Laravel para servicios web

Introducción

Hasta ahora hemos confiado en el servidor tanto para la lógica como para la vista. El flujo que ha seguido una interacción con la aplicación web desde el cliente ha sido el siguiente:

1. El cliente pide un recurso a través de una URL mediante una petición HTTP
2. El servidor recibe la petición HTTP y hace una de 2 cosas:
    a. Si se trata de una página estática (HTML), la devuelve. Hace lo mismo con cualquier otro recurso estático, como imágenes, estilos, JavaScript…
    b. Si se trata de un recurso dinámico, delega en un intérprete que procesa la petición HTTP y genera una página dinámica, también en formato HTML. Como recordaréis, esto tiene varios pasos intermedios, como la consulta contra una BBDD
3. El cliente (navegador web) recibe el recurso y lo procesa según su extensión. Si se trata de un archivo .zip, lo descargará, pero si se trata de un archivo HTML, lo renderizará para el usuario

Esto es funcional, y hay muchas webs que funcionan así, pero tiene una serie de problemas:

- **Sobrecarga en el servidor**: el servidor, para cada petición web, tiene que generar un documento HTML
    - Generar el documento HTML es costoso en procesador y memoria
    - Enviar el documento HTML es costoso en consumo de ancho de banda
    - Enviar todos los recursos que el cliente solicite tras procesar el HTML aumenta aún más el consumo de ancho de banda. Esto puede mitigarse con la caché que almacenan algunos navegadores de estos recursos, pero puede dar lugar a problemas de compatibilidad si cambia alguno de estos ficheros y se mantiene la versión cacheada
- **Navegación y experiencia de usuario**: para cada interacción que hace el usuario en la web, se produce un "cambio de página". Esto puede ser aceptable cuando cambiamos de un apartado de la web a otro, pero no lo es cuando

interaccionamos con algún elemento dentro del mismo apartado en el que estamos:

- Estoy en el feed de Quacker y me voy al apartado de "Mi perfil". Cambio de contexto = recarga web. Esto es aceptable desde el punto de vista de experiencia de usuario
- Estoy en el feed de Quacker y le doy a "requackear". Sigo en el feed, solamente he pulsado un botón que cambia el estado de un Quack. Cambio del estado de un elemento de la web = recarga web. Esto no es aceptable desde el punto de vista de experiencia de usuario
- En las webs modernas, incluso la percepción de recarga al cambiar de apartado, ya no es algo aceptable desde el punto de vista de experiencia de usuario

Haz la siguiente prueba:

- Accede a Wikipedia, y entra en diferentes artículos. Apreciarás que el icono de "recarga" del navegador ⟳ cambia a ✕ cada vez que cambias de artículo. Esta es la primera evidencia de que el navegador se queda esperando a recibir una recurso desde el servidor para mostrárselo al usuario. Otra evidencia es la siguiente: desde la pestaña "Red" de las **herramientas de desarrollador**, cada vez que cambias de artículo se limpia el historial de peticiones. Por último, pero menos evidente, el DOM no cambia, o cambia muy ligeramente entre cada petición. Esto se puede observar desde la pestaña "Inspector"
- Accede a Gmail y muévete por diferentes apartados, incluyendo los ajustes de la cuenta. El icono de "recarga" del navegador no cambiará. Abre las **herramientas de desarrollador**, ve a la pestaña de "Red" y aprecia cómo las peticiones se van "acumulando" mientras interactúas con la aplicación web. Lo mismo sucederá con el DOM. Si recibes un nuevo email, o si borras alguno, podrás apreciar cómo se modifica sin recargar la página

Esta es la diferencia entre **MPA** (Multi-Page Application) y **SPA** (Single Page Application)

La manera de desarrollar una **SPA** desde el punto de vista del backend es la de hacer que se comporte como una **API**

API

Si el servidor ya no tiene que enviar un documento HTML completo cada vez, ¿qué es lo que envía exactamente? **Envía datos puros.**

Una **API** es, en esencia, un "contrato" o un conjunto de reglas que permite que dos aplicaciones se comuniquen entre sí. En el desarrollo web moderno, la API es el puente que conecta el **Backend** (donde residen tus datos y lógica en Laravel) con el **Frontend** (la interfaz dinámica que el usuario maneja en su navegador).

El cambio de paradigma

Cuando trabajamos con APIs, el rol de nuestro servidor Laravel cambia:

- **Antes (MPA):** el servidor era una "imprenta". Recibía una petición, buscaba los datos, los maquetaba en un HTML y te enviaba la vista terminada.
- **Ahora (API):** el servidor es una base de datos con superpoderes. El cliente (la SPA) le pregunta: *"¿Qué nuevos mensajes hay?"* y el servidor responde simplemente: *"[{"id": 1, "texto": "Hola"}, {"id": 2, "texto": "Adiós"}]"*.

Este intercambio de información se realiza normalmente en formato **JSON** (JavaScript Object Notation), que es ligero, fácil de leer para los humanos y, sobre todo, extremadamente fácil de procesar para el JavaScript que se ejecuta en el navegador del usuario.

Ventajas de este modelo

1. **Separación de responsabilidades:** Laravel se encarga de que los datos sean correctos y estén seguros, mientras que tecnologías como Angular se encargan de decidir cómo se ven esos datos.
2. **Multicliente:** una vez que construyes una API, esa misma API puede servir datos no solo a tu web SPA, sino también a una aplicación móvil (iOS/Android) o a un programa de escritorio. No tienes que programar la lógica dos veces; solo cambias la "cara" que muestra los datos.
3. **Eficiencia:** como solo viaja la información necesaria (el texto del mensaje) y no toda la estructura visual (el menú, el logo, los estilos), el consumo de ancho de banda se reduce drásticamente y la velocidad percibida por el usuario es casi instantánea.

Desventajas de este modelo

Como casi siempre en la vida, no podemos ganar sin sacrificar algo. El modelo SPA no es perfecto, pero las ventajas superan sus desventajas sobradamente. Además, aprenderemos a lidiar con algunas de ellas por el camino:

1. **SEO**: los bots que ejecutan los motores de búsqueda para leer el código de las webs e indexarlas adecuadamente funcionan mucho mejor en una **MPA**, ya que reciben el código completo en HTML. En una **SPA**, deben ser capaces de ejecutar el código cliente para generar un documento HTML que poder inspeccionar. Por eso, SPA no es la mejor alternativa para webs de noticias o blogs, que dependen en gran medida de los motores de búsqueda.
2. **Carga inicial más pesada**: la primera vez que entramos debemos descargar todos los recursos que vamos a necesitar. A partir de esta descarga, el intercambio de datos con el servidor es mucho menor.
3. **Seguridad expuesta**: cualquiera podrá inspeccionar las peticiones que realiza el navegador para averiguar los "puntos de entrada" a nuestros datos. Debemos ser muy cuidadosos en la manera en la que protegemos estos puntos de entrada para evitar scrapping.
4. **Consumo de recursos en el cliente**: el esfuerzo de generar el HTML ya no recae en el servidor. Ese consumo de procesador y RAM que comentamos en la Introducción se traslada al dispositivo del usuario. A día de hoy es un problema menor, ya que la mayoría de estos dispositivos tienen hardware de sobra para poder realizar estas operaciones. Aún así, hay aplicaciones web que ofrecen un "modo de compatibilidad", o "modo HTML clásico" en la que funcionan como MPA, sacrificando experiencia de usuario, pero facilitando el uso de estas a dispositivos más antiguos.

Laravel como motor de servicios web

Instalación e introducción

Vamos a crear nuestra aplicación web con comportamiento de API desde 0. Para ello empezaremos con un nuevo proyecto de Laravel con `None` como kit inicial y, una vez termine su preparación, ejecutaremos lo siguiente

```
php artisan install:api
```

Esto instalará el componente de Laravel con el que nos vamos a pelear en los próximos días: Sanctum. Además, el asistente de instalación, nos pedirá añadir el **trait** "Laravel\Sanctum\HasApiTokens" al modelo **User**

```
class User extends Authenticatable
{
    use HasFactory, Notifiable, HasApiTokens;
...
```

Esto es fundamental para poder realizar la autenticación mediante **tokens**.

Hagamos una regresión: la manera en la que autenticamos a un usuario hasta ahora ha sido:

1. Mostramos un formulario donde se solicitan datos de inicio de sesión: email y contraseña
2. En el servidor comprobamos los datos y, si son correctos, guardamos la información en una sesión
3. En el navegador del cliente se almacena una cookie que permite al servidor "conectar" al cliente con su sesión

Sin embargo, al utilizar Laravel como **backend puro**, esto cambia radicalmente. En el modelo de API, el servidor ya no debería tener que recordar quién eres mediante una sesión almacenada en su memoria. Las APIs se rigen, por norma general, por el principio de ser **stateless**, y esto es fundamental, ya que se reduce drásticamente el consumo de RAM en el servidor.

Para solucionar la autenticación sin usar sesiones, introducimos el concepto de **tokens**. El flujo bajo este nuevo paradigma sería el siguiente:

1. **Petición de Login:** el cliente (la SPA) envía las credenciales (email y password) a un *endpoint* de la API (por ejemplo, `/api/login`).
2. **Validación y Generación:** el servidor comprueba los datos y, si son correctos, genera una **cadena de texto aleatoria y única** llamada **token**.
3. **Respuesta con el "token":** el servidor no guarda nada en una sesión; simplemente le devuelve ese token al cliente en formato JSON.
4. **Almacenamiento en el Cliente:** la SPA guarda esetoken (normalmente en el `localStorage` del navegador o en una variable de estado).

5. **Peticiones Autenticadas:** a partir de ese momento, el cliente debe incluir ese token en la **cabecera** de cada petición HTTP que haga. El servidor recibe el token, comprueba a qué usuario pertenece le da acceso.

La lógica que se ejecuta es, en realidad, la misma. La mayor diferencia está en:

- Crear una sesión VS crear un token.
- Devolver una vista VS devolver un JSON.

Sanctum es un componente de Laravel que nos permite realizar la autenticación mediante tokens de forma sencilla. La instalación de este componente creará varios ficheros nuevos en nuestro proyecto. Entre ellos, uno de los más importantes es `routes/api.php`, en el que crearemos nuestra primera ruta, a partir de ahora **endpoint**.

```
Route::get('/', function() {
    return response()->json([
        'mensaje' => 'Laravel ha vuelto, ¡y en forma de API!'
    ], 200);
});
```

Podemos acceder desde el navegador con `localhost:8000/api/`, o bien crear una petición de tipo GET en nuestro cliente API REST favorito con URL `http://localhost:8000/api/`

Nota: es muy importante que usemos a partir de ahora el `/api/` al final de nuestra URL

Muy importante: antes de seguir, es importante que configures en tu cliente API REST, bien sea para cada petición o bien sea a nivel de proyecto, la cabecera `Accept: application/json`. De esta manera, el servidor tendrá la certeza de que las peticiones que le mandamos esperan recibir datos en formato JSON como respuesta. Por lo general, Laravel podrá intuirlo por la ruta a la que vamos a hacer peticiones (`/api/`), pero es una práctica muy común y necesaria para evitar problemas al trabajar con otros backends en el futuro

Solar Games

Para guiar la explicación, vamos a desarrollar una aplicación web que nos permita publicar reseñas sobre videojuegos. Contamos con la siguiente definición:

1. **Genre**: género (RPG, Metroidvania, Soulslike, Plataformas...)
    - Atributos: nombre (string)
    - Un género tiene muchos juegos (**1:N**)
2. **Game**: juego, el recurso principal
    - Atributos: título (string), desarrollador (string) y año de lanzamiento (integer)
    - Un juego pertenece a un Género (esto no es así en la realidad, pero vamos a mantenerlo sencillo)
    - Un juego tiene muchas reseñas (**1:N**)
3. **Review**: reseña, la opinión de los usuarios
    - Atributos: puntuación (integer, 1 a 10) y comentario (text)
    - Una reseña pertenece a un Juego
    - Una reseña pertenece a un Usuario (el autor)
4. **User**: usuario, proporcionado por Laravel
    - Un usuario tiene muchas reseñas (**1:N**)

Vamos a crear rápidamente las migraciones y hacer los cambios en el modelo:

```
php artisan make:model Genre -mfc
php artisan make:model Game -mfc
php artisan make:model Review -mfc
php artisan make:controller UserController
```

Genre

Migración:

```
$table->string('name')->unique();
```

Modelo:

```
protected $fillable = ['name'];

public function games() {
    return $this->hasMany(Game::class);
}
```

Factoría:

```
return [
    'name' => $this->faker->word(),
];
```

Game

Migración:

```
$table->string('title');
$table->string('developer');
$table->integer('release_year');
$table->foreignIdFor(Genre::class)->constrained()-
>onDelete('cascade');
```

Modelo:

```
protected $fillable = ['title', 'developer', 'release_year',
'genre_id'];

public function genre() {
    return $this->belongsTo(Genre::class);
}

public function reviews() {
    return $this->hasMany(Review::class);
}
```

Factoría:

```
return [
    'title' => $this->faker->sentence(3),
    'developer' => $this->faker->company(),
    'release_year' => $this->faker->year(),
    'genre_id' => Genre::inRandomOrder()->first()->id,
];
```

Review

Migración:

```
$table->text('comment');
$table->integer('score');
$table->foreignIdFor(Game::class)->constrained()-
>onDelete('cascade');
$table->foreignIdFor(User::class)->constrained()-
>onDelete('cascade');
```

Modelo:

```
protected $fillable = ['comment', 'score', 'game_id', 'user_id'];

public function game() {
    return $this->belongsTo(Game::class);
}

public function user() {
    return $this->belongsTo(User::class);
}
```

Factoría:

```
return [
    'comment' => $this->faker->paragraph(),
    'score' => $this->faker->numberBetween(1, 10),
    'game_id' => Game::inRandomOrder()->first()->id,
    'user_id' => User::inRandomOrder()->first()->id,
];
```

User

Modelo:

```
public function reviews() {
    return $this->hasMany(Review::class);
}
```

Generamos datos

Vamos a crear 5 géneros, 20 juegos, 5 usuarios y 50 reviews:

```
Genre::factory(5)->create();
Game::factory(20)->create();
User::factory(5)->create();
Review::factory(50)->create();
```

```
php artisan migrate:refresh --seed
```

Si todo ha ido bien, ya tenemos datos para empezar

Todos los Géneros

A partir de ahora, como has podido suponer, trabajaremos en `routes/api.php`

Vamos a crear un endpoint para obtener todos los géneros almacenados. Para ello, vamos a delegar en el controlador. Recordarás que se trata de un `index`, y que antes hacíamos algo como:

```
public function index() {
    return view ('genres.index', [
        'genres' => Genre::all()
    ]);
}
```

Sin embargo, ahora no tenemos vistas (¡yuju, menos código!) por lo que, de momento, vamos a hacer simplemente:

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

Y configuramos `routes/api.php`

```
Route::get('/genres', [GenreController::class, 'index']);
```

Obtendremos algo así:


```
[
  {
    "id": 1,
    "name": "et",
    "created_at": "2026-01-11T18:04:39.000000Z",
    "updated_at": "2026-01-11T18:04:39.000000Z"
  },
  {
    "id": 2,
    "name": "ipsam",
    "created_at": "2026-01-11T18:04:39.000000Z",
    "updated_at": "2026-01-11T18:04:39.000000Z"
  },
  {
    "id": 3,
    "name": "numquam",
    "created_at": "2026-01-11T18:04:39.000000Z",
    "updated_at": "2026-01-11T18:04:39.000000Z"
  },
  {
    "id": 4,
    "name": "sed",
    "created_at": "2026-01-11T18:04:39.000000Z",
    "updated_at": "2026-01-11T18:04:39.000000Z"
  },
  {
```

```
    "id": 5,
    "name": "consequatur",
    "created_at": "2026-01-11T18:04:39.000000Z",
    "updated_at": "2026-01-11T18:04:39.000000Z"
  }
]
```

Lo cual no está mal, pero es mejorable. Vamos a hacer lo siguiente: vamos a crear los endpoints para obtener todos los Juegos y todas las Reseñas, te lo dejo a ti

Todas las reseñas del usuario autenticado

Empieza lo bueno. Queremos obtener todas las reseñas del usuario autenticado, pero para eso nos tenemos que autenticar... Hemos mencionado anteriormente que **Sanctum** facilita una forma cómoda de hacerlo, y así se puede apreciar en `routes/api.php`. Habrás notado que hay un endpoint `/user`:

```
Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:sanctum');
```

¿Te acuerdas del **middleware** que aprendimos anteriormente para proteger recursos?

Intenta hacer un GET a `/api/user`. Devolverá lo siguiente:

```
{
  "message": "Unauthenticated."
}
```

Junto con un código de estado `401 Unauthorized`. Para poder superar este middleware primero deberemos autenticarnos al más puro estilo API

AuthController

Vamos a crear un nuevo controlador llamado `AuthController` y un request `AuthRequest`. Si recuerdas, los requests se utilizan para indicar restricciones sobre los datos que llegan en las peticiones y los mensajes informativos para el usuario. Lo típico de "es obligatorio rellenar el email" o "la contraseña debe tener más de 8 caracteres, al menos 1 mayúscula, al menos 1 minúscula, al menos un número, al menos un carácter especial, al menos 1 carácter chino y la sangre de al menos un animal mitológico":

```
php artisan make:controller AuthController
php artisan make:request AuthRequest
```

En este request concreto, recuerda que debemos poner el `authorize` a `true`

```
public function rules(): array
{
    return [
        'email' => ['required', 'email'],
        'password' => ['required']
    ];
}

public function messages(): array
{
    return [
        'email.required' => 'El campo de correo electrónico es
obligatorio.',
        'email.email' => 'El correo electrónico debe tener un
formato válido.',
        'password.required' => 'El campo de contraseña es
obligatorio.',
    ];
}
```

A continuación, crearemos un método de `login` en el `AuthController`. Este recibirá un `AuthRequest` y, si los datos son válidos, hará "magia". Recordemos la lógica que debe seguir:

1. Llamamos al método `validated()` de la petición. Este devolverá un array con los datos sujetos a validación (email y contraseña) sólo en caso de que se supere la validación. En caso contrario, devolverá alguno de los mensajes de error definidos en `AuthRequest`. La magia de Laravel hará que estos mensajes, que antes devolvíamos a la vista con el formulario de login, ahora se devolverán en un JSON
2. Tras ello intentaremos la autenticación. Si no la superamos, devolveremos un error. Por supuesto, en JSON
3. Si superamos la autenticación, obtendremos el usuario autenticado con esos datos y, ahora viene lo nuevo, generaremos un **Token**. ¡Por fin! Hablamos de los tokens hace 6 o 7 páginas, todo lo bueno se hace esperar.
4. Muy importante, debemos informar al usuario de que todo ha ido bien y darle su token para que lo guarde. A partir de ahora lo usaremos bastante.

```
public function login(AuthRequest $request) {
    $validado = $request->validated();

    if (!Auth::attempt($validado)) {
        return response()->json(['message' => 'Credenciales
inválidas'], 401);
    }

    $auth = Auth::user();
    $token = $auth->createToken('API Token para ' . $auth->email)-
>plainTextToken;

    return response()->json([
        'user' => $auth,
        'token' => $token
    ], 200);
}
```

Por supuesto, tendremos que crear la ruta correspondiente en `routes/api.php`. Esta es de tipo POST

```
Route::post('/login', [AuthController::class, 'login']);
```

Tras ello, inspeccionamos la BBDD para buscar un email de un usuario con el que iniciar sesión y lo configuramos en el cliente API REST. Dependerá de cada cliente, pero el que estoy usando yo (Bruno) me permite añadir a la petición un cuerpo de tipo "Multipart Form". Si lo hemos hecho todo bien, el POST nos devolverá algo así:


```
{
  "user": {
    "id": 1,
    "name": "Anika McGlynn",
    "email": "wunsch.elna@example.org",
    "email_verified_at": "2026-01-11T18:04:39.000000Z",
    "created_at": "2026-01-11T18:04:40.000000Z",
    "updated_at": "2026-01-11T18:04:40.000000Z"
  },
  "token": "1|OpGbGn1zntOUFgPZTE0mFGdrsmjsjc9LLeiVl44sb46314a9"
}
```


Ese token será el que debamos añadir a nuestras peticiones hacia endpoints protegidos. Lo normal es crear una variable dentro del cliente API REST para almacenar el token de manera cómodamente accesible.

A continuación, podemos modificar el `GET` a `/user` para añadirle el token de autenticación o, si te lo permite tu cliente API REST, te aconsejo que lo hagas a nivel de proyecto. En cualquiera de los 2 casos, debes tener una pestaña que sea **Auth**, y una opción que sea **Bearer Token**. Ahí pegaremos el token. Al hacer esto, ya podemos hacer la petición:


```
{
  "id": 1,
  "name": "Anika McGlynn",
  "email": "wunsch.elna@example.org",
  "email_verified_at": "2026-01-11T18:04:39.000000Z",
  "created_at": "2026-01-11T18:04:40.000000Z",
  "updated_at": "2026-01-11T18:04:40.000000Z"
}
```

Endpoint de reseñas del usuario

Ya hemos iniciado sesión, ahora queremos obtener las reseñas del usuario. Tenemos que tomar 2 decisiones:

1. ¿Dónde ponemos el método para obtener las reseñas que ha hecho un usuario concreto? Porque estamos trabajando tanto con reseñas como con usuarios. En este caso no hay discusión. Si devolvemos reseñas, el método debe estar en el controlador de reseñas: `ReviewController`
2. ¿Qué ruta seguimos para llegar hasta esas reseñas? Sobre esto no hay una única mejor opción, pero sí es común utilizar una URL jerárquica (`/user/reviews`, primero obtenemos el usuario y luego sus reseñas)

Por tanto, en `ReviewController`:

```
public function byAuthUser() {
    $user = Auth::user();
    return $user->reviews;
}
```

Y en `routes/api.php`:

```
Route::get('/user/reviews', [ReviewController::class,
'byAuthUser'])
    ->middleware('auth:sanctum');
```

Muy importante añadir el `middleware('auth:sanctum')` por 2 razones:

1. Para proteger esa ruta
2. Para autenticar al usuario y poder obtener sus reseñas (sino, `Auth::user()` no funcionaría)

Creamos la petición `GET`, y recuerda el token si no lo has configurado a nivel de proyecto

Recursos

Los resources de Laravel nos van a permitir definir cómodamente cómo se estructura el JSON que vamos a devolver a las distintas peticiones HTTP. Hasta ahora, devolvemos instancias y Laravel, por detrás, hace una **serialización** de esas instancias, pero no nos va a ayudar mucho más, y nosotros queremos poder personalizar los datos que enviamos. Podríamos crear el JSON a mano antes de devolverlo y, de hecho, es lo que vamos a hacer, pero nuevamente Laravel nos pone las cosas un poco más fáciles con una clase especial llamada `Resource`

```
php artisan make:resource GameResource
```

Esto creará un fichero `GameResource.php` en la ruta `app/Http/Resources/`. En dicho fichero encontraremos un método `toArray` en el cual definiremos la forma del documento JSON que queremos devolver. Vamos a seguir el estándar *de facto*, definido en {json:api}. Iremos rellenando los siguientes campos:

- `type`: indica el tipo de objeto que estamos devolviendo: genre, game, review...
- `id`: indica cómo se identifica esa instancia de manera única e inequívoca. No tiene por qué ser la PK en BBDD, pero la inmensa mayoría de las veces lo será
- `attributes`: lista de atributos del objeto. Puede incluir atributos calculados, **pero no** atributos que representen claves foráneas
- `relationships`: indica qué relaciones existen entre esta instancia y otras. Aquí pondríamos las claves foráneas y otras relaciones que hayamos definido
- `links`: enlaces relacionados con el recurso. Por ejemplo: un enlace al `show` de esa instancia concreta
- `include`: información adicional relacionada con el recurso principal. Este campo es **opcional** y debemos controlar su inclusión mediante el uso del parámetro `GET include=`. Además, esta información debe estar conectada con la información del recurso principal mediante alguna relación. Por ejemplo: existe una relación entre `Game` y `Review` llamada `reviews`. Podemos devolver, junto con un resource de `Game`, el resource de cada una de sus `Review`.

  Cuando se incluye, este documento se llama "compuesto", y requiere de lógica adicional, ya que las relaciones suelen ser bidireccionales, por lo que **sólo puede haber** 1 resource para cada par `type-id`. Así evitaremos información redundante (e incluso recursiva).
- `meta`: información adicional no relacionada con el recurso principal. Por

ejemplo: copyright, autor, número total de páginas (si el recurso está paginado), etc.

Para guiar este apartado, voy a seguir el ejemplo con `Game`.

Nota: todos los datos que incluyamos en el documento JSON, incluyendo valores numéricos, deberán estar en formato de texto.

`type-id`

La manera de identificar una instancia concreta dentro de nuestra aplicación es el par `type-id`

```
public function toArray(Request $request): array
{
    return [
        'type' => 'game',
        'id' => (string) $this->id,
    ];
}
```

`attributes`

Lista de atributos del objeto. Deberemos seguir el estándar de nombrado `camelCase` para los atributos con nombres compuestos, como `created_at` y `updated_at`:

```
...
'attributes' => [
    'title' => $this->title,
    'synopsis' => $this->synopsis,
    'developer' => $this->developer,
    'publisher' => $this->publisher,
    'releaseYear' => (string) $this->release_year,
    'createdAt' => $this->created_at,
    'updatedAt' => $this->updated_at,
],
```

Apreciarás que no he incluido `genreId`. Esto es porque, pese a que sea un atributo de una instancia de `Game`, deberá estar incluido como relación

`links`

Incluiremos enlaces relacionados con el recurso que estamos describiendo. Como mínimo, siempre incluiremos un enlace a `self` que, en el caso de una instancia, se corresponde con el enlace a `show`. Nos valdremos de función que proporciona Laravel `route()`

```
...
'links' => [
    'self' => route('genres.show', ['genre' => $this->genre->id])
]
```

Para que esto funcione, deberemos tener la ruta nombrada al `show` de `Genre`. Podemos comprobar si la tenemos con el siguiente comando:

```
php artisan route:list --except-vendor
```

Si no aparece ningún `genres.show`, deberemos modificar `web/api.php`. Tenemos varias opciones. Una de ellas es nombrar esa ruta concreta:

```
Route::get('/genres/{genre}', [GenreController::class, 'show'])
    ->name('genres.show');
```

O bien agrupar todas las rutas de `Genre` con `Route::apiResource`:

```
Route::apiResource('genres', GenreController::class);
```

El problema de esto es que las 5 rutas (index, show, store, update y destroy) quedan agrupadas bajo una misma sentencia, por lo que no podemos aplicar un `middleware` de manera granular. Vamos a dejarlo así de momento

`relationships`

Aquí incluiremos los enlaces a recursos relacionados con el principal. Ya hemos visto que `Genre` es uno de ellos, pero también lo es `Review`. Es importante que aquí pongamos el **nombre de la relación**. En el caso de `Game-Review` es obvio, pero imagínate la relación `Article-User`. Muy probablemente la relación se llamaría `author`, no `user`. Imagina el caso `User-Quack`. Tenemos: `quacks`, `requacks` y `quavs`.

Podemos seguir exactamente el mismo nombrado que hayamos elegido en el modelo: `genre` y `reviews`. Cada uno de estos campos incluirá, a su vez, su propio `links` y `data`.

El campo `data` incluirá el par `type-id` del recurso relacionado con el principal

El campo `link` indicará tanto la ruta a la relación como un endpoint que nos permita obtener los datos relacionados con dicho recurso. Por ejemplo:

```
localhost:8000/api/game/1/relationships/genre
localhost:8000/api/game/1/genre
```

La ruta con `relationships` deberá devolver un documento que describa solamente los datos que existen en la relación, que coincide con lo que devolvemos en el campo `relationships` del documento principal, pero sólo para el recurso incluido en la URL. Es más difícil de explicar que lo que es en realidad:

```
...
'relationships' => {
    'genre' => # GENRE_RELATIONSHIPS
}
```

La ruta `localhost:8000/api/game/1/relationships/genre` deberá devolver solamente `# GENRE_RELATIONSHIPS`, mientras que la ruta `localhost:8000/api/game/1/genre` deberá devolver un documento con todos los datos del recurso `Genre`. Podemos, por tanto, delegar la creación de esta parte del documento en otro Resource al que podemos llamar `GameGenreRelationshipResource`. Por tanto, quedaría algo así:

```
...
'relationships' => [
    'genre' => new GameGenreRelationshipResource($this),
    'reviews' => new GameReviewRelationshipResource($this)
]
```

Y, como ejemplo de `GameGenreRelationshipResource`, tendríamos:

```
public function toArray(Request $request): array
{
    return [
        'links' => [
            'self' => route('games.relationships.genre', ['game' =>
$this->id]),
            'related' => route('games.genre', ['genre' => $this-
>id])
        ],
        'data' => [
            'type' => 'genre',
            'id' => (string) $this->genre->id
        ]
    ];
}
```

Por supuesto, tendremos que crear las siguientes rutas nombradas:

```
game/{{game}}/relationships/genre -> games.relationships.genre
game/{{game}}/genre -> games.genre
```

La primera (`game/{{game}}/relationships/genre`) la gestionaría `GameController`, o bien un nuevo controlador sólo para las relaciones `GameRelationshipsController`

La segunda (`game/{{game}}/genre`) la gestionaría `GenreController`

`included`

Al igual que con `relationships`, delegaremos la creación en los resources específicos de los objetos implicados, pero con cuidado. Lo modificaré más adelante:

```
...
'included' => [
    new GenreResource($this->genre),
    ReviewResource::collection($this->reviews)
]
```