

ÍNDICE DE CONTENIDOS

Concepto de API

Realizando un API

El controlador

La base de datos

Los métodos

El cliente:
PostMan

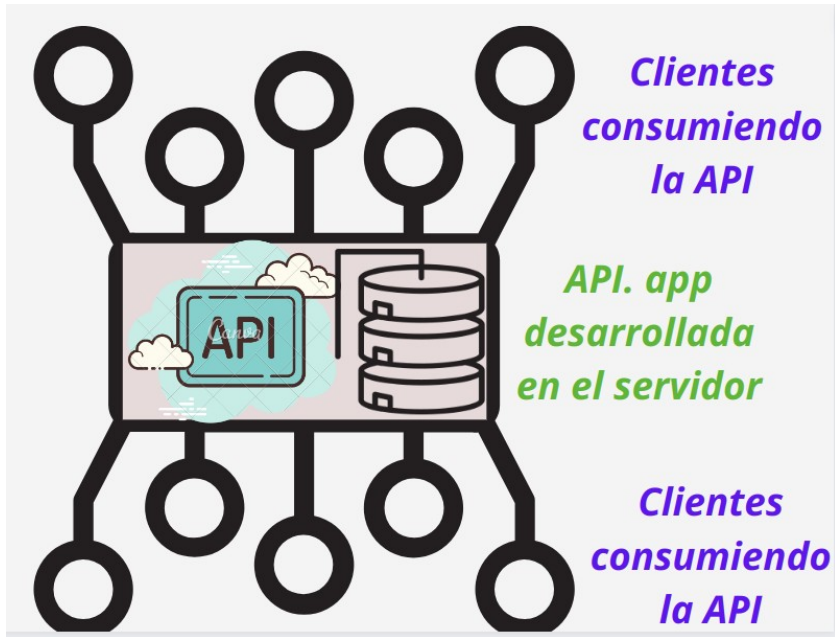
Pruebas unitarias



API(Application Programming Interface)



Un **API** (*Interfaz de Programación de Aplicaciones*) es un conjunto de reglas y protocolos que permiten que diferentes aplicaciones se comuniquen entre sí



Conceptos básicos sobre API estandarizada

Debemos tener claro algunos conceptos.

- Hacer una API, es algo muy sencillo, pero debemos ser conscientes de que estamos haciendo algo para que **lo use cualquier aplicación**
 - Esto implica cumplir unos **estándares establecidos.**
 - Vamos a repasar estos 5 conceptos
 - API
 - REST
 - VERBOS HTTP
 - RECURSOS
 - CÓDIGOS DE ESTADO
-

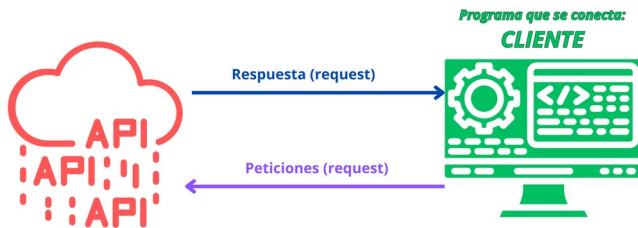
API(Application Programming Interface)



Una API es un programa que va a permitir que otros programas se comuniquen con él

Una API proporcionar una interfaz estandarizada; esto permite integrar y utilizar servicios externos de manera eficiente y segura, sin tener que preocuparse por los detalles de implementación subyacentes.

Lo que vamos a crear es un conjunto de endpoints o URLs que permitirán a los clientes (como Postman u otras aplicaciones) realizar solicitudes HTTP para acceder y manipular los datos.



API REST (REpresentation State Transfer)



Si una **API es REST**, quiere decir que se va a basar o respetar **6 principios de implementación**.

REST es una recomendación, no es un estándar ni un protocolo, como lo puede ser SOAP

Lo podemos ver como una arquitectura de construcción

REST (Principios de implementación)

1.- Arquitectura Cliente-Servidor:

- separa la interfaz de usuario y la lógica del usuario (cliente) de la lógica del servidor. O sea que el API y el CLIENTE son dos entes independientes

2.- Sin Estado (Stateless):

- Cada solicitud del cliente al servidor debe contener toda la información necesaria para entender y procesar la solicitud. El estado de la sesión del usuario se mantiene en el cliente (usando tokens)

3.- Cacheable (Almacenamiento en Caché):

4.- Sistema de Capas (Layered System):

5.- Código Bajo Demanda (Code on Demand) [opcional]:

6.- Interfaz Uniforme:

- Hay que saber cómo solicitar la ejecución del API:
 - **Identificación de Recursos:** Cada recurso (información o servicio) se identifica mediante un URI (Uniform Resource Identifier).
 - **Manipulación de Recursos a través de Representaciones:** Los recursos pueden ser representados y manipulados en diferentes formatos, como JSON o XML.
 - **Mensajes Autodescriptivos:** Cada mensaje incluye suficiente información para describir cómo procesar la solicitud.
 - HATEOAS (Hypertext As The Engine Of Application State): **El servidor debe de facilitar información que nos diga cómo navegar por la API, no solo facilitar datos**

https://proyectosdwa.es/manuel/api/docs/4_api/api/

01 Rest Full

Una api es Rest full cuando además de utilizar las 6 restricciones, utiliza el protocolo **http** para su implementación .

http es un protocolo que atiende a solicitudes según el verbo o palabra especial por el que solicitan; los que usaremos son :

- **GET** : Para solicitar un recurso o lista de recursos
- **POST**: Para enviar un recurso al servidor que queremos crear
- **DELETE**: Para eliminar un recurso
- **PUT**: Para sustituir un recurso por otro
- **PATCH**: Para modificar algún valor de un recurso

Recurso

Es una entidad o concepto que podemos nombrar e identificar y corresponde generalmente a los registros que hay en las talbas: **Usuarios, Profesores, Proyectos**

Representación

La forma concreta en la que se puede ver un recurso para enviarlo o recibirlo. Generalmente va a ser un JSON

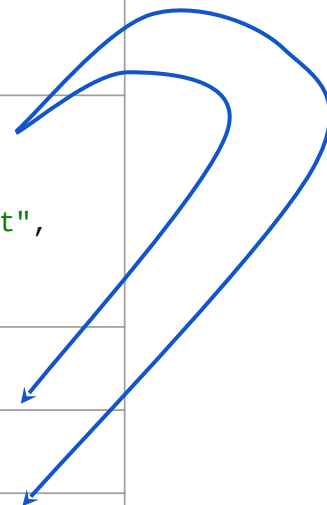
01 Peticiones HTTP

Método

URI (http://server/

Payload o cuerpo

GET	Obtener elementos	teachers	
	Obtener un elemento	teachers/{id}	
POST	Insertar un recurso	teachers	<pre>{ "id": 1, "nombre": "María Ruiz", "email": "maria@gmail.net", "proyecto_id": 1 },</pre>
DELETE	Eliminar un recurso	teachers/{id}	
PATCH	Actualizar un recurso	teachers/{id}	
PUT	Reemplazar un recurso	teachers/{id}	



Respuestas HTTP

El servidor entregará la respuesta junto con un **código de estado (lo más usado)**

código	Tipo de información	Tipos de códigos más concretos
1xx	Información	
2xx	ÉXITO	200 ok 201 Created 204 No content
3xx	REDIRECCIONAMIENTO	301 Movido 302 y 303 otra url 307 Redirección temporal 308 Redirección permanente
4xx	ERROR EN EL CLIENTE	400 Mala solicitud 401 Unauthorized 403 Forbidden
5xx	ERROR EN EL SERVIDOR	500 Error del Servidor 502 Puerta GateGay incorrecta 503 Servicio no alcanzable

Payload de la respuesta: el Json

La respuesta puede estar en un json con diferente estructura. Todas correctas si son conocidas y gestionadas por el cliente

```
{
  "object": "articles",
  "data": {
    "title": "Mi artículo",
    "body": "Contenido"
  }
}
```

```
{
  "status": "success",
  "data": [
    {
      "title": "Mi artículo",
      "body": "Contenido"
    }
  ]
}
```

```
{
  "message": "success",
  "article": {
    "title": "Mi artículo",
    "body": "Contenido"
  }
}
```

```
{
  "message": "OK",
  "result": [
    {
      "title": "Mi artículo",
      "body": "Contenido"
    }
  ]
}
```

01 JSON:API Specification

Es un intento de estandarizar la respuesta json, la comunicación entre cliente y servidor

Cómo se debe de solicitar un recurso para obtener o modificar(request)

Cómo un servidor debe de responder a esa solicitud

Estructura del json,:

- **data y errors** excluyentes entre sí
- data es el principal y tendrá más contenido
- **meta** información libre
- included
- links
- jsonapi

```
{  
  "data": [],  
  "errors": [],  
  "meta": [],  
  "included": [],  
  "links": [],  
  "jsonapi": []  
}
```

JSON:API Specification: data

```
"data": {  
  "type": "project",  
  "id": (string)"1",  
  "attributes": {  
    "id": "1",  
    "titulo": "....",  
  },  
  "relationships": {  
    "users": {  
      "data": {  
        "type": "users",  
        "id": "3"  
      }  
    },  
    "teachers": { // . . .  
  },  
  "links": {  
    "self": "http://localhost:8000/projects/1"  
  },  
  "meta": {  
  }  
}
```

01 JSON:API Specification

```
{  
  "data": [],  
  "errors": [],  
  "jsonapi": []  
}
```

○ **data** (OK) ○ **errors** (No ok)

01 Nuestro JSON

data

```
"type": "project",
"id": (string)"1",
"attributes": {
  "id": "1",
  "titulo": "....",
  //resto de campos
},
"links": {
  "self": "http://localhost:8000/projects/1"
},
```

01 Nuestro JSON

**error
s**

```
"errors": [  
  {  
    "status": "422", //Cada código  
    "title": "Fallo se validación" //En función del código  
    "details": "detalles para comprender el error"  
    "source": {  
      "pointer": "/data/attributes/campo"  
    }  
  }  
]
```

01 Nuestro JSON

jsonapi

```
"jsonapi": {  
  "version": "1.0"  
}
```

01 Parámetros en la url

- **include:**

http://localhost/api/users?include=projects

Qué relación tiene que incluir en la respuesta

- **sort**

http://localhost/api/users?sort=-created-at, name

Criterio de ordenación de izquierda a derecha, un menos delante del campo invierte el orden

- **sparse fieldsets**

http://localhost/api/users?fields [users]=name

Qué atributos queremos pedir

- **filter**

http://localhost/api/users?filter[name]=María

- **page**

Realiza paginación, es decir qué página (number)quiero de toda la consulta y cuántos registros por página (size)

http://localhost/api/users?page[size]=15&page[number]=4

Comunicación entre cliente y API: Content Negotiation

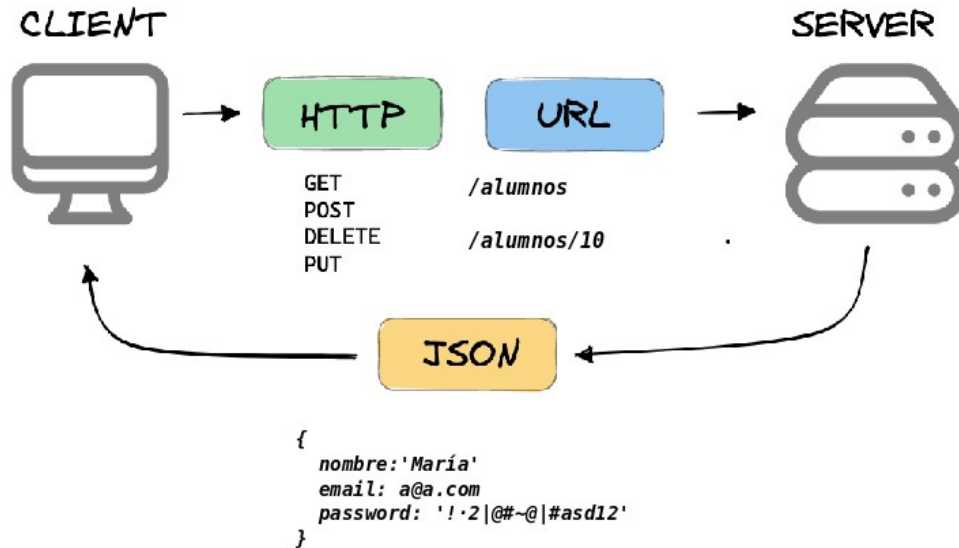
Tanto la solicitud, como la respuesta deben contener el header

Content-Type": "application/vnd.api+json"

Request Type	Method(s)	Header	Status Code
<i>Request</i>	GET, POST, PATCH, DELETE	Accept: application/vnd.api+json	406 Not Acceptable
<i>Request</i>	POST, PATCH	Content-Type: application/vnd.api+json	415 Unsupported Media Type
<i>Response</i>	All	Content-Type: application/vnd.api+json	415 Unsupported Media Type

API(Application Programming Interface)

WHAT IS A REST API?



La estructura del API va a estar estructurada de acuerdo con los principios RESTfull, facilitando su uso, escalabilidad y simplicidad, tanto en el uso, como en su construcción.

En un API RESTful, los recursos (como los datos de los alumnos) se representan mediante URLs y se manipulan a través de los verbos HTTP estándar, como GET, POST, PUT y DELETE.

Cada recurso debe tener una URL única que permita su identificación y acceso.

Con los verbos HTTP se podrá realizar operaciones como recuperar información de los alumnos, agregar nuevos alumnos, actualizar sus datos y eliminar registros existentes.

02 API: Realización

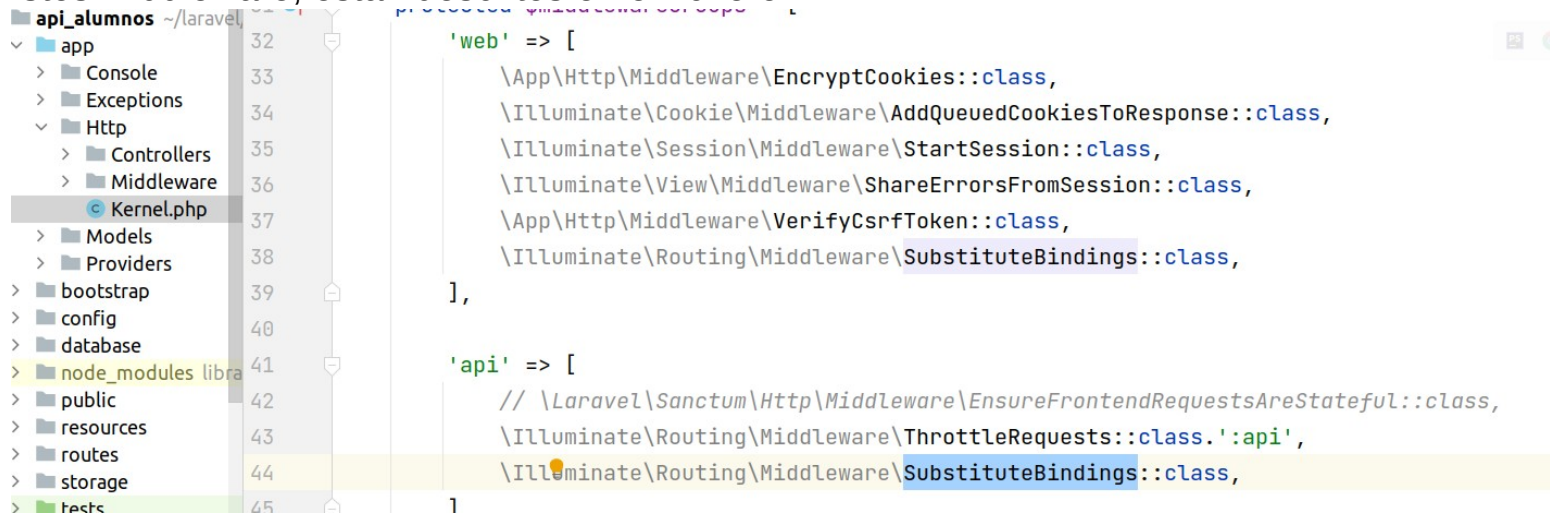
Creamos un nuevo proyecto en Laravel

```
laravel new api_alumnos --git
```

Al crear un api, vamos a establecer las rutas en **api.php**, en lugar de **web.php**

Se diferencian en los middleware que se aplican: en un caso los **web**, y en el otro los **api**.

Estos middleware, están descritos en el fichero



API: Realización

throttle es un middleware que se utiliza para limitar el número de solicitudes.

SubstituteBindings se utiliza para cargar automáticamente los modelos basados en los parámetros de ruta. Gracias a este middleware, cuando un método recibe un **\$id** de un modelo realiza el binding y obtiene el modelo cuyo id coincida con ese entero.

En el caso de **throttle** (limitar o restringir), por ejemplo podríamos establecer `ui'api' => [200 peticiones por minuto con la siguiente configuración:`

```
// \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,  
\Illuminate\Routing\Middleware\ThrottleRequests::class.':200,1',  
\Illuminate\Routing\Middleware\SubstituteBindings::class,  
],
```

Por parte del middleware **SubstituteBindings** no se requiere ninguna configuración añadida.

API: Realización

En las API no necesitamos sesiones, cookies ni compartir errores entre ficheros (son los middleware que aparecen en las rutas de grupo de **web**).

Al hacer un **route:list**, las rutas que aparecen son las de **web.php** y también las de **api.php** que no habíamos visto. Las de **api** tienen prefijo, es por que se establece en el **routes/servicesproviders.php**

```
api_alumnos git:(main) x php artisan route:list --except-vendor

GET|HEAD      / .....
GET|HEAD      api/user .....
```

Ahora vamos a crear un **modelo** para construir nuestra api. Para realizar el ejercicio vamos a poblar la tabla correspondiente al **modelo**, por lo que necesitaremos la **migración** y el **factory** y también un controlador con todos los métodos de gestión **rest** que ya hemos visto

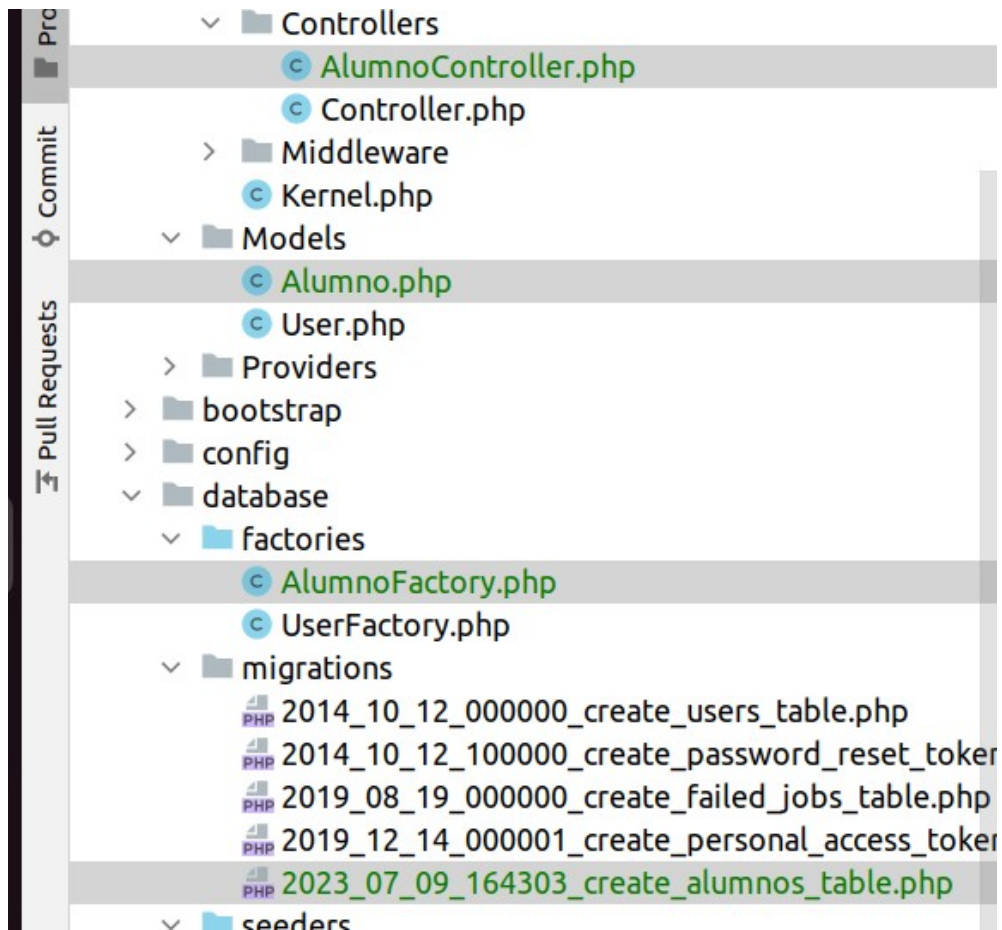
```
cd api_alumnos
```

```
php artisan make:model Alumno -mf --api
```

02 API: Realización

Vemos cómo nos ha creado 4 Clases:

- Controlador **AlumnoController**
- Modelo **Alumno**
- Factoría **AlumnoFactory**
- Migración **xxx_create_alumnos_table**



API: RestFull Json:API

- Si queremos que la representación de cada recurso siga la especificación vista, vamos a crear un resource para adaptarlo
- Con estas clases vamos a poder adaptar lo que enviamos al cliente

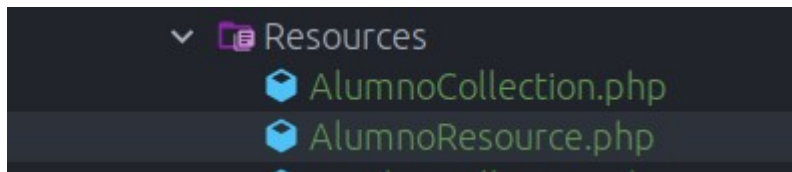
```
php artisan make:Model Alumno
```

```
php artisan make:Controller AlumnoController --api
```

```
php artisan make:request AlumnoFormRequest
```

```
php artisan make:resource AlumnoResource
```

```
php artisan make:resource AlumnoCollection -collection
```



API: El controlador

Un controlador habitual para gestionar una tabla tiene los siguientes métodos, asignados a sus correspondiente solicitudes o entradas:

REST API		
VERBO HTTP	PATH	NOMBRE
GET	/books	index
GET	/books/{id}	show
GET	/books/create	create
POST	/books	store
GET	/books/{id}/edit	edit
PATCH	/books/{id}	update
DELETE	/books/{id}	delete

Para crear las rutas (una vez que existen los métodos del controlador), añadíamos en el fichero **api.php**

```
Route::Resource("alumnos", AlumnoController::class);
```

03 API: El controlador

De todas ellas, como solo estamos creando un api, que es para consultar, no realizaremos la acción de **editar** para modificar, ni de **crear** por lo que las rutas nos quedarían

Para crear estas rutas (los métodos ya existen pues se han creado con la opción **-api** al crear el modelo), añadimos en el fichero **api.php**

VERBO HTTP	PATH	NOMBRE
GET	/books	index
GET	/books/{id}	show
POST	/books	store
PATCH	/books/{id}	update
DELETE	/books/{id}	delete

```
Route::apiResource('alumnos', \App\Http\Controllers\AlumnoController::class);
```

API: El controlador

Ahora podemos comprobar las rutas creadas (con la opción `--path` conseguimos seleccionar las rutas que nos interesan).

```
php artisan route:list --path='api/alumnos'
```

```
→ api_alumnos git:(main) x php artisan route:list --path='api/alumnos'
```

```
GET|HEAD      api/alumnos .....
POST          api/alumnos .....
GET|HEAD      api/alumnos/{alumno} .....
PUT|PATCH    api/alumnos/{alumno} .....
DELETE        api/alumnos/{alumno} .....
```

API: Implementando los métodos y probando Postman

Lo primero creamos la tabla a partir de la migración.

Añadimos 3 campos en la migración para el ejemplo (nombre, password y email)

```
public function up(): void{
    Schema::create('alumnos', function (Blueprint $table) {
        $table->id();
        $table->string('nombre');
        $table->string('password');
        $table->string('email');
        $table->timestamps();
    });
}
```

Creamos registros en el factory

```
public function definition(): array
{
    return [
        'nombre' => $this->faker->name(),
        'password' => bcrypt('12345678'),
        'email' => $this->faker->email(),
    ];
}
```

Invocamos a la población en el seeder (Lo hacemos directamente en **DatabaseSeeder**)

```
public function run(): void{
    \App\Models\Alumno::factory(10)->create();
}
```

API: Creamos la base de datos y la poblamos

Configuramos nuestro fichero `.env`

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=api_alumnos
DB_USERNAME=root
DB_PASSWORD=
```

```
php artisan migrate:fresh --seed
```

`SELECT * FROM `alumnos``

☐ Perfilando [[Editar en línea](#)] [[Editar](#)] [[Explicar SQL](#)] [[Crear código PHP](#)] [[Actualizar](#)]

☐ Mostrar todo | Número de filas: | Filtrar filas: | Ordenar según la clave:

Opciones extra

				Id	nombre	password	email	created_at	updated_at
<input type="checkbox"/>				1	Deron Rippin	\$2y\$10\$026at5PxRJNZbYToe46MxupVzV03c/Z10fqYP2xbkV...	mac86@cummerata.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				2	Brent Haley	\$2y\$10\$VSDU5XikQsZ0065zzDnDp.ykWuRD2NW.YmQwVvFEoK...	anthony27@yahoo.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				3	Abraham Bergstrom	\$2y\$10\$/YzTkz3rwpYIHtGZSYu3eMGontCaEjsKTbjuj7Y2tf...	fbeahan@hotmail.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				4	Miss Darlene Kreiger DVM	\$2y\$10\$AtIsYr59Q0TZ3u/V4OEN2o0TaOI9Sn70riL5TfgE7Gg1...	mebert@hotmail.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				5	Florine Rice	\$2y\$10\$qXK8BMzePjk843kyRB28CuKHSaj5PTq.wWEAJANrNl...	wstrosin@gmail.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				6	Gudrun Gusikowski	\$2y\$10\$HarOqY1TNZJLQCg5i965uoroYcfwxcin43E/o7VdKx...	zswaniawski@hayes.info	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				7	Lexus Miller DDS	\$2y\$10\$q7lwZl.9FeqzVXpfb1JluCZo.x.FLWsEPSjovGRic...	ludwig.davis@gmail.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				8	Harley Armstrong	\$2y\$10\$6uEXw0jhtl6FJL/KOyKquCeF2QGN9tZVaJpSwAt3o...	jedediah59@gmail.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				9	Anabelle Turner	\$2y\$10\$gAepm02QZ5olidGnLawSHuPnaIZNQg2a7m7jkt2dTv...	feest.eliezer@huelts.com	2023-07-09 17:49:48	2023-07-09 17:49:48
<input type="checkbox"/>				10	Holden Frami DVM	\$2y\$10\$GGvEDm8ctfGBXp4ZRAM3duCNSm9SdNbtHkUoP5s9RH...	hermina39@hotmail.com	2023-07-09 17:49:48	2023-07-09 17:49:48

API: Implementando los métodos y probando Postman

Ahora vamos a construir el API, pero no lo vamos a consumir con una aplicación, si no que lo vamos a dejar disponible para que otros lo consuman.

Para probarlo lo vamos a consumir usando una aplicación llamada POSTMAN (<https://www.postman.com/>).

Mejor la instalamos en nuestro equipo y probamos a utilizarla.

Esta aplicación nos va a permitir hacer solicitudes usando la especificación REST contra un servidor.

La instalación la tenemos tanto para **windows** como para linux. (En linux se puede instalar a partir del repositorio usando **snap** (si no lo tenemos instalado lo instalamos)

```
sudo apt install snapd  
  
sudo apt update  
  
sudo snap install postman
```

browser experience,

<https://www.postman.com/downloads/>

The Postman app

Download the app to get started with the Postman API Platform.

Linux (x64)

Linux (arm64)

By downloading and using Postman, I agree to the [Privacy Policy](#) and [Terms](#).

[Release Notes](#) · [Product Roadmap](#)

Not your OS? Download for Windows (x64) or Mac ([Intel Chip](#), [Apple Chip](#))

API: Métodos GET

`http://localhost:8000/api/alumnos => obtener todos los alumnos`

`http://localhost:8000/api/alumnoss/1 => obtener el alumno de id 1`

Para ello escribimos el código de index y show con las respectivas rutas.

Empezamos con el método **index** queremos que retorne un json con todos los datos.

```
public function index()
{
    $alumnos= Alumno::all();
    return response()->json($alumnos);
}
```

Si retornamos

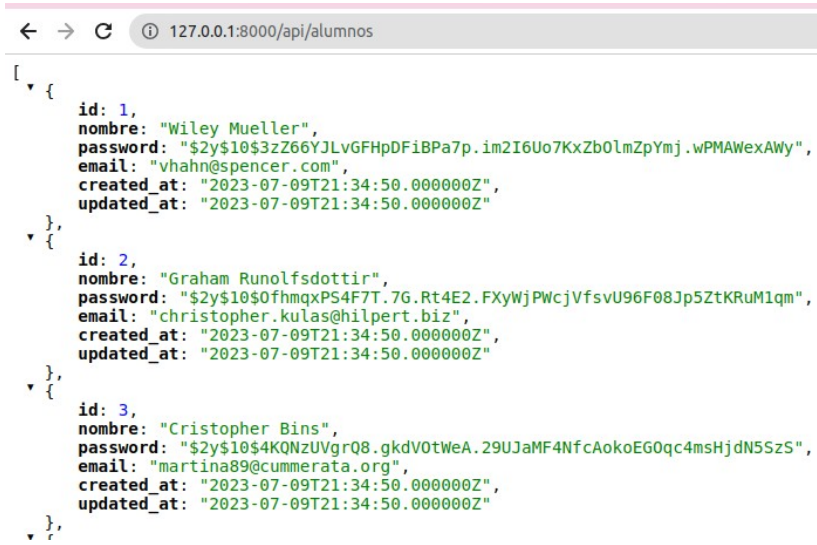
Como lo que nos retorna es un json, podemos instalar algún plugin para verlo de forma mas legible en el navegador, por ejemplo [jsonview](#)

API: Métodos GET

Los modelos de laravel, implementan, entre otras, la interfaz **JsonSerializable**, esto hace que Laravel si detecta que retorna una colección, la serializa automáticamente sin necesidad de hacerlo de forma explícita. Por lo que directamente podríamos hacer:

```
public function index(){  
    return Alumno::all();  
}
```

Como lo que nos retorna es un json, podemos instalar algún plugin para verlo de forma más legible en el navegador, por ejemplo [jsonview](#)



API: Adaptando el json al formato Json:API spec

Pero queremos que nos retorne un json del tipo estándar

Para ello reescribimos el método **toArray** del resource (***AlumnoResource***)

```
public function toArray(Request $request): array
{
    return [
        "id" => (string)$this->id,
        "type" => "Alumnos",
        "attributes" => [
            "id" => $this->id,
            "nombre" => $this->name,
            "email" => $this->email
        ],
        'links' => [
            'self' => url('api/alumnos/' . $this->id)
        ]
    ];
}
```

API: Adaptando el Json al formato Json:API spec

Ahora el index, llamará a Collection que retornará la colección (collection), pero de forma implícita cada elemento de la colección será adaptado a un recurso.

```
// AlumnoController
public function index()
{
    $alumnos = Alumno::all();
    return new AlumnoCollection($alumnos);
    //
}
```

```
// AlumnoCollection
public function toArray(Request $request): array
{
    return ["data"=>$this->collection];
}
```

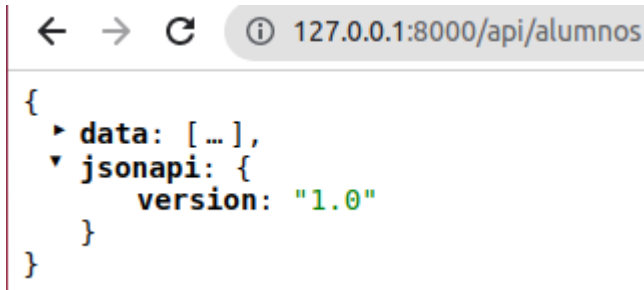
```
{
  "data": [
    {
      "id": "1",
      "type": "Alumnos",
      "attributes": {
        "id": 1,
        "nombre": null,
        "email": "liza.crona@yahoo.com"
      },
      "links": {
        "self": "http://127.0.0.1:8000/api/alumnos/1"
      }
    },
    {
      "id": "2",
      "type": "Alumnos",
      "attributes": {
        "id": 2,
        "nombre": null,
        "email": "emmy27@haag.com"
      },
      "links": {
        "self": "http://127.0.0.1:8000/api/alumnos/2"
      }
    }
  ]
}
```

API: Completando el json

Ahora queremos que agregue algún campo más al JSON, por ejemplo **jsonapi** para especificar la versión

Para ello podemos sobrescribir el método **with** tanto en el resource (lo añadirá a cada elemento del resource, como en el **collection** que lo añadirá al json que retorna.

```
//En AlumnoCollection, añade a nivel general  
public function with(Request $request)  
{  
    return ["jsonapi" =>  
        ["version" => "1.0"]];  
}
```



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000/api/alumnos". The main content area shows a JSON response:

```
{  
  ▶ data: [...],  
  ▼ jsonapi: {  
    version: "1.0"  
  }  
}
```

Esta sería la manera de incluir elementos o campos al json según cosideráramos

API: Errores en caso de get

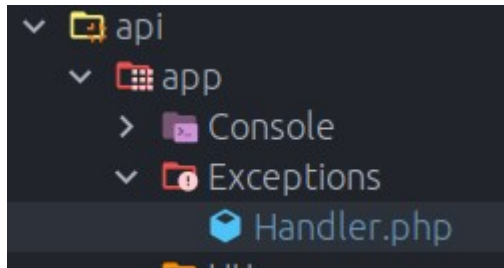
<i>Request</i>	GET, POST, PATCH, DELETE	Accept: application/vnd.api +json	406 Not Acceptable
<i>Request</i>	GET	Error interno, p.e. base de datos	500 Server Error

Los errores los gestionamos en la **clase Handler de la carpeta Exception**.
Sobreescribimos el método render

API: Errores en caso de get: error de servidor

Sobreescribimos el método **render** de la clase **Header**

```
public function render($request, Throwable $exception)
{
    // Errores de base de datos
    if ($exception instanceof QueryException) {
        return response()->json([
            'errors' => [
                [
                    'status' => '500',
                    'title' => 'Database Error',
                    'detail' => 'Error procesando la respuesta. Inténtelo más tarde.'
                ]
            ], 500);
    }
    // Delegar a la implementación predeterminada para otras excepciones no manejadas
    return parent::render($request, $exception);
}
```

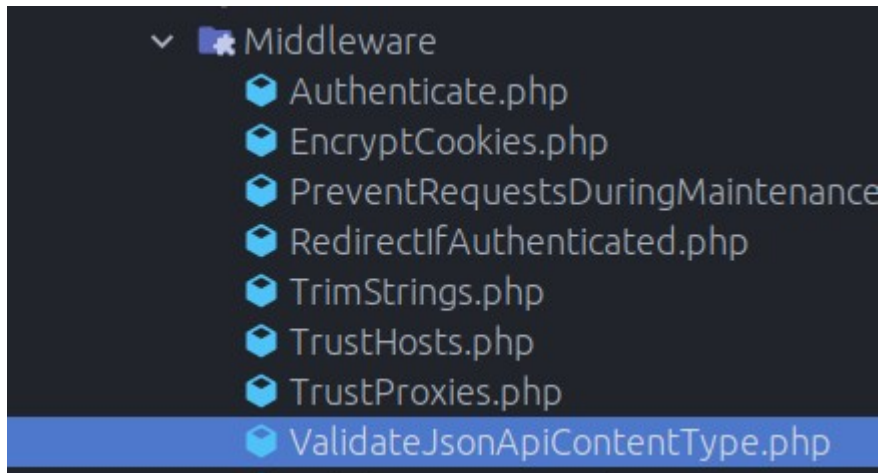


API: Errores en caso de header en solicitud get

Para validar que el encabezado Content-Type hay que escribir un middleware (un software que se va a ejecutar entre el request y el response).

Creamos un middleware, por ejemplo ValidateJsonApiContentType

```
php artisan make:middleware ValidateJsonApiContentType
```



API: Errores en caso de header en solicitud get

Escribimos el contenido en el método **header**

```
public function handle($request, Closure $next)
{
    if ($request->header('accept') != 'application/vnd.api+json') {
        return response()->json([
            'error' => 'Not acceptable',
            'status' => 406
        ], 406);
    }

    return $next($request);
}
```

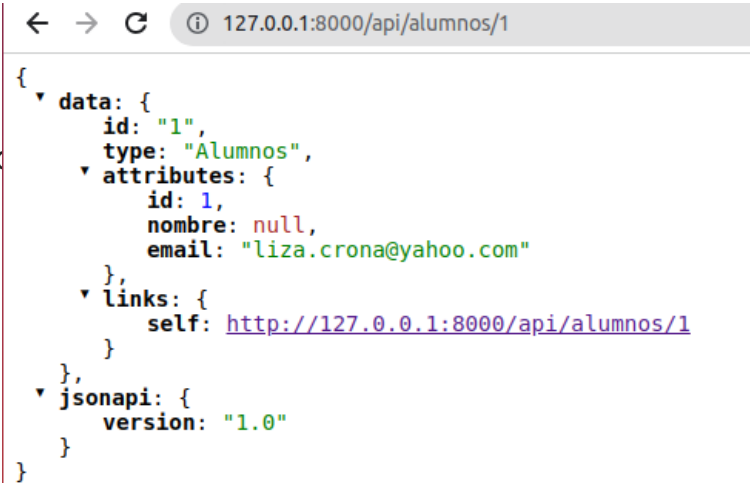
Asociamos el middleware a la ruta a través del **kernel.php**

```
'api' => [
    \App\Http\Middleware\ValidateJsonApiContentType::class|
    // \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class
]
```

05 API: Métodos show

Igualmente el método show visualizará un registro concreto

```
public function show(Alumno $alumno)
{
    Return new AlumnoResource($alumno);
    //
}
```



The screenshot shows a web browser with the address bar displaying "127.0.0.1:8000/api/alumnos/1". The page content is a JSON response from an API. The JSON structure is as follows:

```
{
  "data": {
    "id": "1",
    "type": "Alumnos",
    "attributes": {
      "id": 1,
      "nombre": null,
      "email": "liza.crona@yahoo.com"
    },
    "links": {
      "self": "http://127.0.0.1:8000/api/alumnos/1"
    }
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

Aquí se aprecia el middleware **SubstituteBindings**

Aquí se aprecia el middleware SubstituteBindings. Como ya sabemos, **\$alumno** es el valor del *parámetro en la ruta*. En el caso anterior, en realidad, es el valor **1** (<http://localhost:8000/api/alumnos/1>).

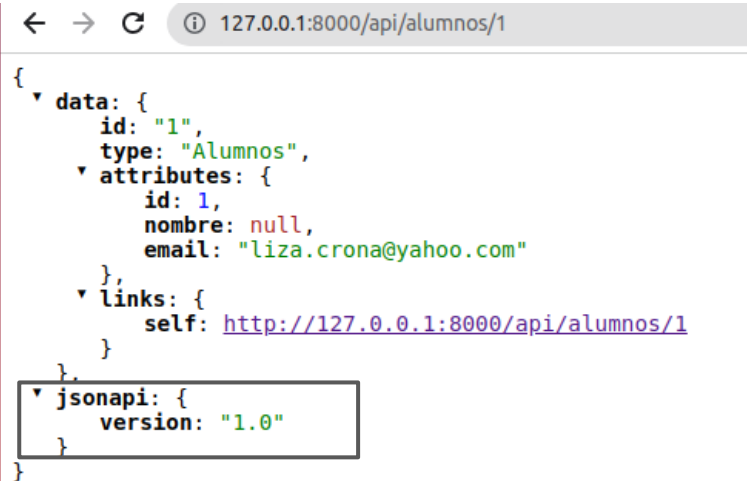
Laravel, al realizar un envoltorio (wrapper) de ese valor al modelo **Alumno**, busca el alumno cuyo ID sea ese valor y **\$alumno** se convierte en un objeto de tipo **Alumno** (un modelo) con el contenido del alumno cuyo ID es 2.

05 API: Métodos show

Para que aparezcan elementos en el json a nivel individual (en este caso **jsonapi**), igualmente hay que sobrescribir el método **with** de la clase

AlumnoResource

```
public function with(Request $request): array
{
    return ["jsonapi" => ["version" => "1.0"]];
}
```



The screenshot shows a web browser at the URL `127.0.0.1:8000/api/alumnos/1`. The response is a JSON object with the following structure:

```
{
  "data": {
    "id": "1",
    "type": "Alumnos",
    "attributes": {
      "id": 1,
      "nombre": null,
      "email": "liza.crona@yahoo.com"
    },
    "links": {
      "self": "http://127.0.0.1:8000/api/alumnos/1"
    }
  },
  "jsonapi": {
    "version": "1.0"
  }
}
```

The `jsonapi` object is highlighted with a red box, indicating the custom header information added by the `with` method.

Al ser un verbo GET, Todo el tema de posibles errores de cabecera o de servidor caído, están ya consideradas contemplado con

API: Métodos show y si no existe

En la especificación JSON:API, si un recurso no se encuentra en una solicitud GET, lo más adecuado es devolver una respuesta con un código de estado HTTP 404 (Not Found).

```
public function show( int $id)
{
    $resource = Alumno::find($id);

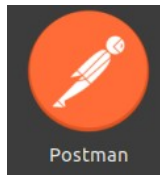
    if (!$resource) {
        return response()->json([
            'errors' => [
                [
                    'status' => '404',
                    'title' => 'Resource Not Found',
                    'detail' => 'The requested resource does not exist or could not be found.'
                ]
            ]
        ], 404);
    }
    return new AlumnoResource($resource);
}
```

API: Métodos con verbos POST DELETE y PUT: postman

Los verbos **get** (*usados en los métodos index y show*), son fáciles de probar en el navegador. Vamos a ver qué ocurre con los métodos que implementan solicitudes realizadas con otros verbos.

Para probar estas solicitudes, como no podemos hacerlo directamente desde el navegador, y **no queremos implementar un cliente**, vamos a realizarlo con la aplicación **postman**

Abrimos la aplicación instalada en nuestro equipo.



Para realizar la prueba, no necesitamos ninguna configuración especial. Lo único que necesitamos es conocer la manera de especificar **la solicitud**: *el modo o verbo y la uri* y **cómo ver los resultados**

API: Métodos con verbos POST

En este caso queremos añadir un recurso en nuestra base de datos

Los posibles errores a considerar serán

- Que algún **campo no cumpla las condiciones exigidas por la base de datos** (requerido, longitud establecida.
- En las cabeceras en el verbo post esperamos recibir

Accept	application/vnd.api+json	406 Not Acceptable
Content-Type	application/vnd.api+json	415 Unsupported Media Type

API: Validando el header para post: ContentType

Añadiremos esta verificación en nuestro middleware

```
if ($request->isMethod('POST') || $request->isMethod('PATCH')) {  
    if ($request->header('content-type') !== "application/vnd.api+json") {  
        // throw new \HttpException(415);  
        return response()->json([  
            'error' => 'Unsupported Media Type',  
            'status' => 415  
        ], 415);  
    }  
}
```

API: Validando datos del formulario

Hay diferentes forma de realizarlo, vamos a optar por crear una clase especializada en hacerlo, un **ModeloRequest**. Suele ser típico (no es obligatorio, solo por legibilidad) que haya una clase para validar el formulario en **store y en update**. **Es posible que ya estén creados**, según lo hayamos especificado en el modelo

```
php artisan make:request AlumnoStoreRequest  
php artisan make:request AlumnoUpdateRequest
```

Son clases que extienden de FormRequest y van cubrir un doble objetivo:

- **Autorizar la acción**
 - **Validar los campos del recurso**
-

API: Validando datos del formulario

```
class UserStoreRequest extends FormRequest
{
  public function authorize(): bool
  {
    return true;
  }
  /**
   * Get the validation rules that apply to the request.
   */
  public function rules(): array
  {
    return [
      'data.attributes.nombre' => 'required|string|max:255',
      'data.attributes.email' => 'required|string|email|max:255|unique:users',
      'data.attributes.password' => 'required|string|min:8|confirmed',
    ];
  }
}
```

En este caso el campo password, con la restricción **confirmed** que debe coincidir, llamado **password_confirmation**
Observa que los campos están estructurados en **data:{attributes:{...}}**

API: Validando datos del formulario

Si hay un error en la validación del formulario se va a gestionar una excepción de de validación

ValidationException

Para gestionarla, en la clase **Header** reescribimos el método **invalidJson** (debe devolver un **JsonResponse**)

```
protected function invalidJson($request, ValidationException $exception):JsonResponse
{
    return response()->json([
        'errors' => collect($exception->errors())->map(function ($message, $field) use
($exception) {
            return [
                'status' => '422',
                'title' => 'Validation Error',
                'details' => $message[0],
                'source' => [
                    'pointer' => '/data/attributes/' . $field
                ]
            ];
        })->values()
    ], $exception->status);
}
```

API: Validando datos del formulario

Como hay varias excepciones (de formulario de acceso a base de datos), vamos a gestionarlas todas en el método **render**

```
public function render($request, Throwable $exception)
{
    // Manejo personalizado de ValidationException
    if ($exception instanceof ValidationException) {
        return $this->invalidJson($request, $exception);
    }

    // Manejo personalizado de QueryException (como ejemplo para errores de base de datos)
    if ($exception instanceof QueryException) {
        return response()->json([
            'errors' => [
                //Código anterior de excepción del servidor
            ], 500);
    }

    // Delegar a la implementación predeterminada para otras excepciones no manejadas
    return parent::render($request, $exception);
}
```

05 API: El controlador

Observa que le pasamos un **StoreAlumnoRequest** en lugar de un **Request**

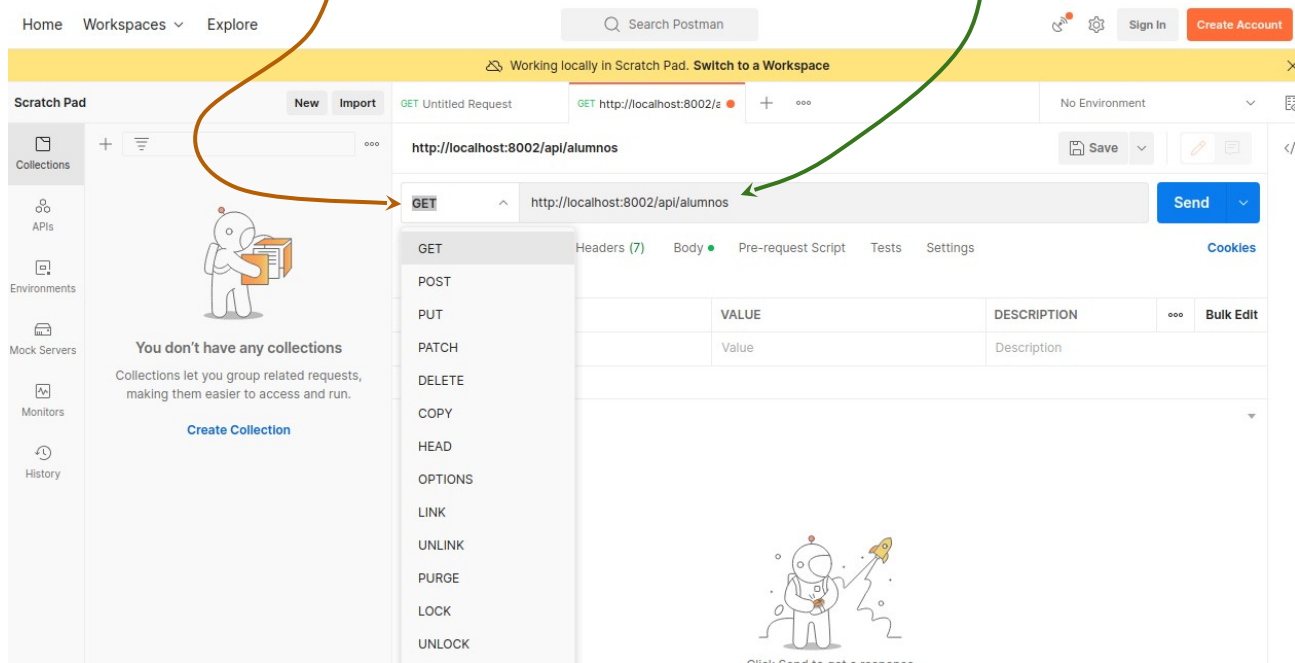
```
public function update(StoreAlumnoRequest $request, Alumno $alumno)
{
    $alumno->update($request->input("data.attributes"));
    info ("Alumno:". $alumno);
    $alumno->save();

    return new AlumnoResource($alumno);
}
```



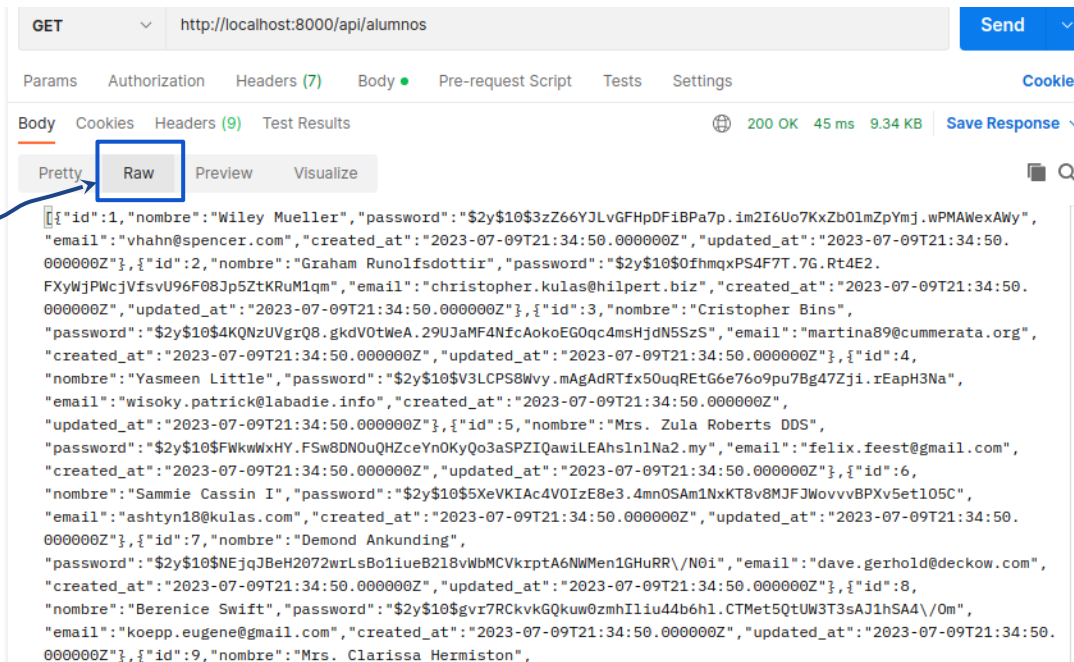
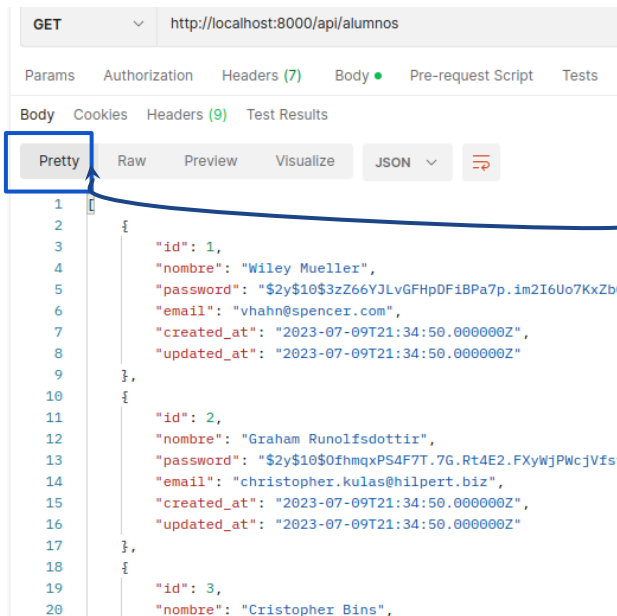
Desplegable con los verbos **http**

URL de la solicitud





Probamos con los métodos ya implementados. Especificamos el verbo get, la solicitud <http://localhost:8000/api/alumnos/get>. La salida en formato json; podemos modificar el formato: **Pretty vs Raw**





Y una solicitud de un alumno solo (cuyo **id** sea 2)

GET ⌵ http://localhost:8000/api/alumnos/2

Params Authorization Headers (7) Body • Pre-request Script Tests Settings

Body Cookies Headers (9) Test Results

200 OK

Pretty

```
1 {
2   "id": 2,
3   "nombre": "Graham Runolfsson",
4   "password": "$2y$10$fhmqxPS4F7T.7G.Rt4E2.FXyWjPwcjVfsvU96F08Jp5ZtKRuM1qm",
5   "email": "christopher.kulas@hilpert.biz",
6   "created_at": "2023-07-09T21:34:50.000000Z",
7   "updated_at": "2023-07-09T21:34:50.000000Z"
8 }
```

GET ⌵ http://localhost:8000/api/alumnos/2

Send

Params Authorization Headers (7) Body • Pre-request Script Tests Settings

Body Cookies Headers (9) Test Results

200 OK 24 ms 522 B Save Responses

Pretty

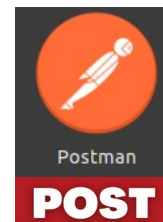
Raw

Preview

Visualize

```
{
  "id":2,"nombre":"Graham Runolfsson","password":"$2y$10$fhmqxPS4F7T.7G.Rt4E2.FXyWjPwcjVfsvU96F08Jp5ZtKRuM1qm","email":"christopher.kulas@hilpert.biz","created_at":"2023-07-09T21:34:50.000000Z","updated_at":"2023-07-09T21:34:50.000000Z"
}
```

06 Postman : POST



Para este tipo de solicitud debemos aportar valores como si se enviara un formulario. Esto se puede aportar en la opción **body** del menú dentro de **form-data** aportamos valores pareja **key-value** como podemos ver en la imagen.

http://localhost:8000/api/alumnos

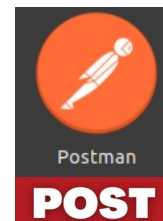
POST http://localhost:8000/api/alumnos

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	KEY	VALUE	DE
<input checked="" type="checkbox"/>	nombre	Manuel	
<input checked="" type="checkbox"/>	password	12345678	
<input checked="" type="checkbox"/>	email	manuel@gmail.com	

06 Postman : POST



Pasando un json al post, debemos especificarlo en la opción raw

POST ▼ http://localhost:8000/api/alumnos/

Params Authorization Headers (11) Body Pre-request Script Tests

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary [JSON](#)

```
1 {  
2   "data": {  
3     "attributes": {  
4       "nombre": "Nombre del Alumno",  
5       "email": "email@dominio.com",  
6       "password": "12345678",  
7       "password_confirmation": "12345678"  
8     }  
9   }  
10 }
```

06 Postman : POST



Ahora escribimos el código en el **controlador**.
En este caso debemos implementar el método **store**

```
public function store(Request $request)
{
    $alumno = new Alumno($request->input());
    $alumno->password = bcrypt($alumno->password);
    $alumno->save();
    //
}
```

En este caso ciframos el password
antes de insertarlo en la bd (**bcrypt**).

Podríamos retornar un mensaje, pero no
estaríamos respetando el estándar de REST

```
public function store(Request $request)
{
    . . .
    return ("{$alumno->nombre se ha creado");
}
```

POST http://localhost:8000/api/alumnos

Params Authorization Headers (9) **Body** Pre-request Script Test

none form-data x-www-form-urlencoded raw binary

	KEY	VALUE
<input checked="" type="checkbox"/>	nombre	Manuel
<input checked="" type="checkbox"/>	password	12345678
<input checked="" type="checkbox"/>	email	manuel@gmail.com

Body Cookies Headers (9) Test Results

Pretty Raw Preview Visualize HTML

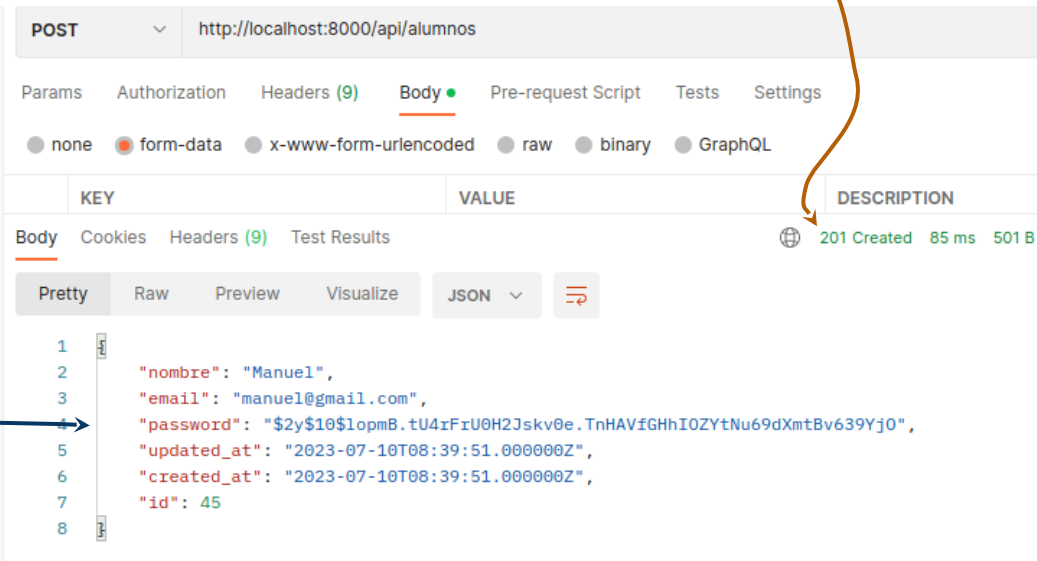
1 Manuel se ha creado

06 Postman : POST

En la especificación REST se recomienda se recomienda devolver el recurso completo, normalmente en formato JSON, mejor con un código 201 (recurso creado)

Ver los códigos (<https://developer.mozilla.org/es/docs/Web/HTTP/Status>)

```
public function store(Request $request) {  
    $alumno = new Alumno($request->input(  
        $alumno->password = bcrypt($alumno->password);  
        $alumno->save();  
        return response()->json($alumno, 201);  
    })  
}
```



POST http://localhost:8000/api/alumnos

Params Authorization Headers (9) Body Pre-request Script Tests Settings

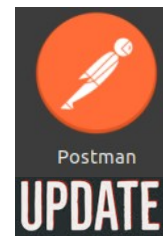
none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION
Body	<pre>{ "nombre": "Manuel", "email": "manuel@gmail.com", "password": "\$2y\$10\$lopmb.tU4rFrU0H2Jskv0e.TnHAVfGHhIOZYtNu69dXmtBv639Yj0", "updated_at": "2023-07-10T08:39:51.000000Z", "created_at": "2023-07-10T08:39:51.000000Z", "id": 45 }</pre>	201 Created 85 ms 501 B

Pretty Raw Preview Visualize JSON



06 Postman : update



Ahora debemos de modificar un alumno existente. Debemos de pasar el alumno que queremos modificar y el o los nuevos datos.

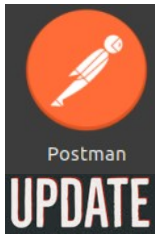
El caso es muy parecido a un store, pero cambiará la forma de realizar la llamada.

En nuestro cliente **postman** para los métodos PUT (actualización completa) o PATCH (actualización parcial), se han de utilizar los parámetros en la opción correspondiente:

A screenshot of the Postman web interface showing a PATCH request configuration. The URL is http://localhost:8000/apl/alumnos/1. The 'Body' tab is selected, and the 'x-www-form-urlencoded' radio button is chosen. A table below lists the request body parameters: 'nombre' with value 'Manuel23', 'password', and 'email' with value 'a@a.com'. Each parameter has a checked checkbox in the first column. A footer row shows 'Key', 'Value', and 'Description' labels.

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	nombre	Manuel23	
<input checked="" type="checkbox"/>	password		
<input checked="" type="checkbox"/>	email	a@a.com	
	Key	Value	Description

06 Postman : update



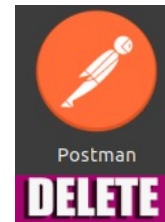
Si queremos hacer una actualización completa (verbo PUT), o una parcial(PATH), podemos actuar de la misma manera: leyendo todos los datos que vienen y con ellos actualizar el recurso

```
public function update(StoreAlumnoRequest $request, Alumno $alumno)
{
    $alumno->update($request->input("data.attributes"));
    $alumno->save();

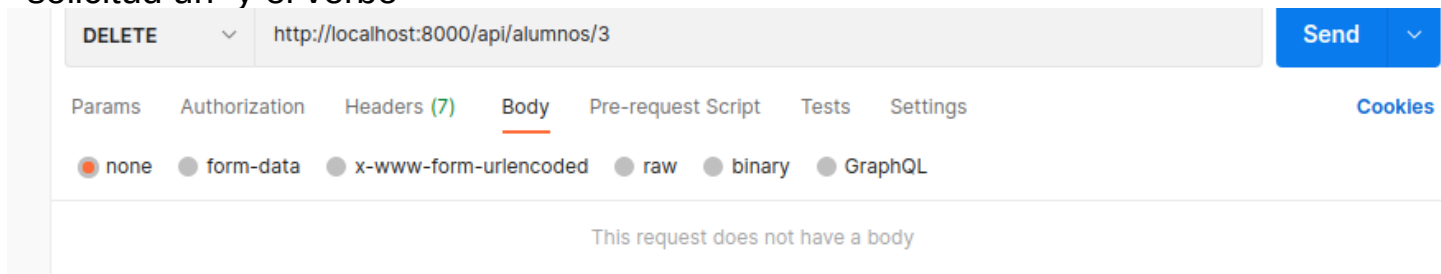
    return new AlumnoResource($alumno);
}
```

Al actualizarlo se va a devolver un código 200 de forma implícita, por lo que no hace falta que lo especifiquemos

06 Postman : delete

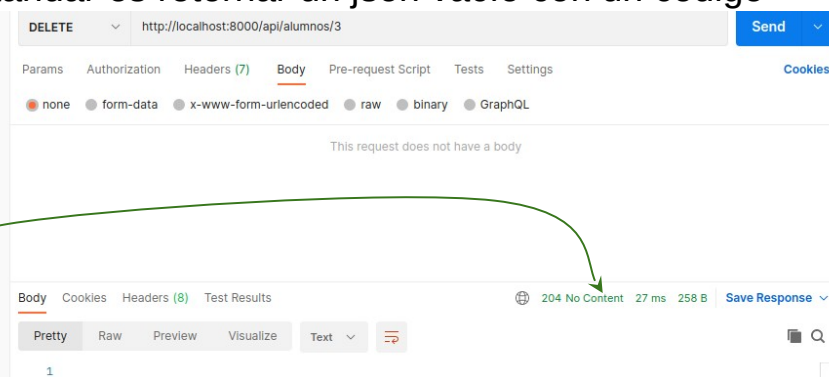


En este caso la acción es inmediata. En postman no hay nada que especificar salvo la solicitud url y el verbo



En el servidor, Realizo la acción y lo más estándar es retornar un json vacío con un código 204 (sin contenido).

```
public function destroy(Alumno $alumno){  
    $alumno->delete();  
    return response()->json(null, 204);  
    // o return response()->noContent();  
}
```



07 Test Unitarios

Realizar test, es **muy importante**.

Muchas veces pensamos que como no los necesitamos para que la aplicación funcione lo dejamos para el final, o simplemente no lo hacemos (para ejemplo yo mismo)

Laravel ya aporta una estructura de test, que podemos ver en la carpeta correspondiente. En ella vemos que tenemos dos tipos de test:

→ Feature:

- ◆ Test de características donde se testean diferentes componentes interactuando entre ellos

→ Unit:

- ◆ Que testean métodos concretos del código

El funcionamiento de los métodos creados es el muy parecido. Para esta api, vamos a realizar los test usando los **FeatureText**

Test Unitarios: ejecutando test

En el terminal (ubicados en el directorio del proyecto) escribimos

```
php artisan test
```

```
public function test_the_application_returns_a_successful_response(): void{  
    $response = $this->get('/');  
    $response->assertStatus(200);  
}
```

```
→ api_alumnos git:(main) x php artisan test  
  
PASS Tests\Unit\ExampleTest  
✓ that true is true  
  
PASS Tests\Feature\ExampleTest  
✓ the application returns a successful response  
  
Tests: 2 passed (2 assertions)  
Duration: 0.12s
```

Por ejemplo si quitáramos la entrada en web.php vemos que no hay assercion

```
FAILED Tests\Feature\ExampleTest > the application  
Expected response status code [200] but received 404.  
Failed asserting that 200 is identical to 404.
```

Test Unitarios: Qué es un test

Un test es un método que nos va a permitir validar otro método

En el método del test, básicamente vamos a hacer 3 acciones:

1. **Invocar al método** que queremos testear pasándole los valores que consideremos
 2. **Comprobar que lo que devuelve** ese método que validamos es lo que esperamos que devuelva
 3. **Informar de ello**
-

Test Unitarios: No son infalibles

Los test no son infalibles, validan lo que les digamos.

Siempre que encontremos un fallo, deberíamos incluirlo en el test para que se valide esa situación y no vuelva a ocurrir.

Debería de haber una cobertura de al menos un 80% de test sobre nuestro código.

Para crear un sección de test, actuaremos de la siguiente manera

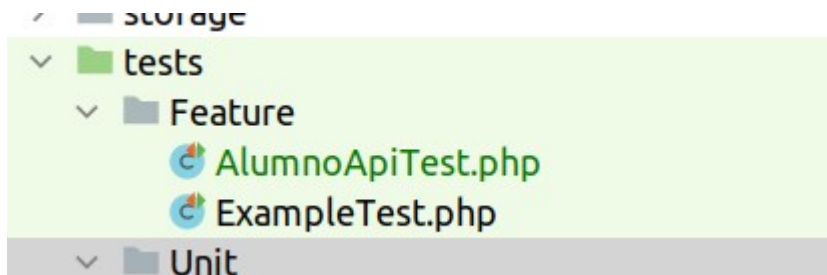
1. Creamos una clase para el test del api.
 2. El método de test, se espera que termine con la palabra Test.
- Para nuestro ejemplo, vamos a crear una clase llamada

AlumnoApiTest

```
php artisan make:test AlumnoApiTest
```

Test Unitarios: No son infalibles

Nos lo habrá creado



Ahora vamos a crear la validación para cada uno de los métodos de nuestro controlador

Todos los métodos que construyamos, tienen que empezar por la palabra reservada **test**, sino, Laravel no lo reconocerá.

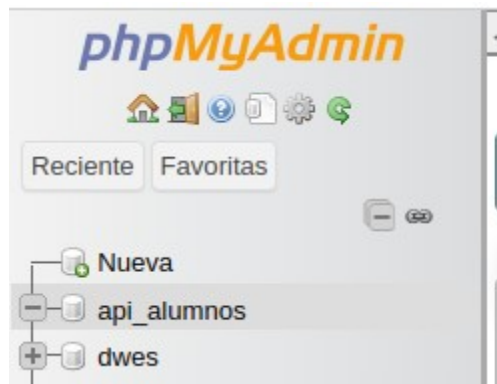
Test Unitarios: Método index

Antes de empezar observamos que, necesitamos tener la base de datos poblada y comprobar que cuando consultamos obtenemos todos los registros

No deberíamos de trabajar con la base de datos original.

Por lo tanto vamos a la clase **(trait) RefreshDatabase** que nos ofrece Laravel.

Creamos una nueva base de datos (también podríamos usar sqlite en memoria) Para configurar la base de datos de test, primero la creamos en nuestro gestor



Test Unitarios: Método index

Ahora accedemos al fichero ubicado en la carpeta raíz del proyecto **phpunit.xml**, y hay una línea comentada, la descomentamos y establecemos el nombre de la base de datos que acabamos de crear

```
. . .  
<php>  
  <env name="APP_ENV" value="testing"/>  
  <env name="BCRYPT_ROUNDS" value="4"/>  
  <env name="CACHE_DRIVER" value="array"/>  
  <!-- <env name="DB_CONNECTION" value="sqlite"/> -->  
  <env name="DB_DATABASE" value="api_alumnos"/>  
  . . .
```

Para poder acceder a la base de datos creada, en lugar de a la original, laravel tiene un traid como hemos comentado antes **RefreshDatabase**, que va a replicar la base de datos original en **api_alumnos** y cada vez que ejecutemos los test parte de la base de datos vacía.

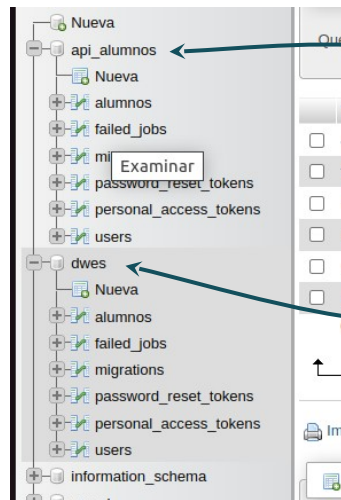
Test Unitarios: Método index

Necesario usar el **trait**

```
class AlumnoApiTest extends TestCase
{
    use RefreshDatabase;

    public function test_can_list_all_Alumnos(){
        $alumnos = Alumno::factory()->count(10)->create();
        dd ($alumnos->count());
    }
}
```

Las creará en la tabla alumnos de la bd api_alumnos



Base de datos para pruebas

Base de datos original

Test Unitarios: Método index

Podemos ver lo que retorno si hago una solicitud **get**

```
public function test_can_list_all_Alumnos(){  
    $alumnos = Alumno::factory()->count(10)->create();  
    $this->get("/api/alumnos")->dump();  
}
```

Ahora quedaría ver de alguna forma que **los alumnos que ha creado**, son los que retorna cuando hago una solicitud **get**.

Para comprobar una igualdad hay que hacer una **assercción**

Ahora es cuando habría que estudiar **qué métodos** tenemos disponibles para realizar esta comparación .

Para este caso vamos a recoger la respuesta y vamos a usar una comparación de valores json

dump() es una herramienta útil para mostrar información de depuración durante las pruebas sin detener la ejecución del código como sí hace **dd**.

Test Unitarios: Método index

```
public function test_can_list_all_Alumnos()  
{  
    $alumnos = Alumno::factory()->count(10)->create();  
    $respuesta = $this->getJson("/api/alumnos");  
    $respuesta->assertJsonFragment([  
        'nombre' => $alumnos[0]->nombre  
    ])  
    ->assertJsonFragment([  
        'nombre' => $alumnos[1]->nombre  
    ]);  
}
```

Si probamos el test, vemos que lo pasa bien.
Podemos probar que index solo devuelva 1 registro o dos, y aquí comprobar 3

```
→ api_alumnos git:(main) ✖ php artisan test  
  
PASS Tests\Unit\ExampleTest  
✓ that true is true  
  
PASS Tests\Feature\AlumnoApiTest  
✓ can list all alumnos  
  
Tests: 2 passed (4 assertions)  
Duration: 0.72s
```

<https://laravel.com/docs/10.x/http-tests#available-assertions>

Test Unitarios: Método show

En este caso queremos verificar que obtenemos un registro

```
public function test_can_get_one_alumno():void{
    $alumno=alumno::factory()->create();
    $response = $this->getJson("/api/alumnos/$alumno->id");
    $response->assertJsonFragment([
        'nombre'=>$alumno->nombre
    ]);
}
```

Test Unitarios: Método store

En este caso queremos verificar que insertamos un registro

```
public function test_can_create_alumno():void{
```

```
    $this->postJson(route('alumnos.store',[  
        'nombre'=>"Pepito de los palotes",  
        'password'=>bcrypt('12345678'),  
        'email'=>'a@a.com'
```

```
    ]))->assertJsonFragment([  
        'nombre'=>"Pepito de los palotes"  
    ]);
```

```
    $this->assertDatabaseHas('alumnos',  
        ['nombre'=>"Pepito de los palotes"]  
    );
```

```
}
```

Realiza una solicitud post de insercción

Espero que me retorne lo mismo

Por otro lado espero que en la base de datos estén estos valores

Test Unitarios: Método update

En este caso queremos verificar que actualizamos un registro

```
public function text_can_update_alumno(){  
    $alumno= alumno::factory()->create();  
    $this->patchJson(route("alumnos.update", $alumno),  
    ["nombre"=>'Nombre updated'])  
    ->assertJsonFragment([  
        'nombre'=>'Nombre updated'  
    ]);  
    $this->assertDatabaseHas('alumnos',  
    ["nombre"=>'Nombre updated']);  
}
```

Creamos un registro

Lo actualizamos con el método patch

Esperamos que lo que nos retorna sea el mismo contenido

Por otro lado espero que en la base de datos estén estos valores

Test Unitarios: Método delete

En este caso creamos un libro, lo borramos y esperamos que nos retorne un 204 o no content

```
public function test_can_delete()
```

```
{
```

```
$alumno = Alumno::factory()->create();
```

```
$this->deleteJson(route("alumnos.destroy", $alumno))
```

```
->assertNoContent();
```

```
$this->assertDatabaseCount('alumnos', 0);
```

```
}
```

```
}
```

Creamos un registro

Lo borramos con el método delete

Esperamos que no retorne nada (204)io

Por otro lado espero que en la base de datos no haya ningún registro