left sidebar displays the project structure:

- /EL-API
- app
  - Http
  - Models
  - Persona.php
  - User.php
  - Providers
  - bootstrap
- config
- database
- factories
- migrations
- seeders
- .gitignore
- database.sqlite
- public
- resources
- routes
- api.php
- console.php
- web.php
- storage
- tests
- vendor
- .editorconfig

The right pane shows the code editor with `Persona.php` open, which contains the following code:<?php  
namespace App\Models;  
use Illuminate\Database\Eloquent\Model;  
class Persona extends Model  
{  
 //  
}

The terminal tab at the bottom shows the command `php artisan make:model Persona` being run, with the output:

- INFO Running migrations.
- INFO Model [F:\PROYFLUTTER\TRANSFERENCIA\laravel-api\app\Models\Persona.php] c
- PS F:\PROYFLUTTER\TRANSFERENCIA\laravel-api>

A yellow arrow points from the terminal output back to the command in the terminal tab.

En este archivo colocamos el nombre de la tabla persona y los campos que se van a gestionar

```
api.php | 2025_03_19_214435_create_personas_table.php | Persona.php
Op > Models > Persona.php > PHP Intelephense > Persona > $fillable
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Persona extends Model
8 {
9     // 
10    protected $table = 'persona';
11
12    protected $fillable = [
13        'identificacion',
14        'nombre',
15        'apellido',
16        'email',
17        'telefono',
18        'direccion'
19    ];
20 }
```

Nombre de la tabla

Nombre de los campos de la tabla

## 5. Creación del Controlador de recurso

Creemos controladores de recursos para cada entidad:

- Generar controlador de la API

```
php artisan make:controller personaController
```

The screenshot shows the Visual Studio Code interface. On the left, the sidebar displays the project structure with 'migrations' highlighted. In the center, the code editor shows the file 'personaController.php' which is empty. The terminal at the bottom shows the command 'php artisan make:controller personaController' being run, with a success message indicating the controller was created successfully.

Implementar el método que se va a llamar desde la ruta

The screenshot shows the Visual Studio Code interface again. The sidebar shows 'migrations' selected. The code editor shows the 'personaController.php' file with the 'index()' method implemented. A yellow callout bubble points to the 'index()' method with the text 'METODO QUE SE PUEDE LLAMAR'.

Ese método lo llamamos desde el archivo de la ruta

## 6. Definir Rutas

Primero importamos el controlador y llamamos el método index que se encuentra en el controlador

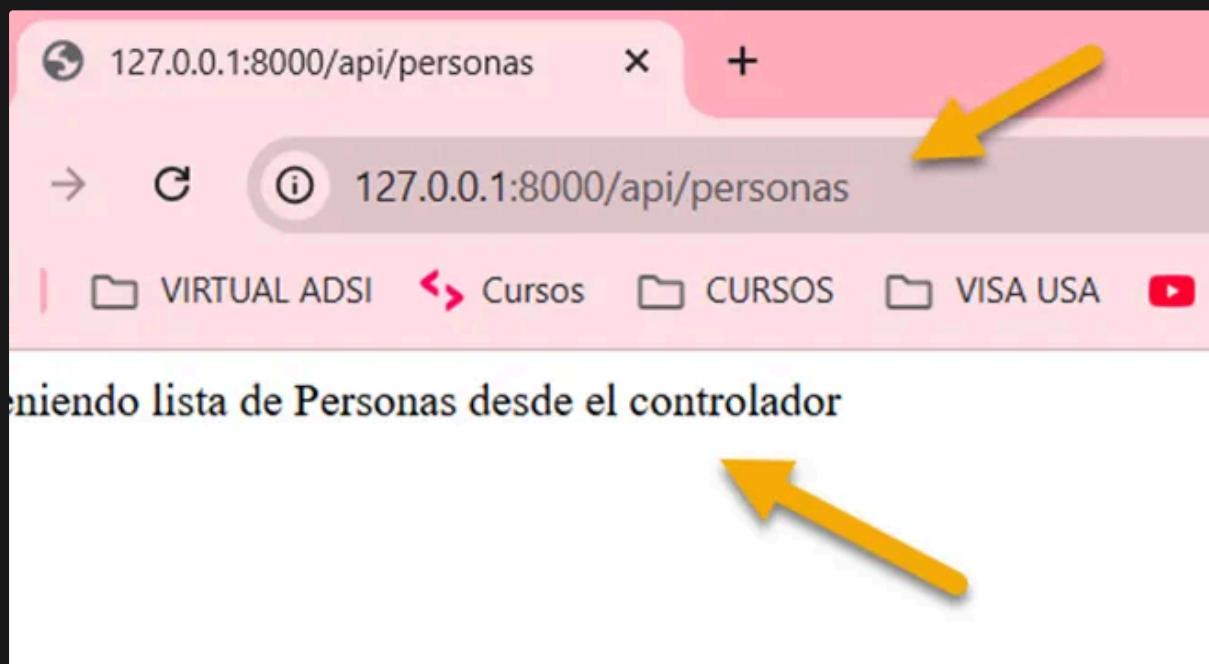
- Crea tus rutas API en el `routes/api.php` archivo:

```

Edit Selection View Go Run ... ← → api.php - laravel-api - Visual Studio Code
EXPLORER ... routes > api.php > ...
routes > api.php > ...
1  <?php
2
3  use Illuminate\Http\Request;
4  use Illuminate\Support\Facades\Route;
5
6  /* Route::get('/user', function (Request $request) {
7      return $request->user();
8  })->middleware('auth:sanctum'); */
9
10 use App\Http\Controllers\personaController;
11
12 Route::get('/personas', [personaController::class, 'index']);
13
14 Route::get('/personas/{id}', function(){
15     return 'Lista de una Persona';
16 });
17
18 Route::post('/personas', function(){
19     return 'Creando datos de Persona';
20 });

```

Probar desde el Navegador



## 7. Creación de los Métodos en el Controlador

### 7.1 Creación del Método all() - consultar todos los registros

Modificamos el archivo personaController.php para traer todos los registros de la tabla persona

Método Consultar Todas los registros de la Tabla Persona

The screenshot shows the file structure of a Laravel application. The sidebar lists files like api.php, Persona.php, personaController.php (selected), Controller.php, and various models and config files. The main area displays the contents of personaController.php:

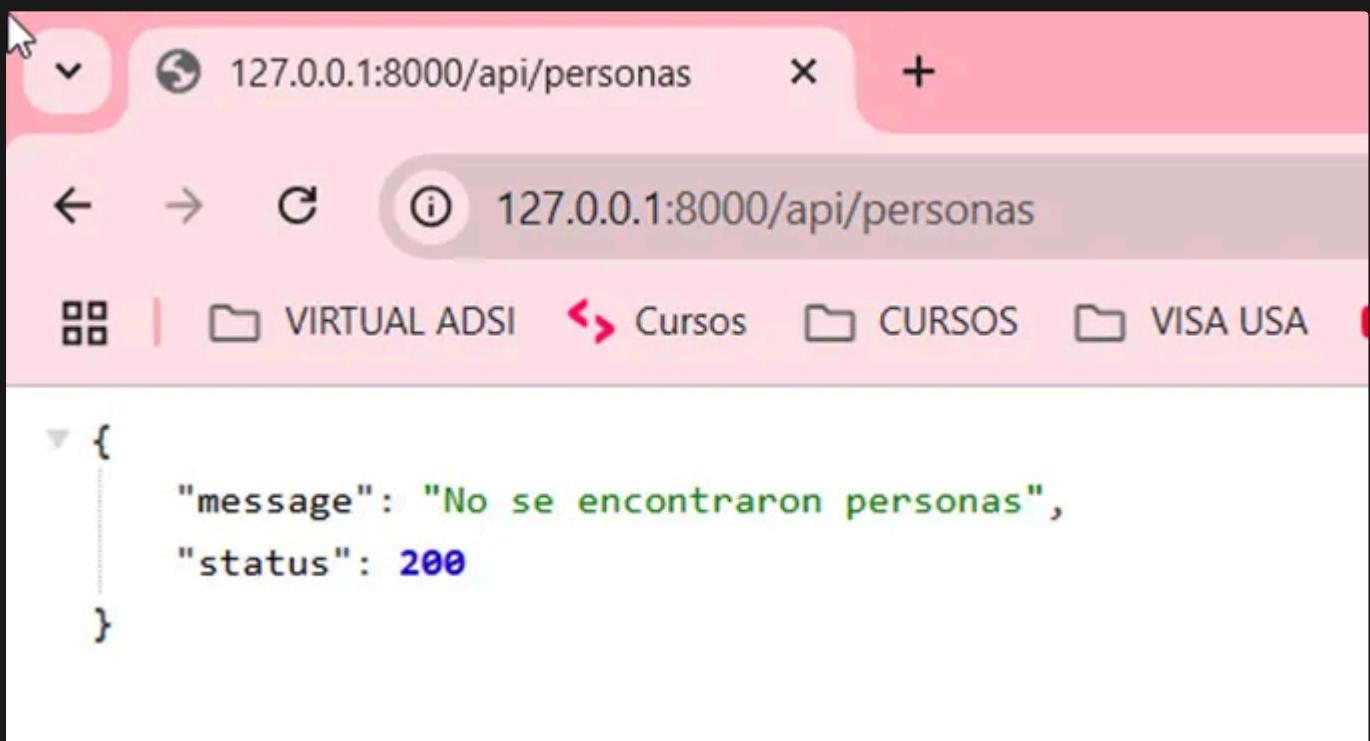
```
namespace App\Http\Controllers;

use App\Models\Persona;
use Illuminate\Http\Request;

class personaController extends Controller
{
    /**
     * @return \Illuminate\Http\JsonResponse
     */
    public function index()
    {
        $personas = Persona::all();
        if ($personas->isEmpty()) {
            $data = [
                'message' => 'No se encontraron personas',
                'status' => 200
            ];
            return response()->json($data, 404);
        }
        return response()->json($personas, 200);
    }
}
```

A yellow callout points to the line '\$personas = Persona::all();' with the text 'Se importa el modelo'. Another callout points to the line 'if (\$personas->isEmpty())' with the text 'Método all() trae todas las personas'.

Probar en el navegador

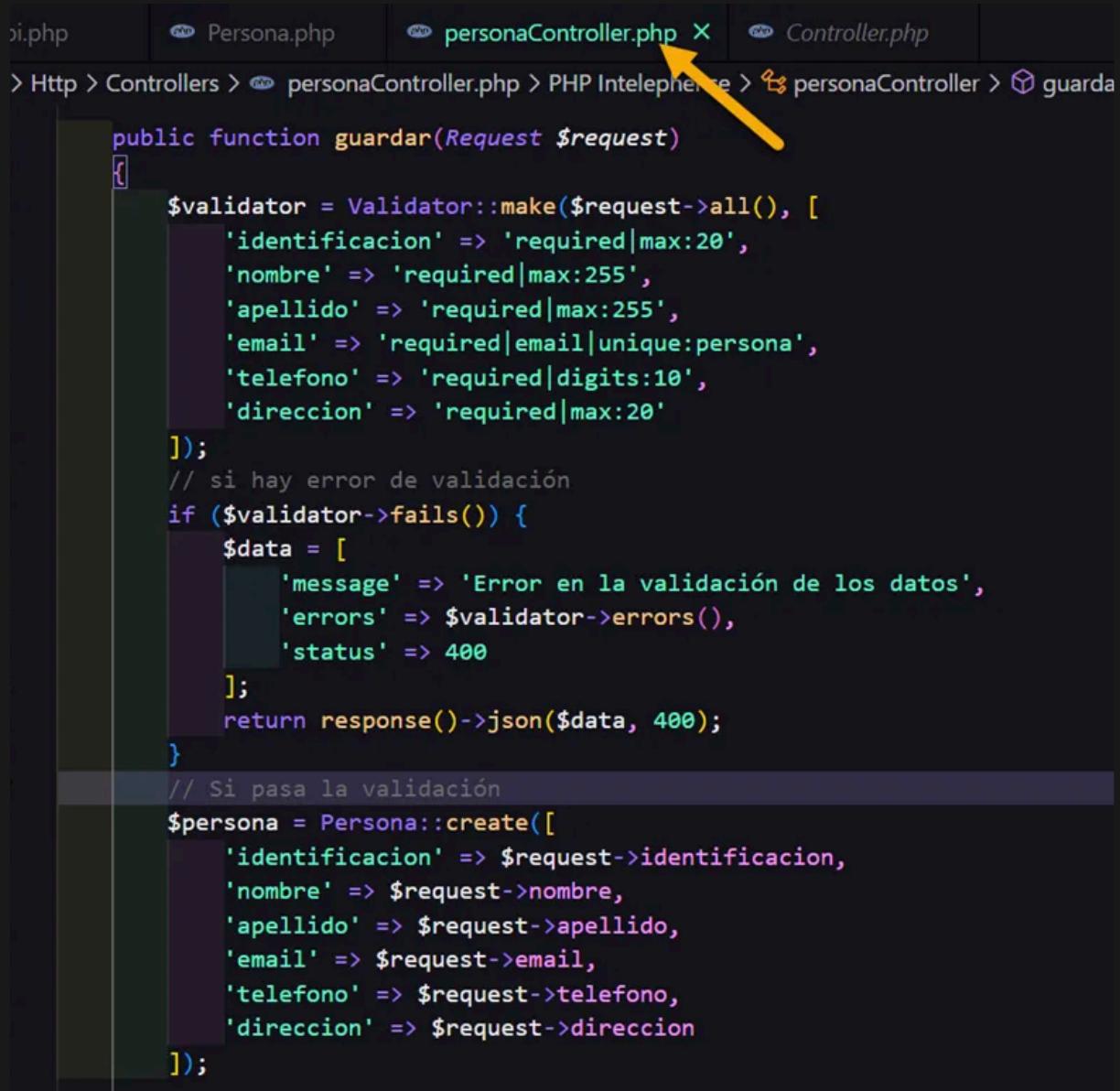


## 7.2 Creación del Método create() - guardar un registro

Esta función implementa un endpoint para una API RESTful que:

1. Recibe datos de una persona
  2. Valida que esos datos cumplan con ciertas reglas
  3. Si la validación falla, devuelve un error 400 con los detalles

4. Si la validación es exitosa, guarda los datos en la base de datos
  5. Si la creación en la base de datos falla, devuelve un error 500
  6. Si todo sale bien, devuelve el objeto creado con un código 201
- Creamos el código en el archivo personaController()



```
public function guardar(Request $request)
{
    $validator = Validator::make($request->all(), [
        'identificacion' => 'required|max:20',
        'nombre' => 'required|max:255',
        'apellido' => 'required|max:255',
        'email' => 'required|email|unique:persona',
        'telefono' => 'required|digits:10',
        'direccion' => 'required|max:20'
    ]);
    // si hay error de validación
    if ($validator->fails()) {
        $data = [
            'message' => 'Error en la validación de los datos',
            'errors' => $validator->errors(),
            'status' => 400
        ];
        return response()->json($data, 400);
    }
    // Si pasa la validación
    $persona = Persona::create([
        'identificacion' => $request->identificacion,
        'nombre' => $request->nombre,
        'apellido' => $request->apellido,
        'email' => $request->email,
        'telefono' => $request->telefono,
        'direccion' => $request->direccion
    ]);
}
```

## Explicación detallada del código

```
public function guardar(Request $request)
```

Esta línea define una función pública llamada `guardar` que recibe un parámetro `$request` de tipo `Request`. En Laravel, este objeto `Request` contiene todos los datos enviados en la petición HTTP (como datos de formularios, parámetros de URL, etc.).

## Validación de datos

```
$validator = Validator::make($request->all(), [ 'identificacion' =>
'required|max:20', 'nombre' => 'required|max:255', 'apellido' =>
'required|max:255', 'email' => 'required|email', 'telefono' =>
'required|max:13', 'direccion' => 'required|max:255' ]);
```

Aquí se crea un validador utilizando la clase `Validator` de Laravel. El método `make()` recibe dos parámetros:

1. `$request->all()` : Todos los datos enviados en la petición
2. Un array con las reglas de validación para cada campo

Las reglas aplicadas son:

- `identificacion` : es obligatorio (`required`) y tiene un máximo de 20 caracteres (`max:20`)
- `nombre` y `apellido` : son obligatorios y tienen un máximo de 255 caracteres
- `email` : es obligatorio y debe tener formato de correo electrónico válido
- `telefono` : es obligatorio y tiene un máximo de 13 caracteres
- `direccion` : es obligatoria y tiene un máximo de 255 caracteres

Manejo de errores de validación

```
if ($validator->fails()) { $data = [ 'message' => 'Error en la validación
de los datos', 'errors' => $validator->errors(), 'status' => 400 ]; return
response()->json($data, 400); }
```

Esta sección verifica si la validación ha fallado usando el método `fails()`. Si hay errores:

1. Se crea un array `$data` con un mensaje general, los errores específicos y un código de estado 400
2. Se devuelve una respuesta JSON con esa información y el código HTTP 400 (Bad Request)

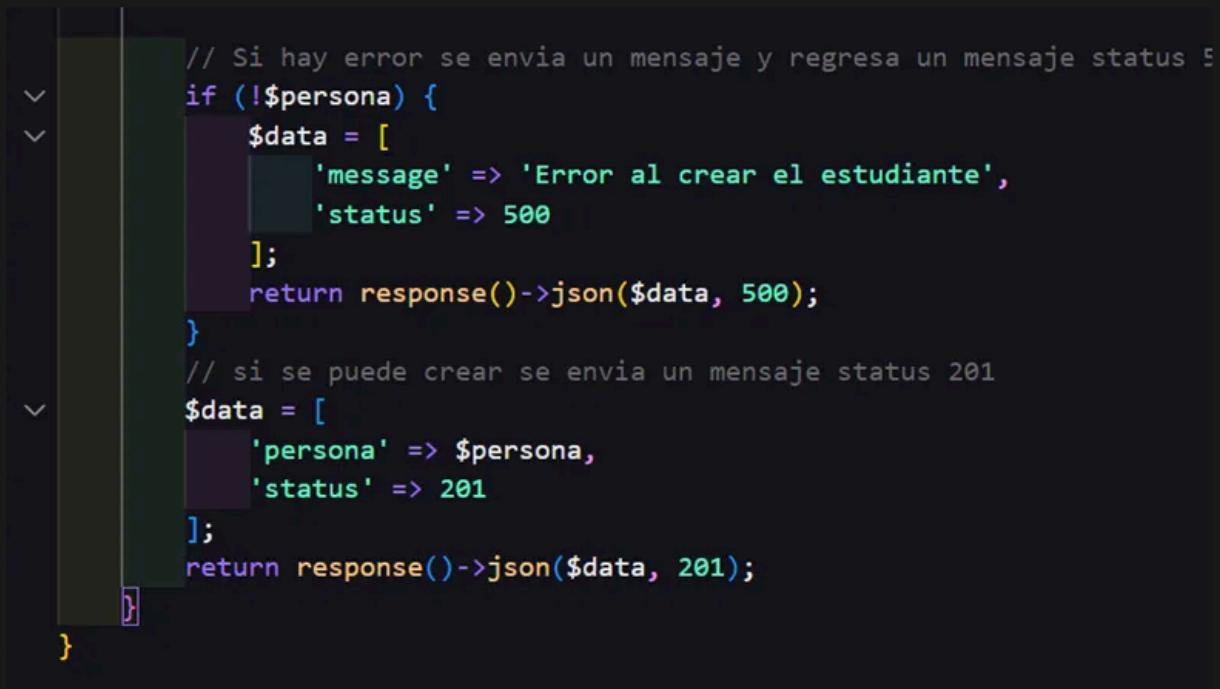
3. La función termina su ejecución aquí si hay errores

## Creación del registro en la base de datos

```
$persona = Persona::create(['identificacion' => $request->identificacion,
    'nombre' => $request->nombre, 'apellido' => $request->apellido, 'email' =>
    $request->email, 'telefono' => $request->telefono, 'direccion' =>
    $request->direccion]);
```

Si no hay errores de validación, el código continúa y crea un nuevo registro en la base de datos:

1. Utiliza el modelo `Persona` (que representa la tabla de personas en la base de datos)
2. El método `create()` inserta un nuevo registro utilizando los datos recibidos en la petición
3. El objeto creado se guarda en la variable `$persona`



```
// Si hay error se envia un mensaje y regresa un mensaje status 500
if (!$persona) {
    $data = [
        'message' => 'Error al crear el estudiante',
        'status' => 500
    ];
    return response()->json($data, 500);
}
// si se puede crear se envia un mensaje status 201
$data = [
    'persona' => $persona,
    'status' => 201
];
return response()->json($data, 201);
```

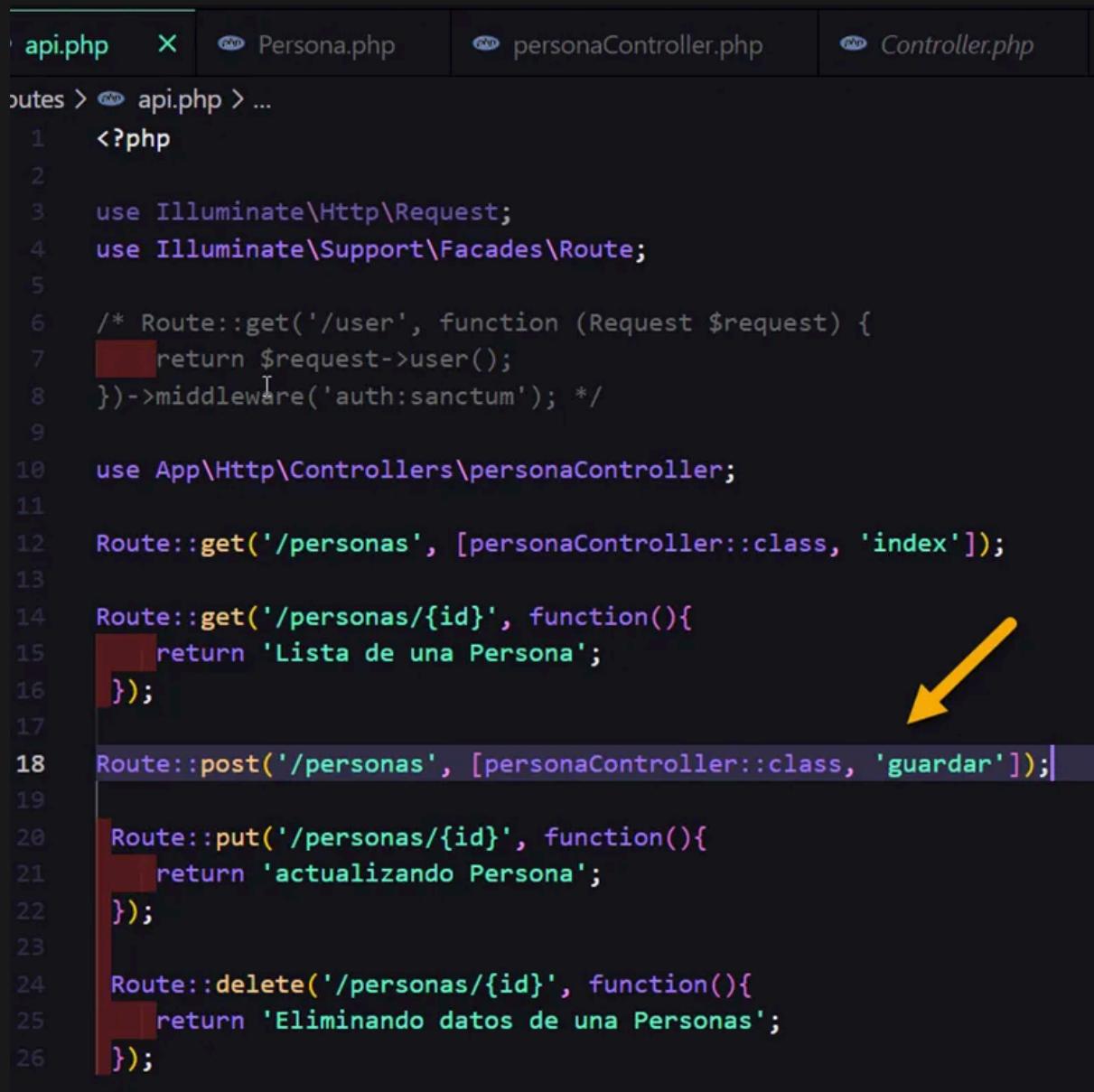
## Verificación de la creación y respuesta final

```
if (!$persona) { $data = [ 'message' => 'Error al crear el persona',
    'status' => 500 ]; return response()->json($data, 500); } $data = [
    'persona' => $persona, 'status' => 201 ]; return response()->json($data,
    201);
```

Esta última parte:

1. Verifica si la creación del registro falló ( !\$persona )
2. Si falló, devuelve un mensaje de error con código 500 (Error del Servidor)
3. Si la creación fue exitosa, devuelve el objeto \$persona creado junto con un código 201 (Created)

Vamos a llamar el método dentro del archivo api.php



```
api.php Personas.php personaController.php Controller.php
routes > api.php > ...
1 <?php
2
3 use Illuminate\Http\Request;
4 use Illuminate\Support\Facades\Route;
5
6 /* Route::get('/user', function (Request $request) {
7     return $request->user();
8 })->middleware('auth:sanctum'); */
9
10 use App\Http\Controllers\personaController;
11
12 Route::get('/personas', [personaController::class, 'index']);
13
14 Route::get('/personas/{id}', function(){
15     return 'Lista de una Persona';
16 });
17
18 Route::post('/personas', [personaController::class, 'guardar']);
19
20 Route::put('/personas/{id}', function(){
21     return 'actualizando Persona';
22 });
23
24 Route::delete('/personas/{id}', function(){
25     return 'Eliminando datos de una Personas';
26 });
```

Realizar prueba en Postman

The screenshot shows the Postman interface with a failed POST request to `http://127.0.0.1:8000/api/personas`. The response status is `400 Bad Request`. The error message in the body is:

```

1  {
2      "message": "Error en la validación de los datos",
3      "errors": [
4          "identificacion": [
5              "The identificacion field is required."
6          ],
7          "nombre": [
8              "The nombre field is required."
9          ],
10         "apellido": [
11             "The apellido field is required."
12         ],
13         "email": [
14             "The email field is required."
15         ]

```

Al pasar validaciones se guarda los datos de Persona

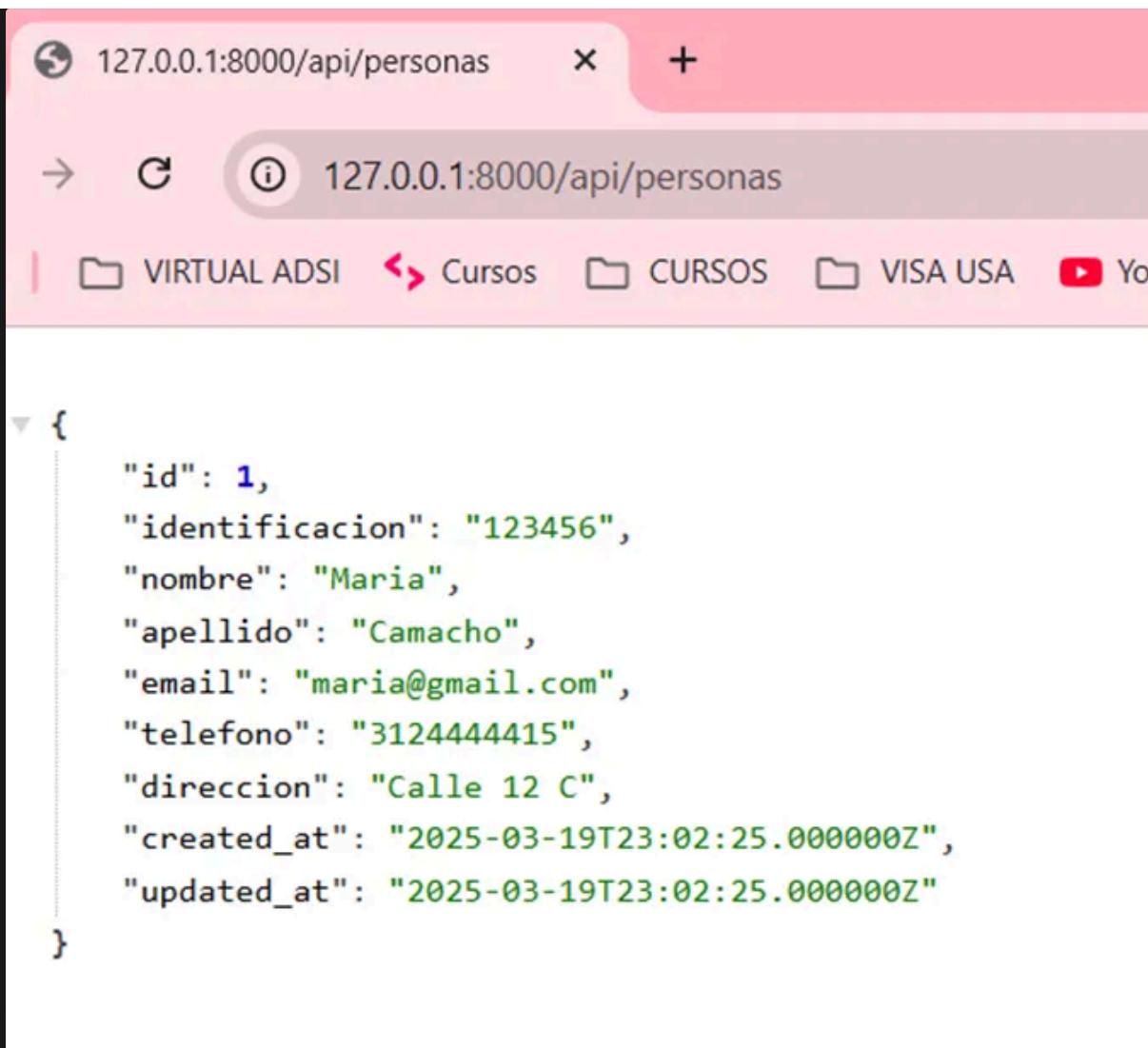
The screenshot shows the Postman interface with a successful POST request to `http://127.0.0.1:8000/api/personas`. The response status is `201 Created`. The JSON response body is:

```

{
    "id": 1,
    "identificacion": "123456",
    "nombre": "Maria",
    "apellido": "Camacho",
    "email": "maria@gmail.com",
    "telefono": "3124444415"
}

```

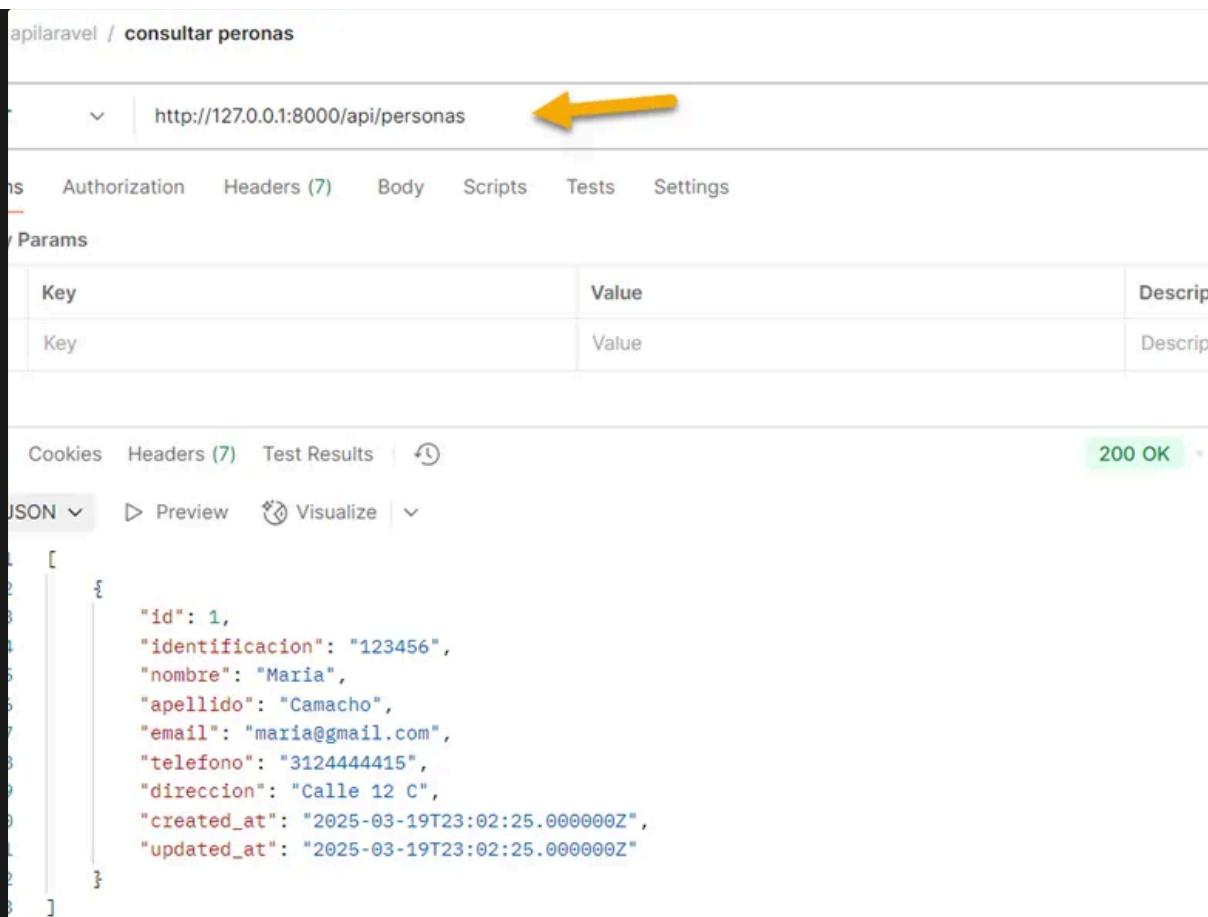
Al realizar la consulta en el navegador



The screenshot shows a browser window with the URL `127.0.0.1:8000/api/personas`. The page displays a single JSON object representing a person:

```
{  
    "id": 1,  
    "identificacion": "123456",  
    "nombre": "Maria",  
    "apellido": "Camacho",  
    "email": "maria@gmail.com",  
    "telefono": "3124444415",  
    "direccion": "Calle 12 C",  
    "created_at": "2025-03-19T23:02:25.000000Z",  
    "updated_at": "2025-03-19T23:02:25.000000Z"  
}
```

Validar en Postman



apilaravel / consultar peronas

http://127.0.0.1:8000/api/personas (arrow)

Authorization Headers (7) Body Scripts Tests Settings

Params

Key	Value	Description
Key	Value	Description

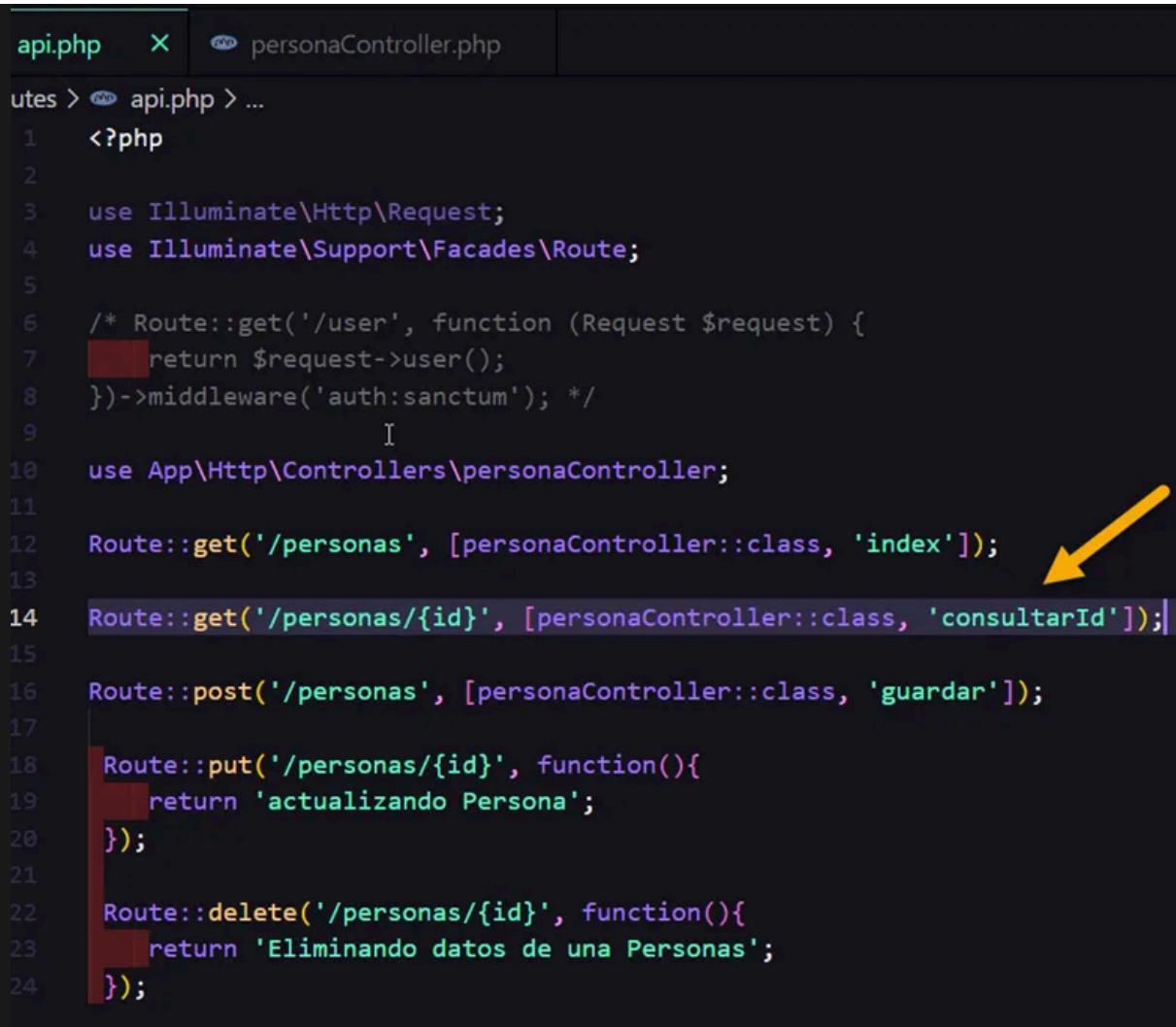
Cookies Headers (7) Test Results 200 OK

JSON ▾ ▶ Preview ⚡ Visualize ▾

```
[{"id": 1, "identificacion": "123456", "nombre": "Maria", "apellido": "Camacho", "email": "maria@gmail.com", "telefono": "3124444415", "direccion": "Calle 12 C", "created_at": "2025-03-19T23:02:25.000000Z", "updated_at": "2025-03-19T23:02:25.000000Z"}]
```

## 7.3 Creación del Método Consultar una Persona por el id

En el controlador llamamos el método consultarId

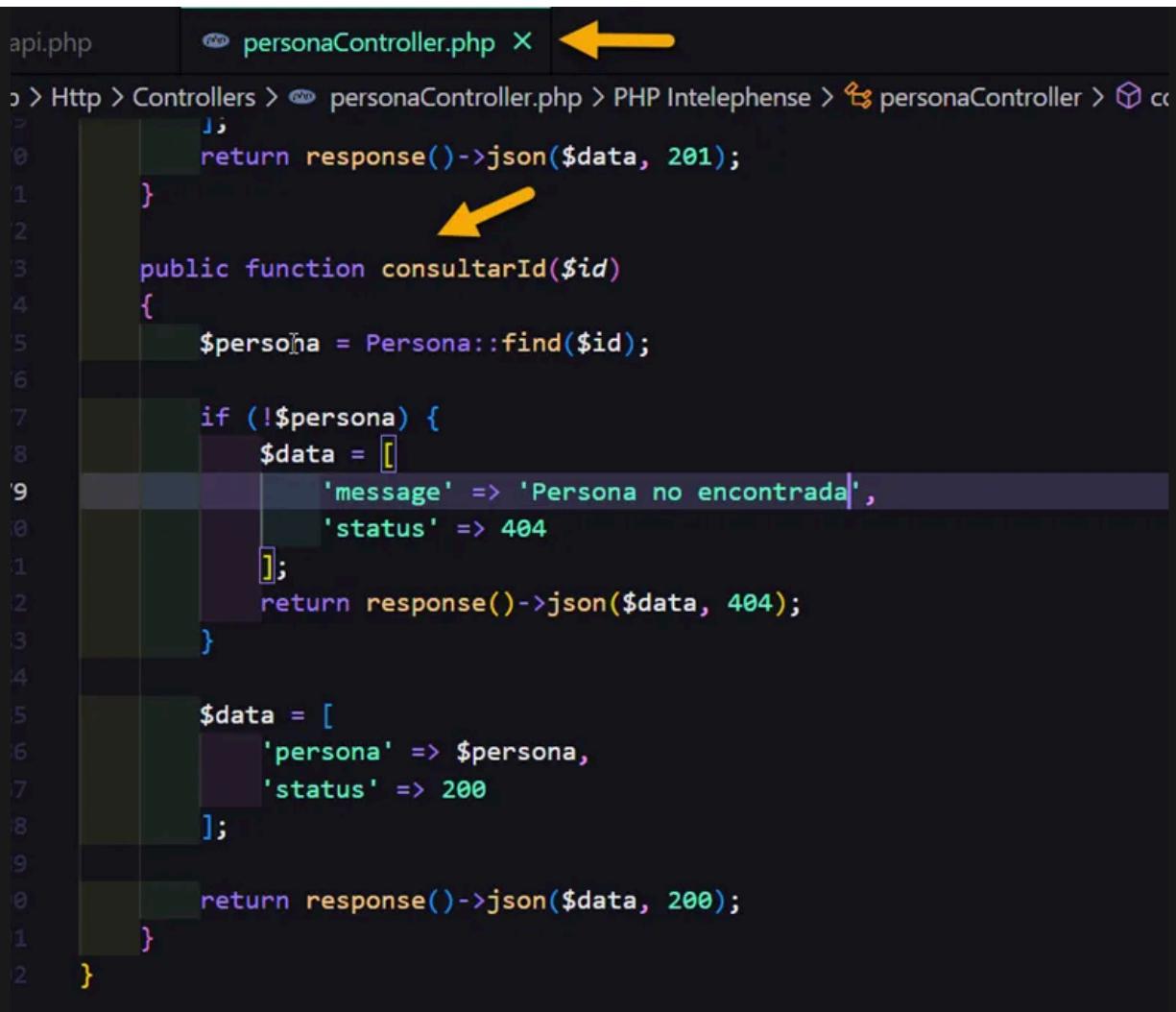


```
api.php  X  personaController.php
routes > api.php > ...
1  <?php
2
3  use Illuminate\Http\Request;
4  use Illuminate\Support\Facades\Route;
5
6  /* Route::get('/user', function (Request $request) {
7      return $request->user();
8  })->middleware('auth:sanctum'); */
9
10 use App\Http\Controllers\personaController;
11
12 Route::get('/personas', [personaController::class, 'index']);
13
14 Route::get('/personas/{id}', [personaController::class, 'consultarId']); ↓
15
16 Route::post('/personas', [personaController::class, 'guardar']);
17
18 Route::put('/personas/{id}', function(){
19     return 'actualizando Persona';
20 });
21
22 Route::delete('/personas/{id}', function(){
23     return 'Eliminando datos de una Personas';
24 });
```

## Explicación del método `consultarId`

Este método forma parte de lo que se conoce como un "endpoint" de API RESTful, específicamente para la operación "READ" (leer) de un recurso individual dentro del patrón CRUD (Create, Read, Update, Delete).

En una aplicación web, este método sería invocado cuando un cliente (como una aplicación móvil o una página web) necesita obtener información detallada sobre una persona específica.



```
api.php personaController.php X ←
o > Http > Controllers > personaController.php > PHP Intelephense > personaController > cc
  0
  1     return response()->json($data, 201);
  2
  3     public function consultarId($id)
  4     {
  5         $persona = Persona::find($id);
  6
  7         if (!$persona) {
  8             $data = [
  9                 'message' => 'Persona no encontrada',
 10                 'status' => 404
 11             ];
 12             return response()->json($data, 404);
 13         }
 14
 15         $data = [
 16             'persona' => $persona,
 17             'status' => 200
 18         ];
 19
 20         return response()->json($data, 200);
 21     }
 22 }
```

Este método `consultarId`, que forma parte de un controlador en Laravel y se encarga de buscar una persona en la base de datos a partir de su identificador único.

```
public function consultarId($id)
```

Este es un método público que recibe un parámetro `$id`, que representa el identificador único de la persona que se desea consultar. En bases de datos, este identificador suele corresponder a la clave primaria del registro en la tabla de personas.

## Búsqueda en la base de datos

```
$persona = Persona::find($id);
```

Esta línea utiliza el modelo `Persona` (que es una representación de la tabla de personas en la base de datos) para buscar un registro específico. El método `find()` es proporcionado por Eloquent, el ORM (Object-Relational Mapping) de Laravel, y busca automáticamente un registro que tenga como clave primaria el valor de `$id` proporcionado.

Si encuentra un registro con ese identificador, devuelve un objeto con todos los datos de esa persona. Si no encuentra ningún registro, devuelve `null`.

#### Verificación de resultados y manejo de error

```
if (!$persona) { $data = [ 'message' => 'Persona no encontrada', 'status' => 404 ]; return response()->json($data, 404); }
```

Esta sección verifica si la búsqueda no encontró ningún resultado (`!$persona` evalúa a `true` cuando `$persona` es `null`). En caso de que la persona no exista:

1. Se crea un array asociativo `$data` con dos elementos:
  - Un mensaje descriptivo: "Persona no encontrada"
  - Un código de estado: 404 (que en HTTP significa "Not Found" o "No encontrado")
2. Se devuelve una respuesta JSON con esa información y el código HTTP 404
3. La función termina su ejecución en este punto, evitando que se ejecute el código siguiente

El código 404 es el estándar en APIs RESTful para indicar que el recurso solicitado no existe, lo cual es apropiado cuando el identificador proporcionado no corresponde a ningún registro en la base de datos.

```
$data = [ 'persona' => $persona, 'status' => 200 ]; return response()->json($data, 200);
```

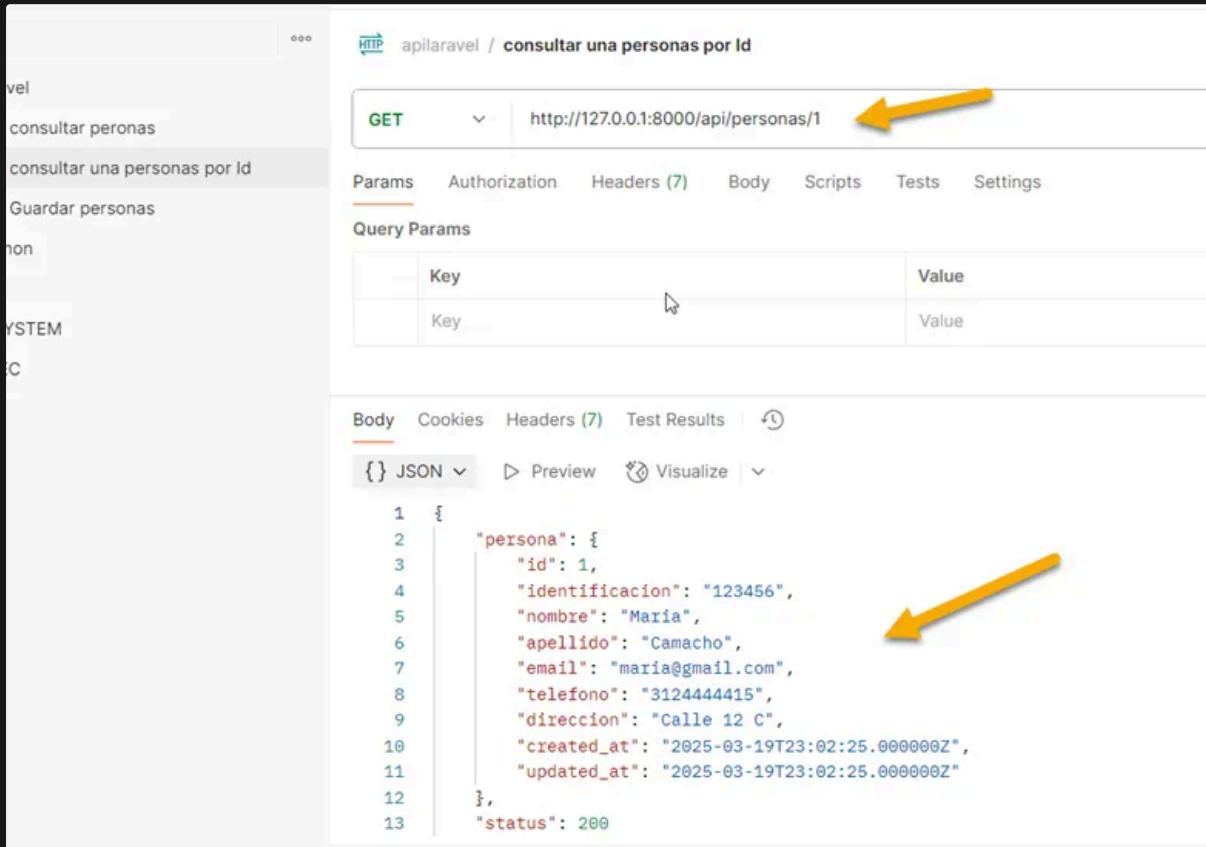
Si el código llega a este punto, significa que se encontró una persona con el identificador solicitado. Entonces:

1. Se crea un array asociativo `$data` con:
  - El objeto `$persona` que contiene todos los datos del registro encontrado
  - Un código de estado: 200 (que en HTTP significa "OK" o "Correcto")

2. Se devuelve una respuesta JSON que contiene esa información y el código HTTP 200

El código 200 es el estándar en APIs RESTful para indicar que la solicitud se ha procesado correctamente y se devuelve la información solicitada.

- Crear el método consultarID
- Prueba en Postman



The screenshot shows the Postman interface. In the top right, there is a search bar with the URL `http://127.0.0.1:8000/api/personas/1`. Two yellow arrows point to this URL: one from the left and another from the bottom right. Below the URL, under the 'Headers' tab, there are seven items listed. In the bottom section, the 'Body' tab is selected, showing a JSON response with the following content:

```

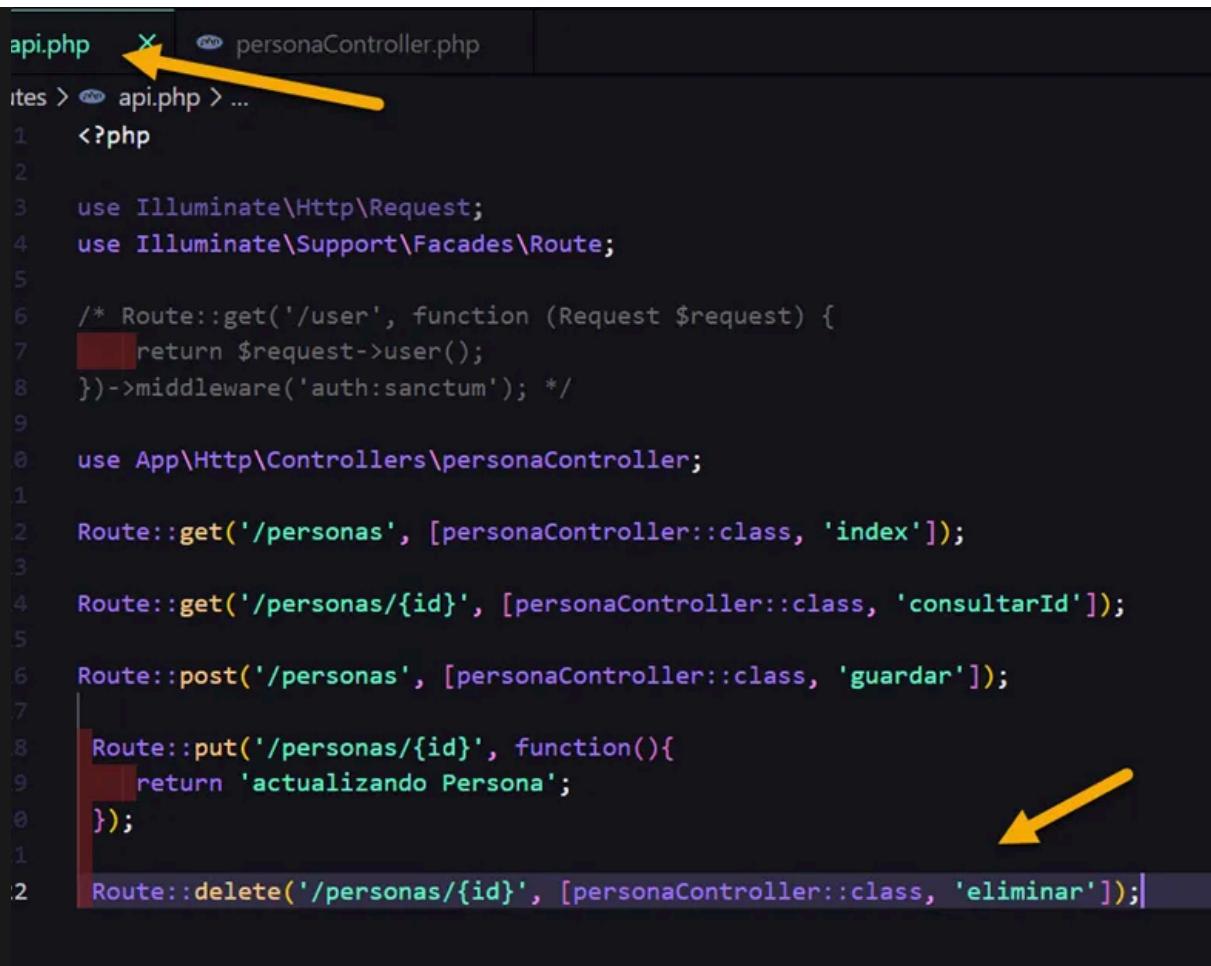
1 {
2   "persona": {
3     "id": 1,
4     "identificacion": "123456",
5     "nombre": "Maria",
6     "apellido": "Camacho",
7     "email": "maria@gmail.com",
8     "telefono": "3124444415",
9     "direccion": "Calle 12 C",
10    "created_at": "2025-03-19T23:02:25.000000Z",
11    "updated_at": "2025-03-19T23:02:25.000000Z"
12  },
13  "status": 200

```

## Encabezado 3

### 7.4 Creación del Método Borrar una persona por id

En el controlador llamamos el método eliminar



```
api.php  X personaController.php
sites > api.php > ...
1 <?php
2
3 use Illuminate\Http\Request;
4 use Illuminate\Support\Facades\Route;
5
6 /* Route::get('/user', function (Request $request) {
7     return $request->user();
8 })->middleware('auth:sanctum'); */
9
10 use App\Http\Controllers\personaController;
11
12 Route::get('/personas', [personaController::class, 'index']);
13
14 Route::get('/personas/{id}', [personaController::class, 'consultarId']);
15
16 Route::post('/personas', [personaController::class, 'guardar']);
17
18 Route::put('/personas/{id}', function(){
19     return 'actualizando Persona';
20 });
21
22 Route::delete('/personas/{id}', [personaController::class, 'eliminar']);|
```

## Explicación del método **eliminar**

Colocamos el método eliminar en el controlador

```
  p |  ➔ personaController.php X ←
  tp > Controllers > ➔ personaController.php > PHP Intelephense > 📁 personaController > ⚙️ e
      return response()->json($data, 200);
  }

  // Método para eliminar una persona

  public function eliminar($id) ←
  {
    $persona = Persona::find($id);

    if (!$persona) {
      $data = [
        'message' => 'Persona no encontrado',
        'status' => 404
      ];
      return response()->json($data, 404);
    }

    $persona->delete();

    $data = [
      'message' => 'Persona eliminada',
      'status' => 200
    ];

    return response()->json($data, 200);
}
```

## Definición del método

```
public function eliminar($id)
```

Este es un método público que recibe un único parámetro `$id`, que representa el identificador único de la persona que se desea eliminar. Al igual que en el método `consultarId` que vimos anteriormente, este identificador corresponde típicamente a la clave primaria del registro en la tabla de personas.

## Búsqueda previa en la base de datos

```
$persona = Persona::find($id);
```

Antes de poder eliminar un registro, primero debemos asegurarnos de que existe. Esta línea utiliza el modelo `Persona` para buscar un registro con el identificador proporcionado. El método `find()` de Eloquent (el ORM de Laravel) busca automáticamente por la clave primaria y devuelve:

- Un objeto con los datos de la persona si se encuentra
- `null` si no existe ningún registro con ese identificador

## Verificación de existencia y manejo de error

```
if (!$persona) { $data = [ 'message' => 'Persona no encontrado', 'status' => 404 ]; return response()->json($data, 404); }
```

Esta sección verifica si el registro a eliminar no fue encontrado. La expresión `!$persona` evalúa a `true` cuando `$persona` es `null`, lo que indica que no se encontró ninguna persona con ese identificador.

Si la persona no existe, el método:

1. Crea un array con un mensaje de error y un código de estado 404
2. Devuelve una respuesta JSON con esa información y el código HTTP 404 (Not Found)
3. Termina la ejecución del método en este punto

Esta verificación es crucial porque no podemos eliminar algo que no existe. El código 404 es el estándar HTTP para indicar que el recurso solicitado no existe, lo que comunica claramente al cliente que intentó eliminar un registro inexistente.

## Eliminación del registro

```
$persona->delete();
```

Si el código llega a este punto, significa que sí se encontró una persona con el identificador proporcionado. Esta línea llama al método `delete()` en el objeto `$persona`.

Cuando se invoca `delete()` en un modelo de Eloquent:

1. Laravel genera y ejecuta una consulta SQL DELETE para eliminar el registro correspondiente de la base de datos
2. Se disparan eventos antes y después de la eliminación (si están configurados)
3. Si existen relaciones con otras tablas, Laravel puede manejar automáticamente la eliminación en cascada o restricciones, dependiendo de cómo esté configurado el modelo

## Preparación y envío de respuesta exitosa

```
$data = [ 'message' => 'Persona eliminada', 'status' => 200 ]; return response()->json($data, 200);
```

Una vez eliminado con éxito el registro, el método:

1. Crea un array con un mensaje de confirmación y un código de estado 200
2. Devuelve una respuesta JSON con esa información y el código HTTP 200 (OK)

El código 200 indica que la operación se completó con éxito. A diferencia del método `guardar` que devolvía el objeto creado, aquí sólo se devuelve un mensaje de confirmación, ya que el objeto ha sido eliminado y ya no existe para ser devuelto.

## El flujo completo: buscar, verificar, eliminar, confirmar

Lo interesante de este método es cómo sigue un flujo lógico que garantiza la integridad de la operación:

1. **Buscar:** Primero intenta recuperar el registro que se desea eliminar
2. **Verificar:** Comprueba si el registro existe antes de intentar eliminarlo
3. **Eliminar:** Solo si existe, procede a eliminarlo de la base de datos
4. **Confirmar:** Comunica el resultado de la operación al cliente

Este enfoque previene errores que podrían ocurrir si intentáramos eliminar directamente sin verificar la existencia previa del registro.

## 7.5 Creación del Método Actualizar (PUT) una persona por id

Vamos a crear el método Actualizar una persona por id

```
public function actualizar(Request $request, $id) { $persona = Persona::find($id); if (!$persona) { $data = [ 'message' => 'Persona no encontrada', 'status' => 404 ]; return response()->json($data, 404); } $validator = Validator::make($request->all(), [ 'identificacion' => 'required|max:20', 'nombre' => 'required|max:255', 'apellido' => 'required|max:255', 'email' => 'required|email', 'telefono' => 'required|max:13', 'direccion' => 'required|max:255' ]); if ($validator->fails()) { $data = [ 'message' => 'Error en la validación de los datos', 'errors' => $validator->errors(), 'status' => 400 ]; return response()->json($data, 400); } $persona->identificacion = $request->identificacion; $persona->nombre = $request->nombre; $persona->apellido = $request->apellido; $persona->email = $request->email; $persona->telefono = $request->telefono; $persona->direccion = $request->direccion; $persona->save(); $data = [ 'message' => 'Datos actualizados de Persona', 'persona' => $persona, 'status' => 200 ]; return response()->json($data, 200); }
```

## Método `actualizar` : Explicación

### Definición y parámetros

```
public function actualizar(Request $request, $id)
```

Este método público recibe dos parámetros:

1. `$request` : Un objeto de tipo `Request` que contiene todos los datos enviados en la petición HTTP (como datos de formularios o JSON)
2. `$id` : El identificador único de la persona cuyos datos queremos actualizar

A diferencia del método `guardar`, aquí necesitamos tanto los nuevos datos como la identificación del registro existente que vamos a modificar.

### Verificación de existencia del registro

```
php Copiar $persona = Persona::find($id); if (!$persona) { $data = [ 'message' => 'Persona no encontrada', 'status' => 404 ]; return response()->json($data, 404); }
```

Esta primera sección realiza una verificación crucial: antes de intentar actualizar datos, debemos comprobar si el registro existe. El flujo es:

1. Buscamos la persona por su identificador usando el método `find`
2. Si no se encuentra (`!$persona` es verdadero), respondemos con un error 404 (Not Found)
3. Incluimos un mensaje claro: "Persona no encontrada"
4. Terminamos la ejecución en este punto, evitando procesamiento innecesario

Es importante realizar esta verificación al inicio del método porque no tiene sentido validar datos ni intentar actualizaciones si el registro objetivo no existe.

## Validación de los datos recibidos

```
php Copiar $validator = Validator::make($request->all(), [ 'identificacion' => 'required|max:20', 'nombre' => 'required|max:255', 'apellido' => 'required|max:255', 'email' => 'required|email', 'telefono' => 'required|max:13', 'direccion' => 'required|max:255' ]); if ($validator->fails()) { $data = [ 'message' => 'Error en la validación de los datos', 'errors' => $validator->errors(), 'status' => 400 ]; return response()->json($data, 400); }
```

La validación en este método es idéntica a la que vimos en el método `guardar`.

Esto muestra una buena práctica de consistencia: las mismas reglas se aplican tanto al crear como al actualizar datos. El proceso es:

1. Creamos un validador que revisa todos los campos según reglas específicas
2. Cada campo debe cumplir con:
  - `identificacion`: obligatorio y máximo 20 caracteres
  - `nombre` y `apellido`: obligatorios y máximo 255 caracteres
  - `email`: obligatorio y debe ser un email válido
  - `telefono`: obligatorio y máximo 13 caracteres
  - `direccion`: obligatorio y máximo 255 caracteres
3. Si alguna validación falla, respondemos con:
  - Un mensaje general: "Error en la validación de los datos"
  - Los errores específicos generados por el validador
  - Un código de estado 400 (Bad Request)

4. Terminamos la ejecución si hay errores de validación

## Actualización de los datos

```
php Copiar $persona->identificacion = $request->identificacion; $persona->nombre = $request->nombre; $persona->apellido = $request->apellido; $persona->email = $request->email; $persona->telefono = $request->telefono; $persona->direccion = $request->direccion; $persona->save();
```

Aquí está la diferencia principal con el método `guardar`. En lugar de usar `Persona::create()` para insertar un nuevo registro, estamos:

1. Asignando cada propiedad del objeto `$persona` (que ya existe) con los nuevos valores del `$request`
2. Llamando al método `save()` para guardar los cambios en la base de datos

Esto es lo que técnicamente se conoce como "actualización por asignación de atributos", donde modificamos propiedades individuales y luego guardamos. Es importante entender que `save()` aquí está actualizando un registro existente, no creando uno nuevo.

Observa que se actualizan todos los campos, independientemente de si cambiaron o no. En aplicaciones más complejas, podríamos implementar actualizaciones parciales (PATCH) donde solo se modifican los campos proporcionados.

## Respuesta al cliente

```
php Copiar $data = [ 'message' => 'Datos actualizados de Persona', 'persona' => $persona, 'status' => 200 ]; return response()->json($data, 200);
```

Finalmente, tras guardar exitosamente los cambios:

1. Creamos un array con:
  - Un mensaje descriptivo: "Datos actualizados de Persona"
  - El objeto `$persona` actualizado (que ahora contiene los datos modificados)
  - Un código de estado 200 (OK)
2. Devolvemos estos datos como una respuesta JSON

Incluir el objeto actualizado en la respuesta es una buena práctica porque:

- Confirma al cliente exactamente qué datos quedaron guardados
- Proporciona cualquier valor calculado o modificado por el servidor
- Facilita la sincronización entre el cliente y el servidor

## Comparación entre los métodos `guardar` y `actualizar`

Aunque ambos métodos modifican datos, existen diferencias importantes:

Aspecto	<code>guardar</code>	<code>actualizar</code>
Operación	Crea nuevo registro	Modifica registro existente
Parámetros	Solo <code>\$request</code>	<code>\$request</code> y <code>\$id</code>
Verificación inicial	No verifica existencia	Verifica si el registro existe
Método Eloquent	<code>Persona::create()</code>	<code>\$persona-&gt;save()</code>
Respuesta exitosa	Código 201 (Created)	Código 200 (OK)

Esta diferenciación sigue las convenciones RESTful, donde distintos métodos HTTP (POST para crear, PUT/PATCH para actualizar) tienen diferentes semánticas y códigos de respuesta.

- Crear Método actualizar por id
- Llamar el método Actualizar en el controlador

```
api.php  X  personaController.php
routes > api.php
1  <?php
2
3  use Illuminate\Http\Request;
4  use Illuminate\Support\Facades\Route;
5
6  /* Route::get('/user', function (Request $request) {
7      return $request->user();
8  })->middleware('auth:sanctum'); */
9
10 use App\Http\Controllers\personaController;
11
12 Route::get('/personas', [personaController::class, 'index']);
13
14 Route::get('/personas/{id}', [personaController::class, 'consultarId']);
15
16 Route::post('/personas', [personaController::class, 'guardar']);
17
18  Route::put('/personas/{id}', [personaController::class, 'actualizar']);
19
20 Route::delete('/personas/{id}', [personaController::class, 'eliminar']);
```

- Realizar prueba en Postman con id que no Existe

The screenshot shows a Postman collection named 'apilaravel / Guardar personas Copy'. A yellow arrow points to the URL field containing 'http://127.0.0.1:8000/api/personas/1'. Another yellow arrow points to the response body, which displays a JSON object with a message and status code:

```
1 {  
2   "message": "Persona no encontrada",  
3   "status": 404  
4 }
```

- Realizar prueba en Postman con id que Existe y validando campos

apilaravel / Guardar personas Copy

PUT  [Yellow Arrow]

Params Authorization Headers (8) Body Scripts Tests Settings

Query Params

Key	Value
Key	Value

Body Cookies Headers (7) Test Results [Yellow Arrow]

{ } JSON [Yellow Arrow] Preview Visualize

```
1 {  
2     "message": "Error en la validación de los datos",  
3     "errors": {  
4         "identificacion": [  
5             "The identificacion field is required."  
6         ],  
7         "nombre": [  
8             "The nombre field is required."  
9         ]  
10    }  
11}
```

Realizar prueba en Postman con id que Existe y actualizando campos

The screenshot shows a POST request in Postman to `http://127.0.0.1:8000/api/personas/2`. The `Body` tab is selected, showing a JSON payload:

```
{
  "identificacion": "1234567",
  "nombre": "Maria",
  "apellido": "Camacho",
  "email": "marialuz@gmail.com",
  "telefono": "3124444415",
  "direccion": "Calle 12 C"
}
```

The response status is `200 OK` with a response time of `143 ms` and a size of `525 B`. The response body is displayed in JSON format:

```
{
  "message": "Datos actualizados de Persona",
  "persona": {
    "id": 9,
    "identificacion": "1234567",
    "nombre": "Maria",
    "apellido": "Camacho",
    "email": "marialuz@gmail.com",
    "telefono": "3124444415",
    "direccion": "Calle 12 C"
  },
  "status": 200
}
```

## 7.5 Creación del Método Actualizar (PATCH) una persona por id

Vamos a crear el método Actualizar parcialmente datos de una persona por id

```
public function actualizarParcial(Request $request, $id) { $persona = Persona::find($id); if (!$persona) { $data = [ 'message' => 'Persona no encontrada', 'status' => 404 ]; return response()->json($data, 404); } $validator = Validator::make($request->all(), [ 'identificacion' => 'max:20', 'nombre' => 'max:255', 'apellido' => 'max:255', 'email' => 'email', 'telefono' => 'max:13', 'direccion' => 'max:255' ]); if ($validator->fails()) { $data = [ 'message' => 'Error en la validación de los datos', 'errors' => $validator->errors(), 'status' => 400 ]; return response()->json($data, 400); } if ($request->has('identificacion')) { $persona->identificacion = $request->identificacion; } if ($request->has('nombre')) { $persona->nombre = $request->nombre; } if ($request->has('apellido')) { $persona->apellido = $request->apellido; } if ($request->has('email')) { $persona->email = $request->email; } if ($request->has('telefono')) { $persona->telefono = $request->telefono; } if ($request->has('direccion')) { $persona->direccion = $request->direccion; } $persona->save(); $data = [ 'message' => 'Persona actualizada', 'persona' => $persona, 'status' => 200 ]; return response()->json($data, 200); }
```

# Explicación del método `actualizarParcial`

El método `actualizarParcial` implementa lo que en APIs RESTful se conoce como una operación PATCH - una actualización parcial que permite modificar solo algunos campos de un recurso sin tener que enviar todos sus datos. Vamos a desglosarlo para entender cómo funciona esta técnica más flexible de actualización.

## Definición y propósito

```
public function actualizarParcial(Request $request, $id)
```

Este método recibe:

- Un objeto `$request` con los datos que desean actualizarse (que pueden ser solo algunos campos)
- El `$id` que identifica el registro de persona a actualizar

A diferencia del método `actualizar` que vimos anteriormente, este está diseñado específicamente para permitir actualizaciones de campos individuales, sin necesidad de enviar todos los datos del recurso.