LARAVEL Framework - Tutorial 01 - Creación de API RESTful (actualizado)

Sumario

- 1 Introducción a Laravel versión 7.x
- 2 Configuración e instalación de herramientas locales
  - 2.1 Instalación de Apache, MySQL con XAMPP
  - 2.2 Configuración del dominio y host virtual
    - 2.2.1 Configuración de las direcciones IP de los dominios virtuales en el sistema operativo
    - 2.2.2 Configuración de los dominios virtuales en Apache
  - 2.3 Instalación de Composer
  - 2.4 Instalación de Sublime Text3
  - 2.5 Instalación de Git y Github en entorno local
  - 2.6 Instalación de extensión Postman o Advanced REST Client en Google Chrome
  - 2.7 Instalación y configuración de Laravel
    - 2.7.1 Error de usuario/contraseña al instalar con Composer
- 3 Creación de estructura y componentes de una API REST con LARAVEL
  - 3.1 Estructura de LARAVEL
  - 3.2 Creación de modelos
    - 3.2.1 Información referente al modelo que vamos a crear en Laravel
    - 3.2.2 Creación de los modelos en Laravel con PHP Artisan
  - 3.3 Creación de Migrations con Laravel y PHP Artisan
  - 3.4 Creación de Seeders para tablas con Laravel y PHP Artisan
  - 3.5 Creación de Controladores y Rutas API RESTful en Laravel
    - 3.5.1 Creación de los Controladores
      - 3.5.1.1 Generación de las plantillas de Controladores para Fabricante y FabricanteAvion y Avion con PHP Artisan y modificaciones básicas
  - 3.6 Creación de las Rutas de la API RESTful
    - 3.6.1 Rutas por defecto de la Aplicación
    - 3.6.2 Rutas Resource a partir de los Controladores
- 4 Paginación de resultados en Laravel
- 5 Versionado de una API RESFful con Laravel
- 6 Implementación de las operaciones de la API RESTful con Laravel
  - 6.1 Códigos de estado HTTP a utilizar en la API RESTful
  - 6.2 Mostrar los valores de un recurso y de un recurso anidado de la API RESTful
  - 6.3 Creación de recursos en la API RESTful
  - 6.4 Actualización de recursos en la API RESTful
  - 6.5 Borrado de recursos en la API RESTful
- 7 Caché de consultas con Laravel para reducir carga en bases de datos
- 8 Los middlewares en Laravel
  - 8.1 Desactivación de la protección CSRF del middleware de Laravel para la API RESTful
  - 8.2 Autenticación básica con middleware en Laravel
    - 8.2.1 Creación del modelo User.php su migration y el seeder
    - 8.2.2 Configuración de la autenticación
- 9 Control CSRF en la API
- 10 Autenticación de Usuarios en Laravel
  - 10.1 Bloquear el acceso a la API solamente para usuarios registrados
- 11 JSON Web Tokens en Laravel
- 12 API RATE Limits
- 13 Validación de Usuarios con OAuth2
  - 13.1 Introducción a OAuth2
  - 13.2 Instalación de Servidor OAuth2 en Laravel
  - 13.3 Middleware para validar con OAuth2
  - 13.4 Pruebas con OAuth2
- 14 Uso de Git para control de Versiones
- 15 Guía de Diseño de APIs HTTP

Introducción a Laravel versión 7.x

VERSIÓN DEL TUTORIAL ACTUALIZADO A LA VERSIÓN 7.X de LARAVEL

Laravel es uno de los frameworks más populares de código abierto para PHP que nos permitirá desarrollar aplicaciones y servicios web con PHP 7.

Fue creado en el año 2011 y su filosofía es la de desarrollar código PHP de forma elegante y simple para crear código de forma sencilla y permitiendo múltiples funcionalidades.

[Página Oficial de Laravel]

Aprovecha lo mejor de otros frameworks y las últimas características de PHP.

Laravel está formado por múltiples dependencias, en especial de Symfony.

Entre sus **características básicas** podemos citar:

- Sistema de rutas y RESTful.
- Motor de plantillas, Blade.
- Peticiones Fluent.
- ORM Eloquent.
- Basado en composer.
- Soporte de caché.

- Soporte MVC.
- Utiliza componentes de Symfony.

Configuración e instalación de herramientas locales

Veamos como instalar las herramientas necesarias para trabajar con el framework Laravel en un entorno local.

Instalación de Apache, MySQL con XAMPP

Vamos a realizar la instalación para un entorno Windows, y en este caso instalaremos un sistema WAMP que incluye Apache, MySQL y PHP.

- Descargaremos **XAMPP desde su página oficial**. https://www.apachefriends.org/download.html
- Procederemos a la instalación indicando la ruta de instalación. Se recomienda en la carpeta raíz c:\xampp o d:\xampp etc..
- Si se desea se puede ver un video de la instalación aquí: https://manuais.iessanclemente.net/index.php/XAMPP_outro_servidor_WAMP_en_Windows

Configuración del dominio y host virtual

Para nuestra instalación de Laravel vamos a crear un dominio.local. La idea es que cuando tecleemos el nombre de ese dominio.local nos conecte con la ruta dónde está instalada la aplicación que realizaremos con Laravel.

Configuración de las direcciones IP de los dominios virtuales en el sistema operativo

Para ello tendremos que realizar los siguientes pasos:

- Editar el fichero **hosts** de windows situado en C:\Windows\System32\drivers\etc\hosts
- Si no nos deja grabar el fichero entonces copiar el fichero hosts al escritorio, editarlo y luego copiar sobreescribiendo el fichero antiguo en la carpeta anterior.
- El **contenido del fichero hosts** debería ser algo similar a:

```
127.0.0.1   www.dominio.local dominio.local
```

- Si queremos añadir más dominios a la misma dirección IP los pondremos en la misma línea separadas por un espacio.

Una vez configuradas las IP's de los dominios virtuales tendremos que editar el fichero de configuración de XAMPP:

Si queremos configurar nuestro servidor virtual en XAMPP para que siga la misma estructura de clases al estilo **www.dominio.local**, lo primero que tendremos que hacer es configurar la resolución del nombre www.dominio.local para que cuando el navegador quiera conectarse a ese dominio lo haga a nuestro servidor web local en **127.0.0.1**

Para ello en Windows 8.1 tendremos que **editar el fichero hosts**.

1. Accederemos a la carpeta: **C:\Windows\System32\drivers\etc**
2. Copiaremos el fichero **hosts** al **escritorio** y lo editaremos.
3. Añadiremos la siguiente entrada al final del fichero: **127.0.0.1 www.dominio.local**
4. Grabamos el fichero y lo pegaremos de nuevo en la carpeta: **C:\Windows\System32\drivers\etc** sobreescribiendo el actual.

A partir de este momento cuando queramos acceder al www.dominio.local nuestro equipo se conectará a nuestro servidor localhost.

Configuración de los dominios virtuales en Apache

Lo que nos falta por hacer ahora es que cuando escribamos la url http://www.dominio.local el XAMPP nos muestre el contenido de nuestras páginas web.

1. Se recomienda crear una carpeta en **\xampp\htdocs\dominios\dominio.local**.
2. **Copiar** a la carpeta **web** todas nuestras páginas web, **EXACTAMENTE** con la misma estructura que tenemos en el instituto (**contenido de la carpeta web del servidor ispconfig**).
3. De esta forma si accedemos por http://localhost se mostrará el panel de gestión del XAMPP y si accedemos por http://www.dominio.local tendrá que acceder a la carpeta **\xampp\htdocs\dominios\dominio.local** que creamos antes.
4. **La idea es tener exactamente la misma estructura que tendríamos en nuestro servidor www.dominio.local del instituto**.
5. A continuación editaremos el fichero **httpd-vhosts.conf** que se encuentra en la carpeta **\xampp\apache\conf\extra\httpd-vhosts.conf**
6. Añadiremos las siguientes instrucciones al final del fichero adaptando las rutas y letras de unidad a nuestra propia configuración:

```
NameVirtualHost *:80

<VirtualHost *:80>
DocumentRoot "C:\xampp\htdocs"
ServerName localhost
</VirtualHost>

<VirtualHost *:80>
    DocumentRoot "C:\xampp\htdocs\dominios\dominio.local"
    ServerName www.dominio.local
    ServerAlias dominio.local
    <Directory "C:\xampp\htdocs\dominios\dominio.local">
AllowOverride All
Order Allow,deny
Allow from all
Require all granted
    </Directory>
</VirtualHost>
```

7. **Grabaremos el fichero y reiniciaremos el servidor Apache** desde el XAMPP Control.

8. **NOTA IMPORTANTE:** Acordarse de modificar el **Netbeans** o **Sublime Text** para que a la hora de **previsualizar** las páginas las abra en la dirección http://www.dominio.local

Instalación de Composer

Composer es una herramienta para gestionar dependencias en aplicaciones PHP, algo así como un instalador de paquetes/librerías referentes a desarrollo web.

Con **Composer** podemos crear un archivo de configuración (en formato JSON) en el cuál indicamos las dependencias (de otras librerías) que tiene nuestro proyecto y Composer es capaz de descargar e instalar automáticamente dichas librerías en la versión que le indiquemos en el archivo de configuración.

Composer se basa en otro proyecto denominado https://packagist.org/ el cuál es el principal repositorio utilizado por Composer para descargar los paquetes.

El uso de Composer es muy recomendable por que a la hora de crear un proyecto en Laravel, le podemos indicar que se descargue Laravel automáticamente, sin tener que ir nosotros a la página y descargarlo manualmente, así como cualquier otro tipo de librería adicional que necesitemos en cualquier momento.

Para **instalar Composer** tendremos que hacer lo siguiente:

- Ir a su **página oficial**: https://getcomposer.org/
- Pulsar en **Download**: https://getcomposer.org/download/
- Elegir el método adecuado a nuestra versión de sistema operativo.

Una vez instalado Composer para poder ejecutarlo tendremos que ir a la línea de comandos del sistema con **Windows+R** teclear **cmd** y escribir **Composer** seguido de la instrucción correspondiente.

Ejemplo de fichero de **configuración .json para Composer**:

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

Ejemplo de comando de creación de un proyecto Laravel:

```
composer create-project --prefer-dist laravel/laravel Directorio
```

Instalación de Sublime Text3

Ver información de cómo **instalar y configurar Sublime Text3** en:

https://manuais.iessanclemente.net/index.php/Tutorial_sobre_editor_Sublime_Text_3

Instalación de Git y Github en entorno local

Git nos permitirá llevar un **control de versiones** de nuestro proyecto.

Incorpora una serie de instrucciones que nos permiten llevar el control de cambios que se han realizado en los archivos, es decir mantener un registro de todos los cambios realizados en el código de nuestro proyecto, crear ramas distintas en el proyecto, etc..

Github nos permite **sincronizar nuestro repositorio local de cambios con un repositorio remoto en github.com**. A través de este repositorio remoto en Github.com podremos sincronizar nuestros desarrollos locales con un servidor de producción, por lo que no necesitaríamos FTP para transferir las actualizaciones al servidor. Usaríamos Github como repositorio intermedio.

Github nos proporciona una consola que nos da acceso a todos los comandos de Git desde un entorno más amigable, en lugar de utilizar la línea de comandos de Git continuamente para todas las operaciones.

Para instalar GitHub:

- Página oficial de github: https://github.com/
- Página de descarga de cliente Windows: https://github-windows.s3.amazonaws.com/GitHubSetup.exe

La instalación de Github no implica el tener acceso por comandos a Git, pero sin embargo incorpora una utilidad llamada **Gitshell** que nos abrirá una línea de comandos donde sí podremos introducir comandos de Git.

Para usar GitHub: Para poder usar GitHub correctamente tendremos que registrarnos en su web https://github.com/join

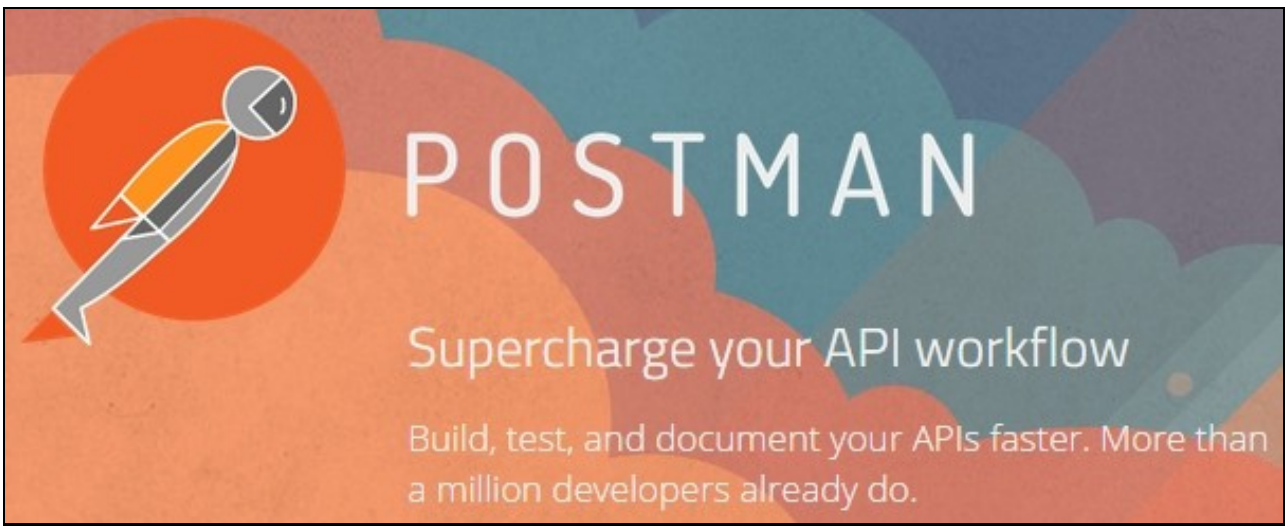
De esta manera podremos crear gratuitamente nuestros **repositorios públicos**. Si queremos que nuestros **repositorios sean privados** tendríamos que pagar una cuota por dicho servicio.

Para instalar Git (opcional):

Para **disponer de comandos Git en cualquier ventana de comandos** del sistema operativo, tendremos que instalar Git. Para este proyecto no lo necesitaremos ya que ejecutaremos los comandos de Git desde el GitShell de GitHub.

- Página oficial de Git: http://git-scm.com/
- Página de descarga de Git: http://git-scm.com/downloads

Instalación de extensión Postman o Advanced REST Client en Google Chrome



Postman es otra extensión para Google Chrome que nos permite ejecutar peticiones a una API REST, muy similar a la extensión **Advanced Rest Client** con la que trabajamos anteriormente..

- Para instalar **Postman** en Chrome, visitar Postman - REST Client
- Para instalar **Advanced REST Client** visitar Advanced REST Client.

Instalación y configuración de Laravel

https://laravel.com/docs/7.x#installation

Si queremos **instalar Laravel en nuestro equipo** podremos hacerlo utilizando **Composer** con las siguientes instrucciones:

- Abrimos una ventana de **terminal**.
- Nos dirigimos a la **carpeta** dónde queremos instalar Laravel.
- En nuestro caso será la carpeta C:\xampp\htdocs\dominios\dominio.local
- Teclear la siguiente instrucción:

```
# Si queremos instalar la versión actual master 5.0.22 (actualmente)
# Si estamos dentro de una carpeta vacía (por ejemplo dominio.local) y queremos que instale el proyecto allí dentro haremos:
composer create-project laravel/laravel  . dev-master

# Composer creará la carpeta_proyecto (o ponemos un . para indicar la carpeta actual) en la ruta en la que estemos situados:
composer create-project laravel/laravel  carpeta_proyecto dev-master

# Si queremos instalar la versión de desarrollo (development)
# Composer creará la carpeta_proyecto en la ruta en la que estemos situados.
composer create-project laravel/laravel  . dev-develop

(Actualizado a 11 Junio 2015)
# Si queremos instalar la versión actual: Laravel Framework (LTS)
composer create-project laravel/laravel . --prefer-dist
```

Error de usuario/contraseña al instalar con Composer

Cuando utilizamos scripts para clonar repositorios desde GitHub, éstos generalmente deben enviar la autenticación del usuario en GitHub que está está clonando el repositorio.

Un error que puede ocurrir al usar composer es que en algún momento nos solicite la autenticación al conectarse a GitHub.com.

En lugar de identificarnos con un usuario/contraseña, GitHub.com nos da la posibilidad de usar en esos scripts un **Token de acceso personal** el cuál se puede utilizar en lugar del nombre de usuario y con la contraseña en blanco.

Utilizando este token personal el script **Composer** cuando se conecte al repositorio de GitHub se identificará y de esta forma indica que somos nosotros los que estamos clonando o descargando un determinado repositorio.

El error que suele dar cuando no tenemos ese **token personal** configurado en composer es algo similar a:

Could not fetch https://api.github.com/repos/laravel/laravel/zipball/7bddbdc2a1f8d9c23205707e74455d74684e3031, enter your GitHub credentials to go over the API rate limit.

A token will be created and stored in C:/Users/wadmin/AppData/Roaming/Composer/auth.json, your password will never be stored. To revoke access to this token you can visit https://github.com/settings/applications

Para solucionar ese error iremos a **GitHub.com** y en **Edit Profile** veremos en el menú izquierdo **Applications** dónde podremos **generar el Token** personalizado.

Pasos a seguir:

- **Loguearse en https://github.com/**
- Acceder a **https://github.com/settings/applications' *o ir a* *Settings*Texto en negrita**.
- Entrar en el menú izquierdo en **Personal access tokens**
- **Generate New Token** (opciones por defecto)
- **Copiar el Token** generado. Por ejemplo: f547bdf6c1c4f4f5154b2fdb4479a80234983279a
- Ir a la **línea de comandos y ejecutar** el siguiente comando:

```
# Comando a ejecutar: composer config -g github-oauth.github.com <oauthtoken>

# Ejemplo de uso:
composer config -g github-oauth.github.com f547bdf6c1c4f4f5154b2fdb4479a80234983279a
```

- **Cerrar la ventana de shell y abrir una nueva**
- **Borrar la carpeta_proyecto creada anteriormente y repetir el comando de creación del proyecto**.

composer create-project laravel/laravel . --prefer-dist

- Si queremos **borrar la caché de Composer** teclearemos:

```
composer clearcache
```

Información final de la instalación de Laravel

Cuando ha terminado el proceso de clonación entonces ya tendremos toda la estructura de Laravel en la carpeta_proyecto que indicamos anteriormente y nos mostrará una **Application Key** que usaremos posteriormente.

```
sebastian/global-state suggests installing ext-uopz (*)
phpunit/php-code-coverage suggests installing ext-xdebug (>=2.2.1)
phpunit/phpunit suggests installing phpunit/php-invoker (~1.1)
Generating autoload files
Generating optimized class loader
Compiling common classes
Application key [BE7Y2zy5jP8EsDsgsLJz3ozaQdHPywnZ] set successfully.
```

Probar la instalación de Laravel

- Laravel por defecto tiene una carpeta **public** que será la única carpeta pública de nuestro dominio.
- Para ello tendremos que modificar nuestro dominio.local para que la carpeta raíz del dominio sea **public**.

```
<VirtualHost *:80>
    DocumentRoot "C:\xampp\htdocs\dominios\dominio.local\public"
    ServerName www.dominio.local
    ServerAlias dominio.local
    <Directory "C:\xampp\htdocs\dominios\dominio.local\public">
                AllowOverride All
                Order Allow,deny
                Allow from all
                Require all granted
    </Directory>
</VirtualHost>
```

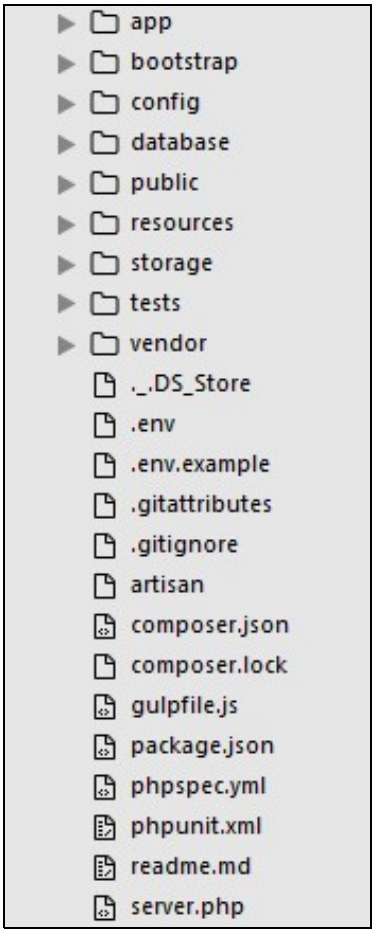
- Aspecto de la página que se muestra al acceder a la aplicación de Laravel.

Creación de estructura y componentes de una API REST con LARAVEL

Estructura de LARAVEL

Laravel sigue la **metodología MVC**. Veamos una pequeña introducción a las rutas principales de este framework.

Cuando abrimos la **estructura de Laravel** veremos algo similar a la imagen siguiente:



- **app**: Contiene los **Modelos**.
- **resources/views**: Contiene las **Vistas**.
- **app/Http/Controllers**: Contiene los **Controladores**.
- **/routes**: Se definen las **Rutas**.
- **app/config/app.php**: Contiene configuración general de la aplicación (debug, timezone, locales, aliases, etc.)
- **public**: Carpeta pública desde dónde se inicia el proceso de ejecución de una aplicación Laravel.
- **readme.md**: Fichero con información referente a la API. Se mostrará automáticamente en la página de GitHub.com

Si abrimos cualquier fichero de ejemplo generalmente veremos que se hace uso de **espacios de nombres: namespace** para facilitar el orden y el uso de más librerías sin que causen problemas al usarlas dentro de otros archivos.

Creación de modelos

Información referente al modelo que vamos a crear en Laravel

Vamos a crear una **API REST de Fabricantes de Aviones**.

- Conceptos sobre API REST
- Un fabricante como Boeing fabricará muchos modelos de aviones
- Un avión es fabricado por 1 fabricante.

Avion

- #Serie (auto incremental)
- Modelo
- Longitud
- Capacidad
- Velocidad
- Alcance

Fabricante

- #Id (auto incremental)
- Nombre
- Direccion
- Telefono

Creación de los modelos en Laravel con PHP Artisan

Laravel en su parte del Modelo usa **Eloquent ORM**. ORM significa Object-Relational Mapping y es una técnica de programación para convertir datos en un sistema orientado a objetos a una base de datos relacional. Además Eloquent ORM implementa el patrón de diseño Active Record.

INFORMACIÓN IMPORTANTE SOBRE PATRÓN ACTIVE RECORD.

1. La librería **Eloquent ORM de Laravel** proporciona una implementación del patrón de diseño de software **Active Record** el cuál se caracteriza por almacenar en memoria objetos de datos que luego se guardarán en bases de datos relacionales.
2. En el patrón **Active Record** cada **tabla o vista es encapsulada en una clase** y **cada objeto que hagamos de esa clase contendrá una fila de la tabla**. Dicha clase se encargaría de implementar todas las operaciones de consultas y modificaciones de una tabla concreta de la base de datos.
3. Cuando se modifican los datos de un objeto de esa clase se está actualizando la fila correspondiente a ese objeto.
4. Cuando se crea un nuevo objeto de esa clase se está creando una nueva fila en la tabla.
5. Cuando se borra un objeto se está borrando una fila de la tabla correspondiente.
6. **Eloquent ORM** en su clase **Model** proporciona múltiples propiedades y métodos que nos permitirán realizar la mayor parte de las tareas que necesitemos con las tablas.
7. **DOCUMENTACIÓN DE LARAVEL ELOQUENT ORM**
8. **Aprende a usar Eloquent en Laravel: Protected, fillable, guarded**

Creación de los modelos en Laravel

- A la hora de crear el modelo tendremos que crear **un modelo para cada tabla** y lo haremos dentro de la carpeta **app**.
- La forma rápida de crear una plantilla para cada modelo es utilizando **PHP Artisan** (utilidad incluída en el framework Laravel). **Artisan** es el nombre de una interfaz en línea de comandos incluída con Laravel.
- **Artisan** proporciona un número de comandos muy útiles a la hora de desarrollar aplicaciones. Por ejemplo nos permitirá generar plantillas para los modelos, generar las tablas MySQL en el servidor, hacer rollbacks, rellenar datos en las tablas de forma automática, etc.
- **DOCUMENTACIÓN SOBRE ARTISAN**

Pasos a seguir para crear cada modelo:

- Abrimos una ventana de comandos en la carpeta principal de Laravel.

```
# Ayuda sobre los comandos de PHP Artisan
php artisan list
```

- Para crear el **modelo para Fabricantes** teclear:

```
php  artisan  make:model  Fabricante
```

- Para crear el **modelo para Aviones** teclear:

```
php  artisan  make:model  Avion
```

- **Plantilla genérica de un Modelo** obtenida con PHP Artisan:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Avion extends Model
{
    //
}
```

- Contenido final del fichero **app\Avion.php**:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Avion extends Model
{
    // Nombre de la tabla en MySQL.
    protected $table='aviones';

    // Eloquent asume que cada tabla tiene una clave primaria con una columna llamada id.
    // Si éste no fuera el caso entonces hay que indicar cuál es nuestra clave primaria en la tabla:
    protected $primaryKey = 'serie';

    /*
    $guarded permite especificar qué campos no queremos que se asignen al modelo. Es decir, se asignan todos excepto los especificad

    Ejemplo:
    protected $guarded= ['is_admin'];

    Y $fillable te permite especificar qué campos sí quieres que se guarden en la base de datos. Es decir, se asignan únicamente los

    Ejemplo:
    protected $fillable= ['modelo','longitud','capacidad','velocidad','alcance'];

    Ambas propiedades se usan en los 'asignamientos de datos en masa' en los modelos Eloquent y permiten protegerse en caso de que u

    Estas propiedades te permiten especificar qué datos se asignarán al modelo en los métodos en los que se usan 'asignamientos en m
    Y, por tanto, qué datos se guardarán posteriormente en la base de datos.
```

```
    Ambos son excluyentes y, por tanto, deberías utilizar sólo uno de ellos. Si declaras ambos, sólo se tendrá en cuenta el contenid

    La diferencia fundamental entre usar uno u otro, está en cómo Laravel procesa ambas propiedades. Si usamos $fillable, cada vez q

    Si se nos olvida incluir un campo crítico que no debería ser modificado desde el formulario, un usuario malicioso podría aprovec
    Si añadimos campos auxiliares del flujo del formulario que no existen en la base de datos, obtendremos un error de SQL porque El
    Por tanto, es recomendable utilizar $fillable en lugar de $guarded.

 */

    // Atributos que se pueden asignar de manera masiva.
    protected $fillable = array('modelo','longitud','capacidad','velocidad','alcance');

    // Aquí ponemos los campos que no queremos que se devuelvan en las consultas.
    protected $hidden = ['created_at','updated_at'];

    // Definimos a continuación la relación de esta tabla con otras.
    // Ejemplos de relaciones:
    // 1 usuario tiene 1 teléfono   ->hasOne() Relación 1:1
    // 1 teléfono pertenece a 1 usuario   ->belongsTo() Relación 1:1 inversa a hasOne()
    // 1 post tiene muchos comentarios  -> hasMany() Relación 1:N
    // 1 comentario pertenece a 1 post ->belongsTo() Relación 1:N inversa a hasMany()
    // 1 usuario puede tener muchos roles  ->belongsToMany()
    //  etc..


    // Relación de Avión con Fabricante:
    public function fabricante()
    {
        // 1 avión pertenece a un Fabricante.
        // $this hace referencia al objeto que tengamos en ese momento de Avión.
        return $this->belongsTo('App\Fabricante');
    }
}
```

- Contenido final del fichero **app\Fabricante.php**:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

// Generalmente cada vez que creamos una clase tenemos que indicar el espacio de nombres
// dónde la estamos creando y suele coincidir con el nombre del directorio.
// El nombre del namespace debe comenzar por UNA LETRA MAYÚSCULA.

// Para más información ver contenido clase Model.php (CTRL + P en Sublime) de Eloquent para ver los atributos disponibles.
// Documentación completa de Eloquent ORM en: https://laravel.com/docs/7.x/eloquent

class Fabricante extends Model
{
// Nombre de la tabla en MySQL.
protected $table="fabricantes";

// Atributos que se pueden asignar de manera masiva.
protected $fillable = array('nombre','direccion','telefono');

// Aquí ponemos los campos que no queremos que se devuelvan en las consultas.
protected $hidden = ['created_at','updated_at'];

// Definimos a continuación la relación de esta tabla con otras.
// Ejemplos de relaciones:
// 1 usuario tiene 1 teléfono   ->hasOne() Relación 1:1
// 1 teléfono pertenece a 1 usuario   ->belongsTo() Relación 1:1 inversa a hasOne()
// 1 post tiene muchos comentarios  -> hasMany() Relación 1:N
// 1 comentario pertenece a 1 post ->belongsTo() Relación 1:N inversa a hasMany()
// 1 usuario puede tener muchos roles  ->belongsToMany()
//  etc..

// Relación de Fabricante con Aviones:
public function aviones()
{
// 1 fabricante tiene muchos aviones
```

```
// $this hace referencia al objeto que tengamos en ese momento de Fabricante.
return $this->hasMany('App\Avion');
}
}
```

Creación de Migrations con Laravel y PHP Artisan

Una vez creado el Modelo, vamos a crear la **estructura de las tablas en MySQL** utilizando **PHP Artisan**. Al usar PHP Artisan nos facilita hacer rollbacks, poblar con datos las tablas, etc..

- Las Migrations (desplazamientos o migraciones) son un tipo de **control de versiones pero para bases de datos**.
- Nos permiten modificar el esquema de la base de datos y estar actualizados correctamente del estado del mismo. Es decir con las migrations podremos crear el esquema de tablas, cubrir datos en la tablas, hacer rollback para volver a estados iniciales, etc...
- Las migrations en Laravel se encuentran en la carpeta **database/migrations**.
- Para **cada tabla de la base de datos** tendremos **1 archivo de migrations**.
- A la hora de ejecutar las migrations Laravel ejecuta los archivos de migration que están en la carpeta **database/migrations**.

Pasos a seguir:

- Primero tenemos que **crear una base de datos, usuario y contraseña con PHPMyAdmin**, ya que PHP Artisan no puede hacer esa tarea por nosotros.
- Configuraremos los **parámetros de conexión** a nuestra base de datos en el fichero **.env** que se encuentra en la ruta **raíz de Laravel**.

```
// Contenido de ejemplo de un fichero .env
APP_ENV=local
APP_DEBUG=true
APP_KEY=q6D00CBznMIdQZuAeTI1wGesD88vRb7J

DB_HOST=localhost
DB_DATABASE=c2base2
DB_USERNAME=c2base2
DB_PASSWORD=xxxxxxxxx

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
```

- Una vez configurada la conexión a la base de datos procederemos a **crear una plantilla de Migration** para cada tabla.
- **ATENCIÓN**: si hemos creado las plantillas de Modelo sin usar PHP Artisan, entonces borrarremos todas las migrations que aparecen por defecto en la carpeta **database/migrations**.
- Si necesitamos crear las plantillas de Migrations para nuestras tablas por que hemos borrado todos los ficheros podemos ejecutar el siguiente comando para cada una de las tablas:

```
# Crearemos primero la plantilla de migration para Fabricantes ya que Aviones tiene una relación que depende de Fabricantes.
# La opción de --create=nombretabla lo que hace es poner el nombre de la tabla dentro de la plantilla.
php artisan make:migration fabricantes_migration --create=fabricantes

# A continuación la plantilla de migration para aviones.
# Con la opción --create=tabla pondrá el nombre de la tabla incluído en la plantilla de la migración.
php artisan make:migration aviones_migration --create=aviones
```

- Editaremos a continuación cada fichero de migrations, indicando tipos de campos, etc...
- Referencia de las tablas y campos que configuraremos dentro de las plantillas de Migration:

```
=> FABRICANTES
* #Id  (auto incremental)
* Nombre string
* Direccion string
* Telefono integer

=> AVIONES
```

```
* #Serie (auto incremental)
* Modelo string
* Longitud float
* Capacidad integer
* Velocidad integer
* Alcance integer
```

- Contenido del fichero **database/migrations/xxxx_fabricantes_migrations.php**:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class FabricantesMigration extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
public function up()
    {
        Schema::create('fabricantes', function(Blueprint $table)
        {
            $table->increments('id');
            $table->string('nombre');
            $table->string('direccion');
            $table->string('telefono');

            // Para que también cree automáticamente los campos timestamps (created_at, updated_at)
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('fabricantes');
    }
}
```

- Contenido del fichero **database/migrations/xxxx_aviones_migrations.php**:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class AvionesMigration extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('aviones', function(Blueprint $table)
        {
            $table->increments('serie');
            $table->string('modelo');
            $table->float('longitud');
            $table->integer('capacidad');
            $table->integer('velocidad');
            $table->integer('alcance');
```

```
            // Añadimos la clave foránea con Fabricante. fabricante_id
            // Acordarse de añadir al array protected $fillable del fichero de modelo "Avion.php" la nueva columna:
            // protected $fillable = array('modelo','longitud','capacidad','velocidad','alcance','fabricante_id');
            $table->integer('fabricante_id')->unsigned();

            // Indicamos cual es la clave foránea de esta tabla:
            $table->foreign('fabricante_id')->references('id')->on('fabricantes');

            // Para que también cree automáticamente los campos timestamps (created_at, updated_at)
            $table->timestamps();
        });
    }


    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('aviones');
    }
}
```

- **ATENCIÓN**:: En **XAMPP** para que funcione correctamente **PDO MySQL** que es el sistema de acceso a base de datos que viene por defecto en XAMPP, hay que habilitar dicha extensión editando el fichero **C:\xampp\php\php.ini** y sacando el ; inicial en la línea **extension=php_pdo_mysql.dll**. Si no lo hacemos al intentar ejecutar las migraciones obtendremos un error del estilo: **[PDOException] could not find driver en Laravel**. Acordarse de **Reiniciar Apache en XAMPP** una vez hecho el cambio.
- PHP Artisan lleva el control de las Migrations a través de una **tabla** llamada **migrations** que tendríamos que tener en el MySQL y que en las últimas versiones de Laravel ya instala de forma automática.
- Si quisiéramos instalarla de forma manual tendríamos que ejecutar: php artisan migrate:install (pero ya digo que ahora no es necesario hacerlo).
- Para **Ejecutar todas las migrations** para que cree las tablas en el MySQL:

```
php artisan migrate

#Migrated: 2015_04_09_105558_fabricantes_migration
#Migrated: 2015_04_09_105827_aviones_migration

# Si vamos a PHPMyAdmin veremos que ha aparecido una nueva tabla llamada migrations y el resto de tablas.

# Si queremos volver a poner la base de datos en su estado inicial podemos hacerlo con:
php artisan migrate:refresh

# Podemos borrar en cualquier momento todas las tablas de MySQL y si ejecutamos
php artisan migrate

# Se volverán a crear de nuevo todas las tablas.
```

Creación de Seeders para tablas con Laravel y PHP Artisan

- Con **PHP Artisan** podremos **llenar de forma masiva las tablas** con datos de ejemplo, utilizando lo que se conoce como **Seeders**.
- Los **Seeders** son una serie de instruciones en las cuales indicamos cómo queremos que se llenen las tablas con datos.
- Los **Seeders** se encuentran en la carpeta **database/seeds**

Pasos a seguir para configurar los seeders:

- Abrir el fichero **database/seeds/DatabaseSeeder.php**
- En ese fichero haremos las llamadas a los seeders que queremos que ejecute. Es importante si queremos un orden (por si los datos de una tabla están relacionados con otra, por ejemplo).
- **Si nos da igual el orden de ejecución de los seeders, no es necesario indicar las llamadas en el fichero database/seeds/DatabaseSeeder.php**.

Código de ejemplo del fichero **database/seeds/DatabaseSeeder.php**:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder {

/**
         * Run the database seeds.
         *
         * @return void
         */
public function run()
{
Model::unguard();

$this->call('FabricanteSeeder');
             $this->call('AvionSeeder');
$this->call('UserTableSeeder');
}

}
```

- Creamos un fichero de **Seeder** para cada tabla **FabricanteSeeder.php** y **AvionSeeder.php**.
- Usaremos para ello el comando:

```
php artisan make:seeder FabricanteSeeder
php artisan make:seeder AvionSeeder
```

- Para poder configurar con qué información rellenamos las tablas necesitamos un proyecto adicional que no viene con Laravel que se llama **Faker**, el cuál nos permitirá generar letras, números y textos aleatorios.
- Podemos buscar ese proyecto con **Composer**:

```
composer search faker

# fzaninotto/faker Faker is a PHP library that generates fake data for you.
# fzaninotto/faker
# gourmet/faker Faker support for CakePHP 3
# typo3/faker
# bobthecow/faker The easiest way to generate fake data in PHP
# yiisoft/yii2-faker Fixture generator. The Faker integration for the Yii framework.
# tomaj/faker Fork of php Faker PHP library with custom Data provider.
# willdurand/faker-bundle Put the awesome Faker lib into the DIC and populate your database with fake data.
# league/factory-muffin-faker The goal of this package is to wrap faker to make it super easy to use with factory muffin.
# davidbadura/faker-bundle
# coduo/tutu-faker-extension This extensions integrates fzaninotto/faker with TuTu.
# emanueleminotto/faker-service-provider Faker Service Provider for Silex
# denheck/faker-context Behat context for generating test data
# burriko/cake-faker CakePHP fixtures plugin using Faker.
# vegas-cmf/faker Vegas CMF Fake Data Generator

# El proyecto que nos interesa es '''fzaninotto/faker'''
```

- **Instalamos Faker** mediante **Composer**:

```
composer require fzaninotto/faker --dev
```

- Pasamos a configurar cada fichero de Seeder empleando las instrucciones de Faker. Documentación de Faker en: https://github.com/fzaninotto/Faker

- Contenido del fichero **database/seeds/FabricanteSeeder.php**:

```
<?php

use Illuminate\Database\Seeder;

use App\Fabricante;

use Faker\Factory as Faker;
```

```
class FabricanteSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */

    public function run()
    {
        // Creamos una instancia de Faker
        $faker = Faker::create();

        // Creamos un bucle para cubrir 5 fabricantes:
        for ($i=1; $i<=5; $i++)
        {
            // Cuando llamamos al método create del Modelo Fabricante
            // se está creando una nueva fila en la tabla.
            Fabricante::create(
                [
                    'nombre'=>$faker->name(),
                    'direccion'=>$faker->streetName(),
                    'telefono'=>$faker->phoneNumber()
                ]
            );
        }
    }
}
```

- Contenido del fichero **database/seeds/AvionSeeder.php**:

```
<?php

use Illuminate\Database\Seeder;

// Hace uso del modelo de Fabricante.
use App\Fabricante;

// Hace uso del modelo de Avion.
use App\Avion;

// Le indicamos que utilice también Faker.
// Información sobre Faker: https://github.com/fzaninotto/Faker
use Faker\Factory as Faker;

class AvionSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
   public function run()
    {
        // Creamos una instancia de Faker
        $faker = Faker::create();

        // Para cubrir los aviones tenemos que tener en cuenta qué fabricantes tenemos.
        // Para que la clave foránea no nos de problemas.
        // Averiguamos cuantos fabricantes hay en la tabla.
        $cuantos= Fabricante::all()->count();

        // Creamos un bucle para cubrir 20 aviones:
        for ($i=1; $i<=20; $i++)
        {
            // Cuando llamamos al método create del Modelo Avion
            // se está creando una nueva fila en la tabla.
            Avion::create(
                [
                 'modelo'=>$faker->word(),
```

```
                'longitud'=>$faker->randomFloat(2,10,150),
                'capacidad'=>$faker->randomNumber(3),  // de 3 dígitos como máximo.
                'velocidad'=>$faker->randomNumber(4),  // de 4 dígitos como máximo.
                'alcance'=>$faker->randomNumber(),
                'fabricante_id'=>$faker->numberBetween(1,$cuantos)
            ]
        );
    }

}
}
```

- Ahora nos queda ejecutar el **comando que insertará registros en las tablas**:

```
php artisan db:seed
```

- Si nos da algún tipo de error de Clase no Encontrada (suele ocurrir al hacer los seeds, ejecutaremos este comando:

```
composer dumpautoload
```

- Si queremos volver a poner la base de datos en su estado inicial y ejecutar el primer seed podemos hacerlo con:

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

- Si nos diese algún tipo de error en el require de la clase Faker haremos:

```
composer update fzaninotto/faker
# Se encarga de descargar el código de la clase y actualizar el fichero composer.json de nuestra aplicación.
```

Creación de Controladores y Rutas API RESTful en Laravel

Creación de los Controladores

Los controladores de Laravel nos permiten agrupar y definir la lógica de las rutas dentro de clases específicas sin que tengamos que crear cada una de las rutas dentro del fichero **app/Http/routes.php**.

- **Documentación de LARAVEL sobre Controladores**

- Los **Controladores de Laravel**, nos permiten hacer la siguientes tareas:
    - En lugar de definir toda la lógica de las rutas en el fichero **app/Http/routes.php**, se puede organizar toda esa lógica a través de un **Controller**.
    - En los Controllers se puede agrupar toda la lógica de las peticiones HTTP dentro de una misma clase.
    - Los Controllers se almacenan típicamente en el directorio **app/Http/Controllers**.
    - Para construir especialmente las rutas o recursos de una API REST disponemos de una ayuda valiosísima en Laravel llamada **Recursos de Controladores RESTful**: **http://laravel.com/docs/5.0/controllers#RESTful-resource-controllers** que es lo que utilizaremos en esta aplicación.

Generación de las plantillas de Controladores para Fabricante y FabricanteAvion y Avion con PHP Artisan y modificaciones básicas

```
php artisan make:controller FabricanteController --resource
# Controller created successfully.

php artisan make:controller FabricanteAvionController --resource
# Controller created successfully.

php artisan make:controller AvionController --resource
# Controller created successfully.

# Aparecerán en la carpeta '''app/Http/Controllers''' 3 ficheros nuevos: '''FabricanteController.php''' y '''FabricanteAvionControll
```

En cada **Controlador programaremos las tareas a realizar para cada una de las peticiones de la API RESTful**.

- Contenido del fichero **app/Http/Controllers/FabricanteController.php**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

class FabricanteController extends Controller {

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index()
{
// Devolverá todos los fabricantes.
return "Mostrando todos los fabricantes de la base de datos.";
}

/**
         * Show the form for creating a new resource.
         *
         * @return Response
         */
public function create()
{
//
return "Se muestra formulario para crear un fabricante.";

}

/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store()
{
//
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
//
return "Se muestra Fabricante con id: $id";
}

/**
         * Show the form for editing the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function edit($id)
{
//
return "Se muestra formulario para editar Fabricante con id: $id";
}

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
```

```
         */
public function update($id)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
{
//
}

}
```

- Contenido del fichero **app/Http/Controllers/FabricanteAvionController.php**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

class FabricanteAvionController extends Controller {

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index($idFabricante)
{
// Devolverá todos los aviones.
return "Mostrando los aviones del fabricante con Id $idFabricante";
}

/**
         * Show the form for creating a new resource.
         *
         * @return Response
         */
public function create($idFabricante)
{
//
return "Se muestra formulario para crear un avión del fabricante $idFabricante.";
}

/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store()
{
//
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($idFabricante,$idAvion)
{
//
return "Se muestra avión $idAvion del fabricante $idFabricante";
}
```

```
/**
         * Show the form for editing the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function edit($idFabricante,$idAvion)
{
//
return "Se muestra formulario para editar el avión $idAvion del fabricante $idFabricante";
}

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($idFabricante,$idAvion)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($idFabricante,$idAvion)
{
//
}
}
```

- Contenido del fichero **app/Http/Controllers/AvionController.php**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

class AvionController extends Controller {

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index()
{
//
}

/**
         * Show the form for creating a new resource.
         *
         * @return Response
         */
public function create()
{
//
}

/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store()
```

```
{
//
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
//
}

/**
         * Show the form for editing the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function edit($id)
{
//
}

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($id)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
{
//
}

}
```

Creación de las Rutas de la API RESTful

Rutas por defecto de la Aplicación

Documentación oficial de Rutas en Laravel

- Es el momento de comenzar a crear las rutas disponibles en nuestro proyecto para la API RESTful.
- La mayor parte de las rutas de nuestro proyecto se definen en **app**/**Http**/**routes.php** o /**routes** (nuevas versiones de Laravel). La forma más básica de rutas admite una URI y un closure() (función anónima). Por ejemplo:

```
Route::get('/', function()
{
    return 'Hola Mundo.';
});
```

- Para **verificar las rutas disponibles en el proyecto** podemos hacerlo empleando **PHP Artisan**:

```
# Ayuda sobre php artisan
php artisan list
```

```
# Listado de las rutas actuales del proyecto
php artisan route:list

// Mostrará algo como ésto:

+--------+----------+----------+------+---------+--------------+
| Domain | Method   | URI      | Name | Action  | Middleware   |
+--------+----------+----------+------+---------+--------------+
|        | GET|HEAD | /        |      | Closure | web          |
|        | GET|HEAD | api/user |      | Closure | api,auth:api |
+--------+----------+----------+------+---------+--------------+
```

Rutas Resource a partir de los Controladores

Todas las rutas de Laravel se definen en sus archivos de ruta, que se encuentran en el directorio de /**routes**. Estos archivos son cargados automáticamente por el framework.

- El archivo **routes/web.php** define rutas que son para su interfaz web. A estas rutas se les asigna el grupo de **middleware web**, que proporciona características como el **estado de la sesión** y la **protección CSRF**.
- Las rutas en **routes/api.php** no tienen estado y se les asigna el grupo de **middleware api**.

- Editaremos el fichero **routes/api.php** añadiendo las rutas a los **recursos de controladores RESTful** creados anteriormente:

```
<?php

use Illuminate\Http\Request;

/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

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

// resource recibe nos parámetros(URI del recurso, Controlador que gestionará las peticiones)
Route::resource('fabricantes','FabricanteController');   // Todos los métodos menos Edit que mostraría un formulario de edición.

// Si queremos dar  la funcionalidad de ver todos los aviones tendremos que crear una ruta específica.
// Pero de aviones solamente necesitamos solamente los métodos index y show.
// Lo correcto sería hacerlo así:
Route::resource('aviones','AvionController');

/*
 php artisan route:list

+--------+-----------+-----------------------------------+---------------------+-------------------------------------------------+
| Domain | Method    | URI                               | Name                | Action                                          |
+--------+-----------+-----------------------------------+---------------------+-------------------------------------------------+
|        | GET|HEAD  | /                                 |                     | Closure                                         |
|        | POST      | api/aviones                       | aviones.store       | App\Http\Controllers\AvionController@store      |
|        | GET|HEAD  | api/aviones                       | aviones.index       | App\Http\Controllers\AvionController@index      |
|        | GET|HEAD  | api/aviones/create                | aviones.create      | App\Http\Controllers\AvionController@create     |
|        | GET|HEAD  | api/aviones/{avione}              | aviones.show        | App\Http\Controllers\AvionController@show       |
|        | PUT|PATCH | api/aviones/{avione}              | aviones.update      | App\Http\Controllers\AvionController@update     |
|        | DELETE    | api/aviones/{avione}              | aviones.destroy     | App\Http\Controllers\AvionController@destroy    |
|        | GET|HEAD  | api/aviones/{avione}/edit         | aviones.edit        | App\Http\Controllers\AvionController@edit       |
|        | GET|HEAD  | api/fabricantes                   | fabricantes.index   | App\Http\Controllers\FabricanteController@index |
|        | POST      | api/fabricantes                   | fabricantes.store   | App\Http\Controllers\FabricanteController@store |
|        | GET|HEAD  | api/fabricantes/create            | fabricantes.create  | App\Http\Controllers\FabricanteController@create|
|        | GET|HEAD  | api/fabricantes/{fabricante}      | fabricantes.show    | App\Http\Controllers\FabricanteController@show  |
|        | PUT|PATCH | api/fabricantes/{fabricante}      | fabricantes.update  | App\Http\Controllers\FabricanteController@update|
|        | DELETE    | api/fabricantes/{fabricante}      | fabricantes.destroy | App\Http\Controllers\FabricanteController@destroy|
```

```
|        | GET|HEAD  | api/fabricantes/{fabricante}/edit | fabricantes.edit    | App\Http\Controllers\FabricanteController@edit    |
|        | GET|HEAD  | api/user                          |                     | Closure                                           |
+--------+----------+-----------------------------------+---------------------+---------------------------------------------------+

*/
```

- La ruta /**api**/**aviones**/**create** (mostraría un formulario para dar de alta un avión)
- La ruta /**api**/**aviones**/**post** (nos permitiría grabar ese avión)
- Viendo esas dos rutas nos damos cuenta de que tenemos un **error conceptual**. No tenemos forma de crear un avión y pasarle al modelo el id del fabricante para que pueda establecer correctamente la clave foránea.
- **Un avión por si sólo no puede existir si no tenemos un fabricante que lo ha fabricado**, por lo tanto la colección principal es fabricantes y necesitaremos definir lo que se conoce como **recursos anidados** para poder crear un avión de un fabricante.

- Las rutas quedarían entonces así:

```
<?php

use Illuminate\Http\Request;

/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

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

// resource recibe nos parámetros(URI del recurso, Controlador que gestionará las peticiones)
Route::resource('fabricantes','FabricanteController');   // Todos los métodos menos Edit que mostraría un formulario de edición.

// Si queremos dar  la funcionalidad de ver todos los aviones tendremos que crear una ruta específica.
// Pero de aviones solamente necesitamos solamente los métodos index y show.
// Lo correcto sería hacerlo así:
Route::resource('aviones','AvionController');

// Como la clase principal es fabricantes y un avión no se puede crear si no le indicamos el fabricante,
// entonces necesitaremos crear lo que se conoce como  "Recurso Anidado" de fabricantes con aviones.
// Definición del recurso anidado:
Route::resource('fabricantes.aviones','FabricanteAvionController');

/*
php artisan route:list

+--------+-----------+-------------------------------------------------------+------------------------------+-------------------------
| Domain | Method    | URI                                                   | Name                         | Action
+--------+-----------+-------------------------------------------------------+------------------------------+-------------------------
|        | GET|HEAD  | /                                                     |                              | Closure
|        | GET|HEAD  | api/aviones                                           | aviones.index                | App\Http\Controllers\Avion
|        | POST      | api/aviones                                           | aviones.store                | App\Http\Controllers\Avion
|        | GET|HEAD  | api/aviones/create                                    | aviones.create               | App\Http\Controllers\Avion
|        | DELETE    | api/aviones/{avione}                                  | aviones.destroy              | App\Http\Controllers\Avion
|        | PUT|PATCH | api/aviones/{avione}                                  | aviones.update               | App\Http\Controllers\Avion
|        | GET|HEAD  | api/aviones/{avione}                                  | aviones.show                 | App\Http\Controllers\Avion
|        | GET|HEAD  | api/aviones/{avione}/edit                             | aviones.edit                 | App\Http\Controllers\Avion
|        | GET|HEAD  | api/fabricantes                                       | fabricantes.index            | App\Http\Controllers\Fabri
|        | POST      | api/fabricantes                                       | fabricantes.store            | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/fabricantes/create                                | fabricantes.create           | App\Http\Controllers\Fabri
|        | DELETE    | api/fabricantes/{fabricante}                          | fabricantes.destroy          | App\Http\Controllers\Fabri
|        | PUT|PATCH | api/fabricantes/{fabricante}                          | fabricantes.update           | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/fabricantes/{fabricante}                          | fabricantes.show             | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/fabricantes/{fabricante}/aviones                  | fabricantes.aviones.index    | App\Http\Controllers\Fabri
|        | POST      | api/fabricantes/{fabricante}/aviones                  | fabricantes.aviones.store    | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/fabricantes/{fabricante}/aviones/create           | fabricantes.aviones.create   | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/fabricantes/{fabricante}/aviones/{avione}         | fabricantes.aviones.show     | App\Http\Controllers\Fabri
```

```
|        | PUT|PATCH | api/fabricantes/{fabricante}/aviones/{avione}      | fabricantes.aviones.update  | App\Http\Controllers\Fabri
|        | DELETE    | api/fabricantes/{fabricante}/aviones/{avione}      | fabricantes.aviones.destroy | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/fabricantes/{fabricante}/aviones/{avione}/edit | fabricantes.aviones.edit    | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/fabricantes/{fabricante}/edit                  | fabricantes.edit            | App\Http\Controllers\Fabri
|        | GET|HEAD  | api/user                                           |                             | Closure
+--------+-----------+----------------------------------------------------+-----------------------------+-----------------------------


*/
```

- Ahora tendremos que ajustar las rutas duplicadas o que no son necesarias:
- No vamos a usar formularios para dar de alta de Fabricantes ni editar, por lo que tendremos que eliminar las rutas /edit y /create: ['except'=>['edit','create']]
- De aviones solamente necesitamos index (para mostrar todos los aviones) y show(para ver los datos de un avion en particular: ['only'=>['index','show']]
- De fabricantesaviones no necesitamos ni show, edit o create: ['except'=>['show','edit','create']]. El método show de fabricantes.aviones.show **no es necesario, ya que ese método se usa para mostrar un avión y ya tenemos un aviones.show que hace esa tarea.**

```
<?php

use Illuminate\Http\Request;

/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

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

// resource recibe nos parámetros(URI del recurso, Controlador que gestionará las peticiones)
Route::resource('fabricantes','FabricanteController',['except'=>['edit','create'] ]);   // Todos los métodos menos Edit que mostraría

// Si queremos dar  la funcionalidad de ver todos los aviones tendremos que crear una ruta específica.
// Pero de aviones solamente necesitamos solamente los métodos index y show.
// Lo correcto sería hacerlo así:
Route::resource('aviones','AvionController',[ 'only'=>['index','show'] ]); // El resto se gestionan en FabricanteAvionController

// Como la clase principal es fabricantes y un avión no se puede crear si no le indicamos el fabricante,
// entonces necesitaremos crear lo que se conoce como  "Recurso Anidado" de fabricantes con aviones.
// Definición del recurso anidado:
Route::resource('fabricantes.aviones','FabricanteAvionController',[ 'except'=>['show','edit','create'] ]);



/*
php artisan route:list

+--------+-----------+-----------------------------------------------+-----------------------------+---------------------------------
| Domain | Method    | URI                                           | Name                        | Action
+--------+-----------+-----------------------------------------------+-----------------------------+---------------------------------
|        | GET|HEAD  | /                                             |                             | Closure
|        | GET|HEAD  | api/aviones                                   | aviones.index               | App\Http\Controllers\AvionContro
|        | GET|HEAD  | api/aviones/{avione}                          | aviones.show                | App\Http\Controllers\AvionContro
|        | GET|HEAD  | api/fabricantes                               | fabricantes.index           | App\Http\Controllers\FabricanteC
|        | POST      | api/fabricantes                               | fabricantes.store           | App\Http\Controllers\FabricanteC
|        | GET|HEAD  | api/fabricantes/{fabricante}                  | fabricantes.show            | App\Http\Controllers\FabricanteC
|        | PUT|PATCH | api/fabricantes/{fabricante}                  | fabricantes.update          | App\Http\Controllers\FabricanteC
|        | DELETE    | api/fabricantes/{fabricante}                  | fabricantes.destroy         | App\Http\Controllers\FabricanteC
|        | GET|HEAD  | api/fabricantes/{fabricante}/aviones          | fabricantes.aviones.index   | App\Http\Controllers\FabricanteA
|        | POST      | api/fabricantes/{fabricante}/aviones          | fabricantes.aviones.store   | App\Http\Controllers\FabricanteA
|        | PUT|PATCH | api/fabricantes/{fabricante}/aviones/{avione} | fabricantes.aviones.update  | App\Http\Controllers\FabricanteA
```

```
|        | DELETE    | api/fabricantes/{fabricante}/aviones/{avione} | fabricantes.aviones.destroy | App\Http\Controllers\FabricanteA
|        | GET|HEAD  | api/user                                      |                             | Closure
+--------+-----------+-----------------------------------------------+-----------------------------+--------------------------------
```

Paginación de resultados en Laravel

- Vamos a ver cómo podemos aplicar paginación a los resultados obtenidos en un listado por ejemplo el de fabricantes.
- Editaremos el método **index()**.

- Contenido del fichero **Http/Controllers/FabricanteController.php**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

// Cargamos Fabricante por que lo usamos más abajo.
use App\Fabricante;

use Response;

// Activamos el uso de las funciones de caché.
use Illuminate\Support\Facades\Cache;

class FabricanteController extends Controller {

public function __construct()
{
$this->middleware('auth.basic',['only'=>['store','update','destroy']]);
}

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index()
{
// return "En el index de Fabricante.";
// Devolvemos un JSON con todos los fabricantes.
// return Fabricante::all();

// Caché se actualizará con nuevos datos cada 15 segundos.
// cachefabricantes es la clave con la que se almacenarán
// los registros obtenidos de Fabricante::all()
// El segundo parámetro son los minutos.
$fabricantes=Cache::remember('cachefabricantes',15/60,function()
{
// Para la paginación en Laravel se usa "Paginator"
// En lugar de devolver
// return Fabricante::all();
// devolveremos return Fabricante::paginate();
//
// Este método paginate() está orientado a interfaces gráficas.
// Paginator tiene un método llamado render() que permite construir
// los enlaces a página siguiente, anterior, etc..
// Para la API RESTFUL usaremos un método más sencillo llamado simplePaginate() que
// aporta la misma funcionalidad
return Fabricante::simplePaginate(10);  // Paginamos cada 10 elementos.

});

// Para devolver un JSON con código de respuesta HTTP sin caché.
// return response()->json(['status'=>'ok', 'data'=>Fabricante::all()],200);

// Devolvemos el JSON usando caché.
// return response()->json(['status'=>'ok', 'data'=>$fabricantes],200);

// Con la paginación lo haremos de la siguiente forma:
```

```
// Devolviendo también la URL a l
return response()->json(['status'=>'ok', 'siguiente'=>$fabricantes->nextPageUrl(),'anterior'=>$fabricantes->previousPageUrl(),'data'=
}

/**
         * Show the form for creating a new resource.
         *
         * @return Response
         */
// No se utiliza este método por que se usaría para mostrar un formulario
// de creación de Fabricantes. Y una API REST no hace eso.
/*
        public function create()
        {
                //
        }
        */

/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store(Request $request)
{
// Método llamado al hacer un POST.
// Comprobamos que recibimos todos los campos.
if (!$request->input('nombre') || !$request->input('direccion') || !$request->input('telefono'))
{
// NO estamos recibiendo los campos necesarios. Devolvemos error.
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para procesar el alta.'])],422);
}

// Insertamos los datos recibidos en la tabla.
$nuevoFabricante=Fabricante::create($request->all());

// Devolvemos la respuesta Http 201 (Created) + los datos del nuevo fabricante + una cabecera de Location + cabecera JSON
$respuesta= Response::make(json_encode(['data'=>$nuevoFabricante]),201)->header('Location','http://www.dominio.local/fabricantes/'.$
return $respuesta;
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
// Corresponde con la ruta /fabricantes/{fabricante}
// Buscamos un fabricante por el ID.
$fabricante=Fabricante::find($id);

// Chequeamos si encontró o no el fabricante
if (! $fabricante)
{
// Se devuelve un array errors con los errores detectados y código 404
return response()->json(['errors'=>Array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// Devolvemos la información encontrada.
return response()->json(['status'=>'ok','data'=>$fabricante],200);

}

/**
         * Show the form for editing the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
/*
        public function edit($id)
```

```
        {
                //
        }
        */

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($id,Request $request)
{
// Vamos a actualizar un fabricante.
// Comprobamos si el fabricante existe. En otro caso devolvemos error.
$fabricante=Fabricante::find($id);

// Si no existe mostramos error.
if (! $fabricante)
{
// Devolvemos error 404.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// Almacenamos en variables para facilitar el uso, los campos recibidos.
$nombre=$request->input('nombre');
$direccion=$request->input('direccion');
$telefono=$request->input('telefono');

// Comprobamos si recibimos petición PATCH(parcial) o PUT (Total)
if ($request->method()=='PATCH')
{
$bandera=false;

// Actualización parcial de datos.
if ($nombre !=null && $nombre!='')
{
$fabricante->nombre=$nombre;
$bandera=true;
}

// Actualización parcial de datos.
if ($direccion !=null && $direccion!='')
{
$fabricante->direccion=$direccion;
$bandera=true;
}

// Actualización parcial de datos.
if ($telefono !=null && $telefono!='')
{
$fabricante->telefono=$telefono;
$bandera=true;
}

if ($bandera)
{
// Grabamos el fabricante.
$fabricante->save();

// Devolvemos un código 200.
return response()->json(['status'=>'ok','data'=>$fabricante],200);
}
else
{
// Devolvemos un código 304 Not Modified.
return response()->json(['errors'=>array(['code'=>304,'message'=>'No se ha modificado ningún dato del fabricante.'])],304);
}
}


// Método PUT actualizamos todos los campos.
// Comprobamos que recibimos todos.
```

```
if (!$nombre || !$direccion || !$telefono)
{
// Se devuelve código 422 Unprocessable Entity.
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan valores para completar el procesamiento.'])],422);
}

// Actualizamos los 3 campos:
$fabricante->nombre=$nombre;
$fabricante->direccion=$direccion;
$fabricante->telefono=$telefono;

// Grabamos el fabricante
$fabricante->save();
return response()->json(['status'=>'ok','data'=>$fabricante],200);

}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
{
// Borrado de un fabricante.
// Ejemplo: /fabricantes/89 por DELETE
// Comprobamos si el fabricante existe o no.
$fabricante=Fabricante::find($id);

if (! $fabricante)
{
// Devolvemos error codigo http 404
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra el fabricante con ese código.'])],404);
}

// Borramos el fabricante y devolvemos código 204
// 204 significa "No Content".
// Este código no muestra texto en el body.
// Si quisiéramos ver el mensaje devolveríamos
// un código 200.
// Antes de borrarlo comprobamos si tiene aviones y si es así
// sacamos un mensaje de error.
// $aviones = $fabricante->aviones()->get();
$aviones = $fabricante->aviones;

if (sizeof($aviones) >0)
{
// Si quisiéramos borrar todos los aviones del fabricante sería:
// $fabricante->aviones->delete();

// Devolvemos un código 409 Conflict.
return response()->json(['errors'=>array(['code'=>409,'message'=>'Este fabricante posee aviones y no puede ser eliminado.'])],409);
}

// Eliminamos el fabricante si no tiene aviones.
$fabricante->delete();

// Se devuelve código 204 No Content.
return response()->json(['code'=>204,'message'=>'Se ha eliminado correctamente el fabricante.'],204);
}

}
```

Versionado de una API RESFful con Laravel

- A la hora de programar una API RESTful deberíamos tener en cuenta el **versionado**, es decir indicar en la ruta de la API la versión de la misma que estamos utilizando.

- Así un ejemplo de una petición a fabricantes cuya ruta era **http://www.dominio.local/api/fabricantes** pasaría a la nueva dirección con la versión incluída: **http://www.dominio.local/api/v1/fabricantes**
- Una forma muy sencilla de crear eso es añadiendo un prefijo con la versión (v1 por ejemplo) para un grupo de rutas de la API:

```
<?php

use Illuminate\Http\Request;

/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

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

// Versionado de la API.
Route::prefix('v1')->group(function () {

    // resource recibe nos parámetros(URI del recurso, Controlador que gestionará las peticiones)
    Route::resource('fabricantes','FabricanteController',['except'=>['edit','create'] ]);   // Todos los métodos menos Edit que mostr

    // Si queremos dar  la funcionalidad de ver todos los aviones tendremos que crear una ruta específica.
    // Pero de aviones solamente necesitamos solamente los métodos index y show.
    // Lo correcto sería hacerlo así:
    Route::resource('aviones','AvionController',[ 'only'=>['index','show'] ]); // El resto se gestionan en FabricanteAvionController

    // Como la clase principal es fabricantes y un avión no se puede crear si no le indicamos el fabricante,
    // entonces necesitaremos crear lo que se conoce como  "Recurso Anidado" de fabricantes con aviones.
    // Definición del recurso anidado:
    Route::resource('fabricantes.aviones','FabricanteAvionController',[ 'except'=>['show','edit','create'] ]);
});


/*
php artisan route:list

+--------+-----------+-----------------------------------------------------+------------------------------+-----------------------------
| Domain | Method    | URI                                                 | Name                         | Action
+--------+-----------+-----------------------------------------------------+------------------------------+-----------------------------
|        | GET|HEAD  | /                                                   |                              | Closure
|        | GET|HEAD  | api/user                                            |                              | Closure
|        | GET|HEAD  | api/v1/aviones                                      | aviones.index                | App\Http\Controllers\AvionCo
|        | GET|HEAD  | api/v1/aviones/{avione}                             | aviones.show                 | App\Http\Controllers\AvionCo
|        | GET|HEAD  | api/v1/fabricantes                                  | fabricantes.index            | App\Http\Controllers\Fabrica
|        | POST      | api/v1/fabricantes                                  | fabricantes.store            | App\Http\Controllers\Fabrica
|        | GET|HEAD  | api/v1/fabricantes/{fabricante}                     | fabricantes.show             | App\Http\Controllers\Fabrica
|        | PUT|PATCH | api/v1/fabricantes/{fabricante}                     | fabricantes.update           | App\Http\Controllers\Fabrica
|        | DELETE    | api/v1/fabricantes/{fabricante}                     | fabricantes.destroy          | App\Http\Controllers\Fabrica
|        | GET|HEAD  | api/v1/fabricantes/{fabricante}/aviones             | fabricantes.aviones.index    | App\Http\Controllers\Fabrica
|        | POST      | api/v1/fabricantes/{fabricante}/aviones             | fabricantes.aviones.store    | App\Http\Controllers\Fabrica
|        | PUT|PATCH | api/v1/fabricantes/{fabricante}/aviones/{avione}    | fabricantes.aviones.update   | App\Http\Controllers\Fabrica
|        | DELETE    | api/v1/fabricantes/{fabricante}/aviones/{avione}    | fabricantes.aviones.destroy  | App\Http\Controllers\Fabrica
+--------+-----------+-----------------------------------------------------+------------------------------+-----------------------------

*/
```

Implementación de las operaciones de la API RESTful con Laravel

Es el momento de implementar dentro de los métodos de los Controllers el código encargado de realizar las acciones correspondientes.

Códigos de estado HTTP a utilizar en la API RESTful

HTTP define un set de significativos **códigos de status** que pueden ser devueltos por la API. Éstos pueden ser nivelados para ayudar a los consumidores de la API dirigir sus respuestas de forma apropiada.

De forma genérica **los códigos HTTP que comienzan por los números indicados abajo, tienen el siguiente significado**:

- 200's usados para respuestas con éxito.
- 300's usados para redirecciones.
- 400's usados cuando hay algún problema con la petición.
- 500's usados cuando hay algún problema con el servidor.

Lista de **códigos HTTP que se deberían utilizar en la API RESTful**: ---

- **200 OK** - Respuesta a un exitoso GET, PUT, PATCH o DELETE. Puede ser usado también para un POST que no resulta en una creación.
- **201 Created** ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado con un encabezado Location, apuntando a la ubicación del nuevo recurso.
- **204 No Content** ? [Sin Contenido] Respuesta a una petición exitosa que no devuelve un body (por ejemplo en una petición DELETE)

- **304 Not Modified** ? [No Modificada] Usado cuando el cacheo de encabezados HTTP está activo y el cliente puede usar datos cacheados.

- **400 Bad Request** ? [Petición Errónea] La petición está malformada, como por ejemplo, si el contenido no fue bien parseado. El error se debe mostrar también en el JSON de respuesta.
- **401 Unauthorized** ? [Desautorizada] Cuando los detalles de autenticación son inválidos o no son otorgados. También útil para disparar un popup de autorización si la API es usada desde un navegador.
- **403 Forbidden** ? [Prohibida] Cuando la autenticación es exitosa pero el usuario no tiene permiso al recurso en cuestión.
- **404 Not Found** ? [No encontrada] Cuando un recurso se solicita un recurso no existente.

- **405 Method Not Allowed** ? [Método no permitido] Cuando un método HTTP que está siendo pedido no está permitido para el usuario autenticado.
- **409 Conflict** - [Conflicto] Cuando hay algún conflicto al procesar una petición, por ejemplo en PATCH, POST o DELETE.
- **410 Gone** ? [Retirado] Indica que el recurso en ese endpoint ya no está disponible. Útil como una respuesta en blanco para viejas versiones de la API
- **415 Unsupported Media Type** ? [Tipo de contenido no soportado] Si el tipo de contenido que solicita la petición es incorrecto
- **422 Unprocessable Entity** ? [Entidad improcesable] Utilizada para errores de validación, o cuando por ejemplo faltan campos en una petición.
- **429 Too Many Requests** ? [Demasiadas peticiones] Cuando una petición es rechazada debido a la tasa límite .

- **500 ? Internal Server Error** ? [Error Interno del servidor] Los desarrolladores de API **NO deberían usar este código**. En su lugar se debería loguear el fallo y no devolver respuesta.

--- Más información en: **http://jsonapi.org/format/**

Mostrar los valores de un recurso y de un recurso anidado de la API RESTful

Utilizaremos el **formato JSON para devolver los datos solicitados a una URI de un recurso**.

Implementación de los métodos index() y show():

Código de ejemplo del controlador **app/Http/Controllers/FabricanteController.php**:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;

use App\Fabricante;

class FabricanteController extends Controller {

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index()
{
// Devolverá todos los fabricantes.
// return "Mostrando todos los fabricantes de la base de datos.";
// return Fabricante::all();  No es lo más correcto por que se devolverían todos los registros. Se recomienda usar Filtros.
// Se debería devolver un objeto con una propiedad como mínimo data y el array de resultados en esa propiedad.
// A su vez también es necesario devolver el código HTTP de la respuesta.
//php http://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-RESTful-pragmatica/
// https://cloud.google.com/storage/docs/json_api/v1/status-codes

return response()->json(['status'=>'ok','data'=>Fabricante::all()], 200);
}

/**
         * Show the form for creating a new resource.
         *
         * @return Response
         */
public function create()
{
//
return "Se muestra formulario para crear un fabricante.";

}

/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store()
{
//
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
//
// return "Se muestra Fabricante con id: $id";
// Buscamos un fabricante por el id.
$fabricante=Fabricante::find($id);

// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

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

```
}

/**
         * Show the form for editing the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function edit($id)
{
//
return "Se muestra formulario para editar Fabricante con id: $id";
}

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($id)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
{
//
}

}
```

Código de ejemplo del controlador **app/Http/Controllers/FabricanteAvionController.php**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Fabricante;
use App\Avion;

class FabricanteAvionController extends Controller {

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index($idFabricante)
{
// Devolverá todos los aviones.
//return "Mostrando los aviones del fabricante con Id $idFabricante";
$fabricante=Fabricante::find($idFabricante);

if (! $fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

return response()->json(['status'=>'ok','data'=>$fabricante->aviones()->get()],200);
//return response()->json(['status'=>'ok','data'=>$fabricante->aviones],200);
}
```

```
/**
         * Show the form for creating a new resource.
         *
         * @return Response
         */
public function create($idFabricante)
{
//
return "Se muestra formulario para crear un avión del fabricante $idFabricante.";
}

/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store()
{
//
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($idFabricante,$idAvion)
{
//
return "Se muestra avión $idAvion del fabricante $idFabricante";
}

/**
         * Show the form for editing the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function edit($idFabricante,$idAvion)
{
//
return "Se muestra formulario para editar el avión $idAvion del fabricante $idFabricante";
}

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($idFabricante,$idAvion)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($idFabricante,$idAvion)
{
//
}

}
```

Creación de recursos en la API RESTful

- Vamos a programar a continuación en los controladores los métodos para insertar datos en las tablas a través de la API RESTful.
- Para poder insertar registros se hará una petición **HTTP POST** a la API RESTful.
- Laravel utiliza una **Fachada Request** (es un patrón de diseño orientado a objetos). más información sobre el patrón Facade aquí).
- **Para poder utilizar Request** hay que asegurarse de que si estamos trabajando en un namespace, tenemos que importar la fachada Request usando **use Illuminate\Http\Request;**
- Para poder obtener una **instancia de la actual petición HTTP** a través de **Inyección de Dependencias** (patrón de diseño orientado a objetos en el cuál se suministran objetos a una clase en lugar de ser la la propia clase quién crea el objeto), tendremos que pasar como parámetro al método que lo necesite algo similar a (**Request $datos**). Al pasar ésto como parámetro la fachada Request se encarga de crear el objeto $datos que contendrá todos los datos recibimos del formulario.
- **Documentación sobre Request en Laravel**.

- **Editaremos** el fichero del controlador **app/Http/Controllers/FabricanteController.php**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;

use App\Fabricante;

class FabricanteController extends Controller {

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index()
{
// Devolverá todos los fabricantes.
// return "Mostrando todos los fabricantes de la base de datos.";
// return Fabricante::all();  No es lo más correcto por que se devolverían todos los registros. Se recomienda usar Filtros.
// Se debería devolver un objeto con una propiedad como mínimo data y el array de resultados en esa propiedad.
// A su vez también es necesario devolver el código HTTP de la respuesta.
//php http://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-RESTful-pragmatica/
// https://cloud.google.com/storage/docs/json_api/v1/status-codes

return response()->json(['status'=>'ok','data'=>Fabricante::all()], 200);
}


/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */

// Pasamos como parámetro al método store todas las variables recibidas de tipo Request
// utilizando inyección de dependencias
// Para acceder a Request necesitamos asegurarnos que está cargado use Illuminate\Http\Request;
// Información sobre Request en: http://laravel.com/docs/5.0/requests
// Ejemplo de uso de Request:  $request->input('name');
public function store(Request $request)
{

// Primero comprobaremos si estamos recibiendo todos los campos.
if (!$request->input('nombre') || !$request->input('direccion') || !$request->input('telefono'))
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesable] Utiliza
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],422);
}

// Insertamos una fila en Fabricante con create pasándole todos los datos recibidos.
```

```
// En $request->all() tendremos todos los campos del formulario recibidos.
$nuevoFabricante=Fabricante::create($request->all());

        // Más información sobre respuestas en http://jsonapi.org/format/
        // Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado c
        return response()->json(['data'=>$nuevoFabricante], 201)->header('Location',  url('/api/v1/').'/fabricantes/'.$nuevoFabricant
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
//
// return "Se muestra Fabricante con id: $id";
// Buscamos un fabricante por el id.
$fabricante=Fabricante::find($id);

// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

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

}



/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($id)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
{
//
}

}
```

- **Editaremos** el fichero del controlador **app/Http/Controllers/FabricanteAvionController.php**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Fabricante;
use App\Avion;

class FabricanteAvionController extends Controller {
/**
         * Display a listing of the resource.
         *
```

```
         * @return Response
         */
public function index($idFabricante)
{
// Devolverá todos los aviones.
//return "Mostrando los aviones del fabricante con Id $idFabricante";
$fabricante=Fabricante::find($idFabricante);

if (! $fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

return response()->json(['status'=>'ok','data'=>$fabricante->aviones()->get()],200);
//return response()->json(['status'=>'ok','data'=>$fabricante->aviones],200);
}


/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store(Request $request,$idFabricante)
{
/* Necesitaremos el fabricante_id que lo recibimos en la ruta
                 #Serie (auto incremental)
                Modelo
                Longitud
                Capacidad
                Velocidad
                Alcance */

// Primero comprobaremos si estamos recibiendo todos los campos.
if ( !$request->input('modelo') || !$request->input('longitud') || !$request->input('capacidad') || !$request->input('velocidad') ||
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesable] Utiliza
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],422);
}

// Buscamos el Fabricante.
$fabricante= Fabricante::find($idFabricante);

// Si no existe el fabricante que le hemos pasado mostramos otro código de error de no encontrado.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// Si el fabricante existe entonces lo almacenamos.
// Insertamos una fila en Aviones con create pasándole todos los datos recibidos.
$nuevoAvion=$fabricante->aviones()->create($request->all());

        // Más información sobre respuestas en http://jsonapi.org/format/
        // Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado c
        return response()->json(['data'=>$nuevoAvion], 201)->header('Location',  url('/api/v1/').'/aviones/'.$nuevoAvion->serie);
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($idFabricante,$idAvion)
{
//
return "Se muestra avión $idAvion del fabricante $idFabricante";
}
```

```
/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($idFabricante,$idAvion)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($idFabricante,$idAvion)
{
//
}

}
```

Actualización de recursos en la API RESTful

- Si queremos actualizar datos a través de la API RESTful necesitaremos programar los métodos correspondientes en nuestros controladores.
- Para poder actualizar total o parcialmente registros hará una petición HTTP **PUT** o **PATCH** a la API RESTful.
- El método en el **controlador** que se encarga de gestionar dichas peticiones es **update()**.
- Necesitamos **detectar si la petición es PUT o PATCH** ya que el método es el mismo para los dos casos.
- En una actualización **PUT** se actualizan **todos los datos**.
- En una actualización **PATCH** se actualizarían **solamente algunos datos**.
- Para probar estas peticiones los datos en **POSTMAN** tienen que ir en **x-www-form-urlencoded**. Si se pasan como **form-data** no se recibirá ningún dato.

- **Editaremos** el fichero del controlador **app/Http/Controllers/FabricanteController.php**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;

use App\Fabricante;

class FabricanteController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return response()->json(['status'=>'ok','data'=>Fabricante::all()], 200);
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        return ("muestra formulario de creación de fabricante");
    }

    /**
```

```
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{

    // Primero comprobaremos si estamos recibiendo todos los campos.
    if (!$request->input('nombre') || !$request->input('direccion') || !$request->input('telefono'))
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesa
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],4
    }

    // Insertamos una fila en Fabricante con create pasándole todos los datos recibidos.
    // En $request->all() tendremos todos los campos del formulario recibidos.
    $nuevoFabricante=Fabricante::create($request->all());

    // Más información sobre respuestas en http://jsonapi.org/format/
    // Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado c
    return response()->json(['data'=>$nuevoAvion], 201)->header('Location',  url('/api/v1/').'/aviones/'.$nuevoAvion->serie);
}

/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    $fabricante=Fabricante::find($id);

    // Si no existe ese fabricante devolvemos un error.
    if (!$fabricante)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
    }

    return response()->json(['status'=>'ok','data'=>$fabricante],200);
}

/**
 * Show the form for editing the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    //
}

/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    // Comprobamos si el fabricante que nos están pasando existe o no.
    $fabricante=Fabricante::find($id);

    // Si no existe ese fabricante devolvemos un error.
    if (!$fabricante)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
```

```
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
    }

    // Listado de campos recibidos teóricamente.
    $nombre=$request->input('nombre');
    $direccion=$request->input('direccion');
    $telefono=$request->input('telefono');

    // Necesitamos detectar si estamos recibiendo una petición PUT o PATCH.
    // El método de la petición se sabe a través de $request->method();
    if ($request->method() === 'PATCH')
    {
        // Creamos una bandera para controlar si se ha modificado algún dato en el método PATCH.
        $bandera = false;

        // Actualización parcial de campos.
        if ($nombre)
        {
            $fabricante->nombre = $nombre;
            $bandera=true;
        }

        if ($direccion)
        {
            $fabricante->direccion = $direccion;
            $bandera=true;
        }


        if ($telefono)
        {
            $fabricante->telefono = $telefono;
            $bandera=true;
        }

        if ($bandera)
        {
            // Almacenamos en la base de datos el registro.
            $fabricante->save();
            return response()->json(['status'=>'ok','data'=>$fabricante], 200);
        }
        else
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 304 Not Modified ? [No Modificada] Usado
            // Este código 304 no devuelve ningún body, así que si quisiéramos que se mostrara el mensaje usaríamos un código 20
            return response()->json(['errors'=>array(['code'=>304,'message'=>'No se ha modificado ningún dato de fabricante.'])]
        }
    }


    // Si el método no es PATCH entonces es PUT y tendremos que actualizar todos los datos.
    if (!$nombre || !$direccion || !$telefono)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesa
        return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan valores para completar el procesamiento.'])],42
    }

    $fabricante->nombre = $nombre;
    $fabricante->direccion = $direccion;
    $fabricante->telefono = $telefono;

    // Almacenamos en la base de datos el registro.
    $fabricante->save();
    return response()->json(['status'=>'ok','data'=>$fabricante], 200);
}
/**
 * Remove the specified resource from storage.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id)
```

```
    {
       //
    }
}
```

- **Editaremos** el fichero del controlador **app/Http/Controllers/FabricanteAvionController.php**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Fabricante;
use App\Avion;

class FabricanteAvionController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index($idFabricante)
    {
        // Devolverá todos los aviones.
        //return "Mostrando los aviones del fabricante con Id $idFabricante";
        $fabricante=Fabricante::find($idFabricante);

        if (! $fabricante)
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
            return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
        }

        return response()->json(['status'=>'ok','data'=>$fabricante->aviones()->get()],200);
        //return response()->json(['status'=>'ok','data'=>$fabricante->aviones],200);
    }


    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request,$idFabricante)
    {
        /* Necesitaremos el fabricante_id que lo recibimos en la ruta
         #Serie (auto incremental)
        Modelo
        Longitud
        Capacidad
        Velocidad
        Alcance */

        // Primero comprobaremos si estamos recibiendo todos los campos.
        if ( !$request->input('modelo') || !$request->input('longitud') || !$request->input('capacidad') || !$request->input('velocid
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesa
```

```
            return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],4
        }

        // Buscamos el Fabricante.
        $fabricante= Fabricante::find($idFabricante);

        // Si no existe el fabricante que le hemos pasado mostramos otro código de error de no encontrado.
        if (!$fabricante)
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
            return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
        }

        // Si el fabricante existe entonces lo almacenamos.
        // Insertamos una fila en Aviones con create pasándole todos los datos recibidos.
        $nuevoAvion=$fabricante->aviones()->create($request->all());

        // Más información sobre respuestas en http://jsonapi.org/format/
        // Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado c
        return response()->json(['data'=>$nuevoAvion], 201)->header('Location',  url('/api/v1/').'/aviones/'.$nuevoAvion->serie);
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $idFabricante, $idAvion)
    {
        // Comprobamos si el fabricante que nos están pasando existe o no.
        $fabricante=Fabricante::find($idFabricante);

        // Si no existe ese fabricante devolvemos un error.
        if (!$fabricante)
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
            return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
        }

        // El fabricante existe entonces buscamos el avion que queremos editar asociado a ese fabricante.
        $avion = $fabricante->aviones()->find($idAvion);

        // Si no existe ese avión devolvemos un error.
        if (!$avion)
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
```

```
    return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un avión con ese código asociado al fal
}


// Listado de campos recibidos teóricamente.
$modelo=$request->input('modelo');
$longitud=$request->input('longitud');
$capacidad=$request->input('capacidad');
$velocidad=$request->input('velocidad');
$alcance=$request->input('alcance');

// Necesitamos detectar si estamos recibiendo una petición PUT o PATCH.
// El método de la petición se sabe a través de $request->method();
/*  Modelo      Longitud        Capacidad       Velocidad       Alcance */
if ($request->method() === 'PATCH')
{
    // Creamos una bandera para controlar si se ha modificado algún dato en el método PATCH.
    $bandera = false;

    // Actualización parcial de campos.
    if ($modelo)
    {
        $avion->modelo = $modelo;
        $bandera=true;
    }

    if ($longitud)
    {
        $avion->longitud = $longitud;
        $bandera=true;
    }

    if ($capacidad)
    {
        $avion->capacidad = $capacidad;
        $bandera=true;
    }

    if ($velocidad)
    {
        $avion->velocidad = $velocidad;
        $bandera=true;
    }

    if ($alcance)
    {
        $avion->alcance = $alcance;
        $bandera=true;
    }

    if ($bandera)
    {
        // Almacenamos en la base de datos el registro.
        $avion->save();
        return response()->json(['status'=>'ok','data'=>$avion], 200);
    }
    else
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 304 Not Modified ? [No Modificada] Usado
        // Este código 304 no devuelve ningún body, así que si quisiéramos que se mostrara el mensaje usaríamos un código 20
        return response()->json(['errors'=>array(['code'=>304,'message'=>'No se ha modificado ningún dato del avión.'])],304
    }

}

// Si el método no es PATCH entonces es PUT y tendremos que actualizar todos los datos.
if (!$modelo || !$longitud || !$capacidad || !$velocidad || !$alcance)
{
    // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesa
    return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan valores para completar el procesamiento.'])],42
}

$avion->modelo = $modelo;
```

```
        $avion->longitud = $longitud;
        $avion->capacidad = $capacidad;
        $avion->velocidad = $velocidad;
        $avion->alcance = $alcance;

        // Almacenamos en la base de datos el registro.
        $avion->save();

        return response()->json(['status'=>'ok','data'=>$avion], 200);
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy()
    {
        //
    }
}
```

Borrado de recursos en la API RESTful

- Por último nos falta implementar en la API la opción para eliminar registros.
- Para poder borrar un registro se hará una petición HTTP **DELETE** a la API RESTful.
- El método en el **controlador** que se encarga de gestionar dichas peticiones es **destroy()**.

- **Editaremos** el fichero del controlador **app/Http/Controllers/FabricanteController.php**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;

use App\Fabricante;

class FabricanteController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return response()->json(['status'=>'ok','data'=>Fabricante::all()], 200);
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        return ("muestra formulario de creación de fabricante");
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
```

```
    // Primero comprobaremos si estamos recibiendo todos los campos.
    if (!$request->input('nombre') || !$request->input('direccion') || !$request->input('telefono'))
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesa
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],4
    }

    // Insertamos una fila en Fabricante con create pasándole todos los datos recibidos.
    // En $request->all() tendremos todos los campos del formulario recibidos.
    $nuevoFabricante=Fabricante::create($request->all());

    // Más información sobre respuestas en http://jsonapi.org/format/
    // Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado c
    return response()->json(['data'=>$nuevoAvion], 201)->header('Location',  url('/api/v1/').'/aviones/'.$nuevoAvion->serie);
}

/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    $fabricante=Fabricante::find($id);

    // Si no existe ese fabricante devolvemos un error.
    if (!$fabricante)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
    }

    return response()->json(['status'=>'ok','data'=>$fabricante],200);
}

/**
 * Show the form for editing the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    //
}

/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    // Comprobamos si el fabricante que nos están pasando existe o no.
    $fabricante=Fabricante::find($id);

    // Si no existe ese fabricante devolvemos un error.
    if (!$fabricante)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
    }

    // Listado de campos recibidos teóricamente.
    $nombre=$request->input('nombre');
    $direccion=$request->input('direccion');
    $telefono=$request->input('telefono');
```

```
    // Necesitamos detectar si estamos recibiendo una petición PUT o PATCH.
    // El método de la petición se sabe a través de $request->method();
    if ($request->method() === 'PATCH')
    {
        // Creamos una bandera para controlar si se ha modificado algún dato en el método PATCH.
        $bandera = false;

        // Actualización parcial de campos.
        if ($nombre)
        {
            $fabricante->nombre = $nombre;
            $bandera=true;
        }

        if ($direccion)
        {
            $fabricante->direccion = $direccion;
            $bandera=true;
        }


        if ($telefono)
        {
            $fabricante->telefono = $telefono;
            $bandera=true;
        }

        if ($bandera)
        {
            // Almacenamos en la base de datos el registro.
            $fabricante->save();
            return response()->json(['status'=>'ok','data'=>$fabricante], 200);
        }
        else
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 304 Not Modified ? [No Modificada] Usado
            // Este código 304 no devuelve ningún body, así que si quisiéramos que se mostrara el mensaje usaríamos un código 20
            return response()->json(['errors'=>array(['code'=>304,'message'=>'No se ha modificado ningún dato de fabricante.'])]
        }
    }


    // Si el método no es PATCH entonces es PUT y tendremos que actualizar todos los datos.
    if (!$nombre || !$direccion || !$telefono)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesa
        return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan valores para completar el procesamiento.'])],42
    }

    $fabricante->nombre = $nombre;
    $fabricante->direccion = $direccion;
    $fabricante->telefono = $telefono;

    // Almacenamos en la base de datos el registro.
    $fabricante->save();
    return response()->json(['status'=>'ok','data'=>$fabricante], 200);
}
/**
 * Remove the specified resource from storage.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id)
{
    // Primero eliminaremos todos los aviones de un fabricante y luego el fabricante en si mismo.
    // Comprobamos si el fabricante que nos están pasando existe o no.
    $fabricante=Fabricante::find($id);

    // Si no existe ese fabricante devolvemos un error.
    if (!$fabricante)
    {
```

```
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
            return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
        }

        // El fabricante existe entonces buscamos todos los aviones asociados a ese fabricante.
        $aviones = $fabricante->aviones; // Sin paréntesis obtenemos el array de todos los aviones.

        // Comprobamos si tiene aviones ese fabricante.
        if (sizeof($aviones) > 0)
        {

            // ! Ésta solución no sería el standard !
            /*
            foreach($aviones as $avion)
            {
                $avion->delete();
            }
            */

            // Lo correcto en la API REST sería ésto:

            // Devolveremos un código 409 Conflict - [Conflicto] Cuando hay algún conflicto al procesar una petición, por ejemplo en
            return response()->json(['code'=>409,'message'=>'Este fabricante posee aviones y no puede ser eliminado.'],409);
        }

        // Procedemos por lo tanto a eliminar el fabricante.
        $fabricante->delete();

        // Se usa el código 204 No Content ? [Sin Contenido] Respuesta a una petición exitosa que no devuelve un body (como una peti
        // Este código 204 no devuelve body así que si queremos que se vea el mensaje tendríamos que usar un código de respuesta HTT
        return response()->json(['code'=>204,'message'=>'Se ha eliminado el fabricante correctamente.'],204);

    }
}
```

- **Editaremos** el fichero del controlador **app/Http/Controllers/FabricanteAvionController.php**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Fabricante;
use App\Avion;

class FabricanteAvionController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index($idFabricante)
    {
        // Devolverá todos los aviones.
        //return "Mostrando los aviones del fabricante con Id $idFabricante";
        $fabricante=Fabricante::find($idFabricante);

        if (! $fabricante)
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
            return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
        }

        return response()->json(['status'=>'ok','data'=>$fabricante->aviones()->get()],200);
        //return response()->json(['status'=>'ok','data'=>$fabricante->aviones],200);
    }


    /**
     * Show the form for creating a new resource.
```

```
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request,$idFabricante)
    {
        /* Necesitaremos el fabricante_id que lo recibimos en la ruta
         #Serie (auto incremental)
        Modelo
        Longitud
        Capacidad
        Velocidad
        Alcance */

        // Primero comprobaremos si estamos recibiendo todos los campos.
        if ( !$request->input('modelo') || !$request->input('longitud') || !$request->input('capacidad') || !$request->input('velocid
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesal
            return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],4
        }

        // Buscamos el Fabricante.
        $fabricante= Fabricante::find($idFabricante);

        // Si no existe el fabricante que le hemos pasado mostramos otro código de error de no encontrado.
        if (!$fabricante)
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
            return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
        }

        // Si el fabricante existe entonces lo almacenamos.
        // Insertamos una fila en Aviones con create pasándole todos los datos recibidos.
        $nuevoAvion=$fabricante->aviones()->create($request->all());

        // Más información sobre respuestas en http://jsonapi.org/format/
        // Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado c
        return response()->json(['data'=>$nuevoAvion], 201)->header('Location',  url('/api/v1/').'/aviones/'.$nuevoAvion->serie);
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }
```

```
/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $idFabricante, $idAvion)
{
    // Comprobamos si el fabricante que nos están pasando existe o no.
    $fabricante=Fabricante::find($idFabricante);

    // Si no existe ese fabricante devolvemos un error.
    if (!$fabricante)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
    }

    // El fabricante existe entonces buscamos el avion que queremos editar asociado a ese fabricante.
    $avion = $fabricante->aviones()->find($idAvion);

    // Si no existe ese avión devolvemos un error.
    if (!$avion)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un avión con ese código asociado al fa
    }


    // Listado de campos recibidos teóricamente.
    $modelo=$request->input('modelo');
    $longitud=$request->input('longitud');
    $capacidad=$request->input('capacidad');
    $velocidad=$request->input('velocidad');
    $alcance=$request->input('alcance');

    // Necesitamos detectar si estamos recibiendo una petición PUT o PATCH.
    // El método de la petición se sabe a través de $request->method();
    /*  Modelo      Longitud        Capacidad       Velocidad       Alcance */
    if ($request->method() === 'PATCH')
    {
        // Creamos una bandera para controlar si se ha modificado algún dato en el método PATCH.
        $bandera = false;

        // Actualización parcial de campos.
        if ($modelo)
        {
            $avion->modelo = $modelo;
            $bandera=true;
        }

        if ($longitud)
        {
            $avion->longitud = $longitud;
            $bandera=true;
        }

        if ($capacidad)
        {
            $avion->capacidad = $capacidad;
            $bandera=true;
        }

        if ($velocidad)
        {
            $avion->velocidad = $velocidad;
            $bandera=true;
        }

        if ($alcance)
```

```
        {
            $avion->alcance = $alcance;
            $bandera=true;
        }

        if ($bandera)
        {
            // Almacenamos en la base de datos el registro.
            $avion->save();
            return response()->json(['status'=>'ok','data'=>$avion], 200);
        }
        else
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 304 Not Modified ? [No Modificada] Usado
            // Este código 304 no devuelve ningún body, así que si quisiéramos que se mostrara el mensaje usaríamos un código 20
            return response()->json(['errors'=>array(['code'=>304,'message'=>'No se ha modificado ningún dato del avión.'])],304
        }

    }

    // Si el método no es PATCH entonces es PUT y tendremos que actualizar todos los datos.
    if (!$modelo || !$longitud || !$capacidad || !$velocidad || !$alcance)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesa
        return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan valores para completar el procesamiento.'])],42
    }

    $avion->modelo = $modelo;
    $avion->longitud = $longitud;
    $avion->capacidad = $capacidad;
    $avion->velocidad = $velocidad;
    $avion->alcance = $alcance;

    // Almacenamos en la base de datos el registro.
    $avion->save();

    return response()->json(['status'=>'ok','data'=>$avion], 200);
}

/**
 * Remove the specified resource from storage.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function destroy($idFabricante,$idAvion)
{
    // Comprobamos si el fabricante que nos están pasando existe o no.
    $fabricante=Fabricante::find($idFabricante);

    // Si no existe ese fabricante devolvemos un error.
    if (!$fabricante)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404)
    }

    // El fabricante existe entonces buscamos el avion que queremos borrar asociado a ese fabricante.
    $avion = $fabricante->aviones()->find($idAvion);

    // Si no existe ese avión devolvemos un error.
    if (!$avion)
    {
        // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
        // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
        return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un avión con ese código asociado a ese
    }

    // Procedemos por lo tanto a eliminar el avión.
    $avion->delete();

    // Se usa el código 204 No Content ? [Sin Contenido] Respuesta a una petición exitosa que no devuelve un body (como una peti
```

```
        // Este código 204 no devuelve body así que si queremos que se vea el mensaje tendríamos que usar un código de respuesta HTT
        return response()->json(['code'=>204,'message'=>'Se ha eliminado el avión correctamente.'],204);
    }
}
```

- Contenido del fichero del controlador **app/Http/Controllers/AvionController.php**:

No se ha editado desde la última vez ya que las tareas de Avión de store, update y destroy se han programado en el recurso anidado de FabricanteAvionController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

// Necesitaremos el modelo Avion para ciertas tareas.
use App\Avion;

class AvionController extends Controller {
    /**
     * Display a listing of the resource.
     *
     * @return Response
     */

    public function index()
    {
        // Devolverá todos los fabricantes.
        // return "Mostrando todos los fabricantes de la base de datos.";
        // return Fabricante::all();  No es lo más correcto por que se devolverían todos los registros. Se recomienda usar Filtros.
        // Se debería devolver un objeto con una propiedad como mínimo data y el array de resultados en esa propiedad.
        // A su vez también es necesario devolver el código HTTP de la respuesta.
        // php http://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-RESTful-pragmatica/
        // https://cloud.google.com/storage/docs/json_api/v1/status-codes

        return response()->json(['status'=>'ok','data'=>Avion::all()], 200);
    }


    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        //
        // return "Se muestra Fabricante con id: $id";
        // Buscamos un fabricante por el id.
        $avion=Avion::find($id);

        // Si no existe ese avion devolvemos un error.
        if (!$avion)
        {
            // Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
            // En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
            return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un avión con ese código.'])],404);
        }

        return response()->json(['status'=>'ok','data'=>$avion],200);
    }


// Se han borrado el resto de métodos que no se necesitan.
}
```

Caché de consultas con Laravel para reducir carga en bases de datos

- Se puede acelerar mucho el rendimiento de la base de datos si hacemos caché de consultas, de tal forma que si por ejemplo recibimos una petición la respuesta a esa petición esté disponible en caché durante al menos 15 segundos. De esta forma conseguimos acelerar muchísimo el rendimiento de la aplicación.
- Claro está que dependiendo del tipo de aplicación a veces el sistema de caché no es un sistema válido, pero en muchísimos casos en los que se hace consulta a una base de datos mediante una API REST es un sistema que tendríamos que tener muy en cuenta para acelerar el rendimiento.
- Para hacer caché, por ejemplo lo podremos hacer en el método **index()** o método **show()** de nuestros controladores y haremos algo como lo siguiente:
- Documentación Oficial sobre caché en Laravel: **http://laravel.com/docs/5.0/cache**.

- Fichero de **caché (durante 20 segundos) en el método index() de FabricanteController**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

// Activamos uso de caché.
use Illuminate\Support\Facades\Cache;

// Necesitaremos el modelo Fabricante para ciertas tareas.
use App\Fabricante;

// Necesitamos la clase Response para crear la respuesta especial con la cabecera de localización en el método Store()
use Response;

class FabricanteController extends Controller {

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
        public function index()
        {
            // Devuelve todos los fabricantes en JSON.
            // return Fabricante::all();

            // Mejora en la respuesta.
            // Devolvemos explícitamente el código 200 http de datos encontrados.
            // Se puede poner como 404 cuando no se encuentra nada.
            //return response()->json(['datos'=>Fabricante::all()],200);

            // Activamos la caché de los resultados.
            //  Cache::remember('tabla', $minutes, function()
            $fabricantes=Cache::remember('fabricantes',20/60, function()
            {
                // Caché válida durante 20 segundos.
                return Fabricante::all();
            });

            // Con caché.
            return response()->json(['status'=>'ok','data'=>$fabricantes], 200);

            // Sin caché.
            //return response()->json(['status'=>'ok','data'=>Fabricante::all()], 200);
        }


/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */

// Pasamos como parámetro al método store todas las variables recibidas de tipo Request
```

```
// utilizando inyección de dependencias
// Para acceder a Request necesitamos asegurarnos que está cargado use Illuminate\Http\Request;
// Información sobre Request en: http://laravel.com/docs/5.0/requests
// Ejemplo de uso de Request:  $request->input('name');
public function store(Request $request)
{

// Primero comprobaremos si estamos recibiendo todos los campos.
if (!$request->input('nombre') || !$request->input('direccion') || !$request->input('telefono'))
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesable] Utiliza
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],422);
}

// Insertamos una fila en Fabricante con create pasándole todos los datos recibidos.
// En $request->all() tendremos todos los campos del formulario recibidos.
$nuevoFabricante=Fabricante::create($request->all());

// Más información sobre respuestas en http://jsonapi.org/format/
// Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado con un en
$response = Response::make(json_encode(['status'=>'ok','data'=>$nuevoFabricante]), 201)->header('Location', 'http://www.dominio.local
return $response;
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
//
// return "Se muestra Fabricante con id: $id";
// Buscamos un fabricante por el id.
$fabricante=Fabricante::find($id);

// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Si queremos mantener una tabla de códigos de error en nuestra aplicación  lo ideal sería enviar un mensaje de error como:
// codigo 1000 (código específico de error en nuestra app)
// código http a enviar 404 de recurso solicitado no existe.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

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

}



/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update(Request $request, $id)
{
// Comprobamos si el fabricante que nos están pasando existe o no.
$fabricante=Fabricante::find($id);

// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// Listado de campos recibidos teóricamente.
$nombre=$request->input('nombre');
```

```
$direccion=$request->input('direccion');
$telefono=$request->input('telefono');

// Necesitamos detectar si estamos recibiendo una petición PUT o PATCH.
// El método de la petición se sabe a través de $request->method();
if ($request->method() === 'PATCH')
{
// Creamos una bandera para controlar si se ha modificado algún dato en el método PATCH.
$bandera = false;

// Actualización parcial de campos.
if ($nombre != null && $nombre!='')
{
$fabricante->nombre = $nombre;
$bandera=true;
}

if ($direccion != null && $direccion!='')
{
$fabricante->direccion = $direccion;
$bandera=true;
}


if ($telefono != null && $telefono!='')
{
$fabricante->telefono = $telefono;
$bandera=true;
}

if ($bandera)
{
// Almacenamos en la base de datos el registro.
$avion->save();
return response()->json(['status'=>'ok','data'=>$fabricante], 200);
}
else
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 304 Not Modified ? [No Modificada] Usado cuando el cacheo
// Este código 304 no devuelve ningún body, así que si quisiéramos que se mostrara el mensaje usaríamos un código 200 en su lugar.
return response()->json(['errors'=>array(['code'=>304,'message'=>'No se ha modificado ningún dato de fabricante.'])],304);
}
}


// Si el método no es PATCH entonces es PUT y tendremos que actualizar todos los datos.
if (!$nombre || !$direccion || !$telefono)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesable] Utiliza
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],422);
}

$fabricante->nombre = $nombre;
$fabricante->direccion = $direccion;
$fabricante->telefono = $telefono;

// Almacenamos en la base de datos el registro.
$fabricante->save();
return response()->json(['status'=>'ok','data'=>$fabricante], 200);
}


/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
{
// Primero eliminaremos todos los aviones de un fabricante y luego el fabricante en si mismo.
// Comprobamos si el fabricante que nos están pasando existe o no.
$fabricante=Fabricante::find($id);
```

```
// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// El fabricante existe entonces buscamos todos los aviones asociados a ese fabricante.
$aviones = $fabricante->aviones; // Sin paréntesis obtenemos el array de todos los aviones.

// Comprobamos si tiene aviones ese fabricante.
if (sizeof($aviones) > 0)
{
// Devolveremos un código 409 Conflict - [Conflicto] Cuando hay algún conflicto al procesar una petición, por ejemplo en PATCH, POST
return response()->json(['code'=>409,'message'=>'Este fabricante posee vehiculos asociados y no puede ser eliminado.'],409);
}

// Procedemos por lo tanto a eliminar el fabricante.
$fabricante->delete();

// Se usa el código 204 No Content ? [Sin Contenido] Respuesta a una petición exitosa que no devuelve un body (como una petición DEL
// Este código 204 no devuelve body así que si queremos que se vea el mensaje tendríamos que usar un código de respuesta HTTP 200.
return response()->json(['code'=>204,'message'=>'Se ha eliminado el fabricante correctamente.'],204);

}
}
```

- Véase el siguiente ejemplo dónde se usa **caché (durante 20 segundos) en el método index() de FabricanteAvionController**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

// Necesita los dos modelos Fabricante y Avion
use App\Fabricante;
use App\Avion;

// Necesitamos la clase Response para crear la respuesta especial con la cabecera de localización en el método Store()
use Response;

// Activamos uso de caché.
use Illuminate\Support\Facades\Cache;

class FabricanteAvionController extends Controller {
// Configuramos en el constructor del controlador la autenticación usando el Middleware auth.basic,
// pero solamente para los métodos de crear, actualizar y borrar.
public function __construct()
{
$this->middleware('auth.basic',['only'=>['store','update','destroy']]);
}

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index($idFabricante)
{
// Devolverá todos los aviones.
//return "Mostrando los aviones del fabricante con Id $idFabricante";
$fabricante=Fabricante::find($idFabricante);

if (! $fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}
```

```
// Activamos la caché de los resultados.
// Como el closure necesita acceder a la variable $ fabricante tenemos que pasársela con use($fabricante)
// Para acceder a los modelos no haría falta puesto que son accesibles a nivel global dentro de la clase.
//  Cache::remember('tabla', $minutes, function()
$avionesFabri=Cache::remember('claveAviones',2, function() use ($fabricante)
{
// Caché válida durante 2 minutos.
return $fabricante->aviones()->get();
});

// Respuesta con caché:
return response()->json(['status'=>'ok','data'=>$avionesFabri],200);

// Respuesta sin caché:
//return response()->json(['status'=>'ok','data'=>$fabricante->aviones()->get()],200);
//return response()->json(['status'=>'ok','data'=>$fabricante->aviones],200);
}


/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store(Request $request,$idFabricante)
{
/* Necesitaremos el fabricante_id que lo recibimos en la ruta
                 #Serie (auto incremental)
                Modelo
                Longitud
                Capacidad
                Velocidad
                Alcance */

// Primero comprobaremos si estamos recibiendo todos los campos.
if ( !$request->input('modelo') || !$request->input('longitud') || !$request->input('capacidad') || !$request->input('velocidad') ||
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesable] Utiliza
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan datos necesarios para el proceso de alta.'])],422);
}

// Buscamos el Fabricante.
$fabricante= Fabricante::find($idFabricante);

// Si no existe el fabricante que le hemos pasado mostramos otro código de error de no encontrado.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// Si el fabricante existe entonces lo almacenamos.
// Insertamos una fila en Aviones con create pasándole todos los datos recibidos.
$nuevoAvion=$fabricante->aviones()->create($request->all());

// Más información sobre respuestas en http://jsonapi.org/format/
// Devolvemos el código HTTP 201 Created ? [Creada] Respuesta a un POST que resulta en una creación. Debería ser combinado con un en
$response = Response::make(json_encode(['data'=>$nuevoAvion]), 201)->header('Location', 'http://www.dominio.local/aviones/'.$nuevoAvi
return $response;
}


/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($idFabricante,$idAvion)
{
//
```

```
return "Se muestra avión $idAvion del fabricante $idFabricante";
}

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update(Request $request, $idFabricante, $idAvion)
{
// Comprobamos si el fabricante que nos están pasando existe o no.
$fabricante=Fabricante::find($idFabricante);

// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// El fabricante existe entonces buscamos el avion que queremos editar asociado a ese fabricante.
$avion = $fabricante->aviones()->find($idAvion);

// Si no existe ese avión devolvemos un error.
if (!$avion)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un avión con ese código asociado a ese fabricante.
}


// Listado de campos recibidos teóricamente.
$modelo=$request->input('modelo');
$longitud=$request->input('longitud');
$capacidad=$request->input('capacidad');
$velocidad=$request->input('velocidad');
$alcance=$request->input('alcance');

// Necesitamos detectar si estamos recibiendo una petición PUT o PATCH.
// El método de la petición se sabe a través de $request->method();
/*      Modelo          Longitud               Capacidad              Velocidad              Alcance */
if ($request->method() === 'PATCH')
{
// Creamos una bandera para controlar si se ha modificado algún dato en el método PATCH.
$bandera = false;

// Actualización parcial de campos.
if ($modelo != null && $modelo!='')
{
$avion->modelo = $modelo;
$bandera=true;
}

if ($longitud != null && $longitud!='')
{
$avion->longitud = $longitud;
$bandera=true;
}

if ($capacidad != null && $capacidad!='')
{
$avion->capacidad = $capacidad;
$bandera=true;
}

if ($velocidad != null && $velocidad!='')
{
$avion->velocidad = $velocidad;
$bandera=true;
}
```

```
if ($alcance != null && $alcance!='')
{
$avion->alcance = $alcance;
$bandera=true;
}

if ($bandera)
{
// Almacenamos en la base de datos el registro.
$avion->save();
return response()->json(['status'=>'ok','data'=>$avion], 200);
}
else
{
// Devolveremos un código 304 Not Modified ? [No Modificada] Usado cuando el cacheo de encabezados HTTP está activo
// Este código 304 no devuelve ningún body, así que si quisiéramos que se mostrara el mensaje usaríamos un código 200 en su lugar.
return response()->json(['code'=>304,'message'=>'No se ha modificado ningún dato de fabricante.'],304);
}

}

// Si el método no es PATCH entonces es PUT y tendremos que actualizar todos los datos.
if (!$modelo || !$longitud || !$capacidad || !$velocidad || !$alcance)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 422 Unprocessable Entity ? [Entidad improcesable] Utiliza
return response()->json(['errors'=>array(['code'=>422,'message'=>'Faltan valores para completar el procesamiento.'])],422);
}

$avion->modelo = $modelo;
$avion->longitud = $longitud;
$avion->capacidad = $capacidad;
$avion->velocidad = $velocidad;
$avion->alcance = $alcance;

// Almacenamos en la base de datos el registro.
$avion->save();

return response()->json(['data'=>$avion], 200);
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($idFabricante,$idAvion)
{
// Comprobamos si el fabricante que nos están pasando existe o no.
$fabricante=Fabricante::find($idFabricante);

// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

// El fabricante existe entonces buscamos el avion que queremos borrar asociado a ese fabricante.
$avion = $fabricante->aviones()->find($idAvion);

// Si no existe ese avión devolvemos un error.
if (!$avion)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un avión con ese código asociado a ese fabricante.
}

// Procedemos por lo tanto a eliminar el avión.
$avion->delete();
```

```
// Se usa el código 204 No Content ? [Sin Contenido] Respuesta a una petición exitosa que no devuelve un body (como una petición DEL
// Este código 204 no devuelve body así que si queremos que se vea el mensaje tendríamos que usar un código de respuesta HTTP 200.
return response()->json(['code'=>204,'message'=>'Se ha eliminado el avión correctamente.'],204);
}

}
```

Los middlewares en Laravel

Los **middleware HTTP** proporcionan un **mecanismo de filtrado de peticiones HTTP** que acceden a nuestra aplicación. Por ejemplo Laravel incluye un middleware que verifica si el usuario de nuestra aplicación está autenticado. Si el usuario no está autenticado, el middleware redirecciona al usuario a una página de autenticación. Si el usuario está autenticado, el middleware permitirá acceder al recurso solicitado en la aplicación.

En el framework Laravel se incluyen muchos middlewares como por ejemplo para mantenimiento, autenticación, protección CSRF, etc.

Todos los middleware se localizan en la carpeta **app**/**Http**/**Middleware**.

Para generar una plantilla de **middleware** se puede hacer con **PHP Artisan**:

```
php artisan make:middleware NombreMiddleware
```

En el fichero **app/Http/Kernel.php** se definen los **middlewares que se cargarán al inicio de nuestra aplicación**.

```
<?php namespace App\Http;

use Illuminate\Foundation\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel {

/**
         * The application's global HTTP middleware stack.
         *
         * @var array
         */
// Aquí se programa el Middleware que se cargará siempre y globalmente en la aplicación.
protected $middleware = [
'Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode',
'Illuminate\Cookie\Middleware\EncryptCookies',
'Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse',
'Illuminate\Session\Middleware\StartSession',
'Illuminate\View\Middleware\ShareErrorsFromSession',
'App\Http\Middleware\VerifyCsrfToken',
];

/**
         * The application's route middleware.
         *
         * @var array
         */

// Aquí se programa el Middleware que se cargará dependiendo de la ruta.
protected $routeMiddleware = [
'auth' => 'App\Http\Middleware\Authenticate',
'auth.basic' => 'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',
'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',
];

}
```

Desactivación de la protección CSRF del middleware de Laravel para la API RESTful

Se supone que cuando se producen las peticiones de tipo POST, PUT y DELETE se están enviando datos que proceden de un formulario. Pues bien, dichos formularios deberían tener algún tipo de protección contra CSRF.

Si intentamos hacer un envío POST con la extensión **API REST de Chrome o POSTMAN** a la dirección /**fabricantes** vemos que nos va a dar un error del tipo:

```
TokenMismatchException
```

Teóricamente se debería estar ejecutando el método **store()** de nuestro controlador de **fabricantes**, pero se supone que previamente se tendría que haber mostrado un formulario para cubrir esos datos y haber enviado junto con los datos el token CSRF, pero no se ha enviado dicho token por que ni siquiera tenemos formulario.

Para evitar este tipo de error en nuestra API RESTful podríamos comentar en **app**/**Http**/**Kernel.php** en el array $middleware la siguiente línea:

```
//'App\Http\Middleware\VerifyCsrfToken',
```

ATENCIÓN: Ésto no implicará ningún tipo de agujero de seguridad en nuestra API ya que más adelante veremos como evitar CSRF a través de autenticación de usuarios.

Autenticación básica con middleware en Laravel

- Crearemos un sistema muy básico de autenticación para validar a los usuarios que pueden insertar, actualizar o eliminar datos de nuestra aplicación usando la API RESTful.
- Para este tipo de autenticación en Laravel se utilizará un Middleware que nos proporciona el framework.
- **La idea sería tener una tabla de Usuarios y cada vez que se requiera algún tipo de operación de inserción, actualización o borrado se solicite mediante autenticación HTTP Básica las credenciales de acceso y se validen contra la base de datos de Usuarios**.

Creación del modelo User.php su migration y el seeder

Laravel incorpora un Modelo **app**/**User.php** que permite llevar el **control de usuarios de la aplicación**.

- Podemos **editar** el modelo **User.php** e indicar que campos necesitaremos para la gestión de usuarios.

- Contenido **original** del fichero **app**/**User.php**:

```
<?php namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;

class User extends Model implements AuthenticatableContract, CanResetPasswordContract {

use Authenticatable, CanResetPassword;

/**
         * The database table used by the model.
         *
         * @var string
         */
protected $table = 'users';

/**
         * The attributes that are mass assignable.
         *
         * @var array
         */
protected $fillable = ['name', 'email', 'password'];

/**
         * The attributes excluded from the model's JSON form.
         *
         * @var array
         */
protected $hidden = ['password', 'remember_token'];

}
```

- Contenido **editado** del fichero **app/User.php**:

```
<?php namespace App;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;

class User extends Model implements AuthenticatableContract, CanResetPasswordContract {

use Authenticatable, CanResetPassword;

/**
         * The database table used by the model.
         *
         * @var string
         */
protected $table = 'users';

/**
         * The attributes that are mass assignable.
         *
         * @var array
         */

// ELiminamos el campo name, nos vamos a autenticar solamente con email y password.
protected $fillable = ['email', 'password'];

/**
         * The attributes excluded from the model's JSON form.
         *
         * @var array
         */

// Eliminamos el campo remember_token
protected $hidden = ['password'];

}
```

- Crearemos la **migration** para la tabla de **Users**:

```
php artisan make:migration users --create=users
```

- Editamos la **migration de Users** para indicar los campos de la tabla MySQL:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class Users extends Migration {

/**
         * Run the migrations.
         *
         * @return void
         */
public function up()
{
Schema::create('users', function(Blueprint $table)
{
$table->increments('id');
$table->string('email')->unique();
$table->string('password');
$table->timestamps();
});
}

/**
         * Reverse the migrations.
```

```
         *
         * @return void
         */
public function down()
{
Schema::drop('users');
}

}
```

- **Ejecutamos** la migración:

```
php artisan migrate
```

- Creamos un **seeder** en **database/seeds/UserSeeder.php**:

```
<?php

use Illuminate\Database\Seeder;

// Hace uso del modelo de User.
use App\User;

class UserSeeder extends Seeder {

/**
         * Run the database seeds.
         *
         * @return void
         */
public function run()
{
User::create(
[
'email'=>'test@test.es',
'password'=> Hash::make('abc123')//Cifrado de la contraseña abc123
]);

}
}
```

- Edición de **database/seeds/DatabaseSeeder.php**:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

// Hacemos uso del modelo User.
use App\User;

class DatabaseSeeder extends Seeder {

/**
         * Run the database seeds.
         *
         * @return void
         */
public function run()
{
Model::unguard();

$this->call('FabricanteSeeder');
$this->call('AvionSeeder');

// Solo queremos un único usuario en la tabla, así que truncamos primero la tabla
// Para luego rellenarla con los registros.
User::truncate();

// LLamamos al seeder de Users.
$this->call('UserSeeder');
}
```

```
}
```

- Ejecutamos el seeder:

```
# Ejecutar el comando para evitar el error [ReflectionException] Class UserSeeder does not exist:
composer dump-autoload

# Ejecutamos los seeders:
php artisan db:seed

# Si queremos dejar la base de datos en su estado inicial y que ejecute 1 seed haremos:
php artisan migrate:refresh --seed
```

Configuración de la autenticación

La autenticación de usuarios solamente la vamos a utilizar para el caso de crear, actualizar o eliminar datos a través de la API REST.

- Para poder usar la autenticación usaremos el **middleware AuthenticateWithBasicAuth** de Laravel.
- Editaremos los controladores para indicarles que hagan uso de ese Middleware pero solamente para unos métodos específicos y además borraremos los métodos de /**create** y /**edit** ya que no usamos formularios.

- **Editaremos** el fichero del controlador **app/Http/Controllers/FabricanteController.php**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

// Necesitaremos el modelo Fabricante para ciertas tareas.
use App\Fabricante;

class FabricanteController extends Controller {

// Configuramos en el constructor del controlador la autenticación usando el Middleware auth.basic,
// pero solamente para los métodos de crear, actualizar y borrar.
public function __construct()
{
$this->middleware('auth.basic',['only'=>['store','update','destroy']]);
}

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index()
{
// Devolverá todos los fabricantes.
// return "Mostrando todos los fabricantes de la base de datos.";
// return Fabricante::all();  No es lo más correcto por que se devolverían todos los registros. Se recomienda usar Filtros.
// Se debería devolver un objeto con una propiedad como mínimo data y el array de resultados en esa propiedad.
// A su vez también es necesario devolver el código HTTP de la respuesta.
//php http://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-RESTful-pragmatica/
// https://cloud.google.com/storage/docs/json_api/v1/status-codes

return response()->json(['status'=>'ok','data'=>Fabricante::all()], 200);
}


/**
         * Store a newly created resource in storage.
         *
         * @return Response
```

```
         */
public function store()
{
//
return "petición post recibida.";
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
//
// return "Se muestra Fabricante con id: $id";
// Buscamos un fabricante por el id.
$fabricante=Fabricante::find($id);

// Si no existe ese fabricante devolvemos un error.
if (!$fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

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

}



/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($id)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
{
//
}

}
```

- Si ahora intentamos acceder a aviones por alguno de los métodos de **POST, PUT o DELETE de Fabricante nos pedirá la autenticación**:

- El controlador para **app/Http/Controllers/AvionController.php** no es necesario modificarlo por que no tenemos ninguno de esos métodos:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

// Necesitaremos el modelo Avion para ciertas tareas.
use App\Avion;
```

```
class AvionController extends Controller {
/**
         * Display a listing of the resource.
         *
         * @return Response
         */

public function index()
{
// Devolverá todos los fabricantes.
// return "Mostrando todos los fabricantes de la base de datos.";
// return Fabricante::all();  No es lo más correcto por que se devolverían todos los registros. Se recomienda usar Filtros.
// Se debería devolver un objeto con una propiedad como mínimo data y el array de resultados en esa propiedad.
// A su vez también es necesario devolver el código HTTP de la respuesta.
//php http://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-RESTful-pragmatica/
// https://cloud.google.com/storage/docs/json_api/v1/status-codes

return response()->json(['status'=>'ok','data'=>Avion::all()], 200);
}


/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store()
{
//
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($id)
{
//
// return "Se muestra Fabricante con id: $id";
// Buscamos un fabricante por el id.
$avion=Avion::find($id);

// Si no existe ese avion devolvemos un error.
if (!$avion)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un avión con ese código.'])],404);
}

return response()->json(['status'=>'ok','data'=>$avion],200);
}

/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($id)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($id)
```

```
{
//
}

}
```

- **Editaremos** el controlador para **app/Http/Controllers/FabricanteAvionController.php**:

```
<?php namespace App\Http\Controllers;

use App\Http\Requests;
use App\Http\Controllers\Controller;

use Illuminate\Http\Request;

// Necesita los dos modelos Fabricante y Avion
use App\Fabricante;
use App\Avion;


class FabricanteAvionController extends Controller {
// Configuramos en el constructor del controlador la autenticación usando el Middleware auth.basic,
// pero solamente para los métodos de crear, actualizar y borrar.
public function __construct()
{
$this->middleware('auth.basic',['only'=>['store','update','destroy']]);
}

/**
         * Display a listing of the resource.
         *
         * @return Response
         */
public function index($idFabricante)
{
// Devolverá todos los aviones.
//return "Mostrando los aviones del fabricante con Id $idFabricante";
$fabricante=Fabricante::find($idFabricante);

if (! $fabricante)
{
// Se devuelve un array errors con los errores encontrados y cabecera HTTP 404.
// En code podríamos indicar un código de error personalizado de nuestra aplicación si lo deseamos.
return response()->json(['errors'=>array(['code'=>404,'message'=>'No se encuentra un fabricante con ese código.'])],404);
}

return response()->json(['status'=>'ok','data'=>$fabricante->aviones()->get()],200);
//return response()->json(['status'=>'ok','data'=>$fabricante->aviones],200);
}


/**
         * Store a newly created resource in storage.
         *
         * @return Response
         */
public function store()
{
//
}

/**
         * Display the specified resource.
         *
         * @param  int  $id
         * @return Response
         */
public function show($idFabricante,$idAvion)
{
//
return "Se muestra avión $idAvion del fabricante $idFabricante";
}
```

```
/**
         * Update the specified resource in storage.
         *
         * @param  int  $id
         * @return Response
         */
public function update($idFabricante,$idAvion)
{
//
}

/**
         * Remove the specified resource from storage.
         *
         * @param  int  $id
         * @return Response
         */
public function destroy($idFabricante,$idAvion)
{
//
}

}
```

Control CSRF en la API

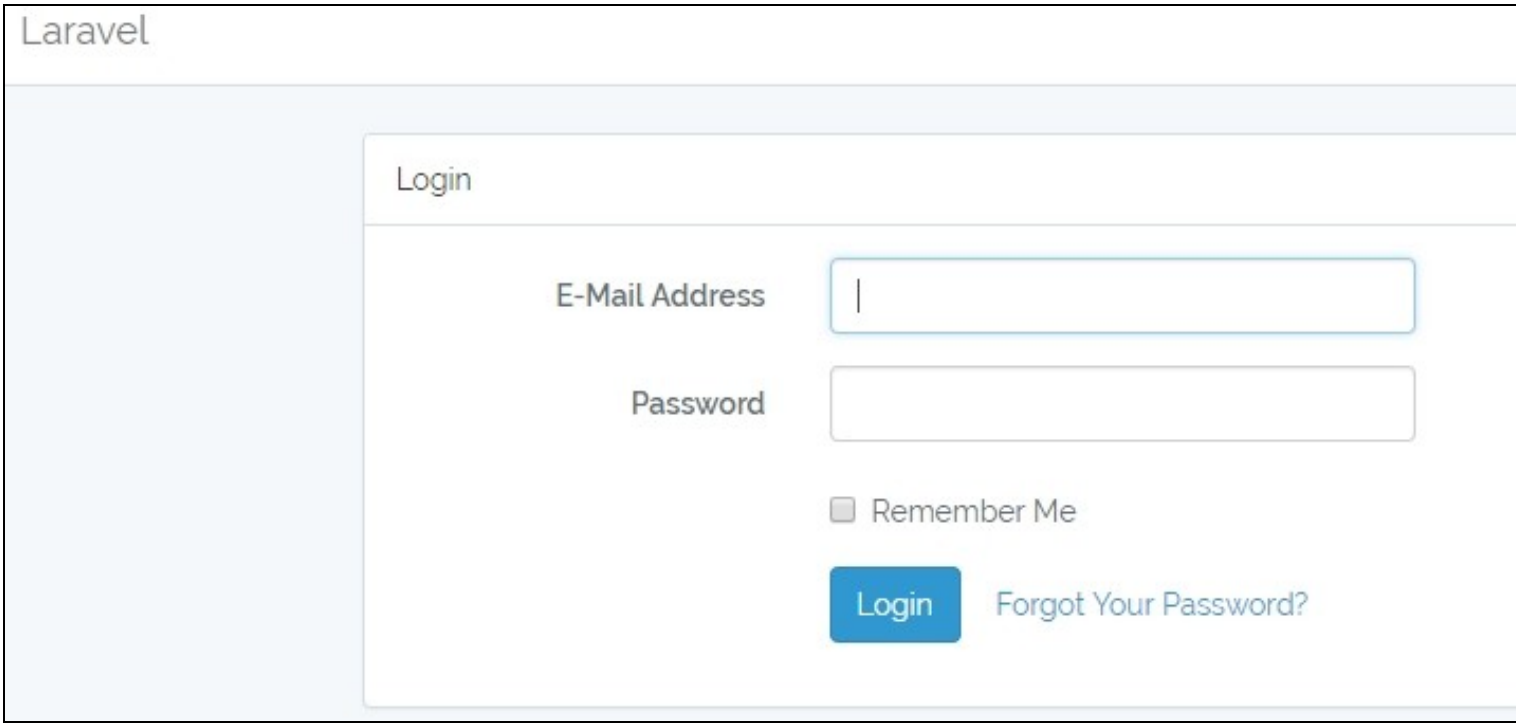
- Cuando usamos la API desde **API REST Client** o **Postman**, vemos que en este momento nos pide autenticación para poder insertar o actualizar datos.
- Si hacemos un formulario nuevo en otro dominio, podríamos (una vez autenticados en alguna de las extensiones anteriores y desde el mismo navegador) seguir insertando o actualizando datos, con lo que vemos que no se está haciendo **control CSRF** (puesto que lo hemos desactivado anteriormente).
- Para dar más seguridad podríamos en principio forzar a que vuelva a solicitar autenticación y que no use la cookie que está en el navegador (de autenticaciones previas), es lo que se conoce como **Autenticación Stateless**.
- Es decir se autentica hace la operación y cierra la sesión destruyendo el estado de autenticación previo.
- Para ello tenemos que editar el fichero **.env** y en la sección **SESSION_DRIVER=file** modificarlo a **SESSION_DRIVER=array**.
- Al indicar que utilice **array Laravel va a mantener ese array en memoria y no mantendrá esa sesión entre una petición y otra**. Se suele usar para hacer test unitarios.
- Si usamos file o una base de datos para la gestión de sesiones estamos manteniendo el estado de sesión entre una petición y otra.
- Información sobre las sesiones: **https://laravel.com/docs/7.x/session**

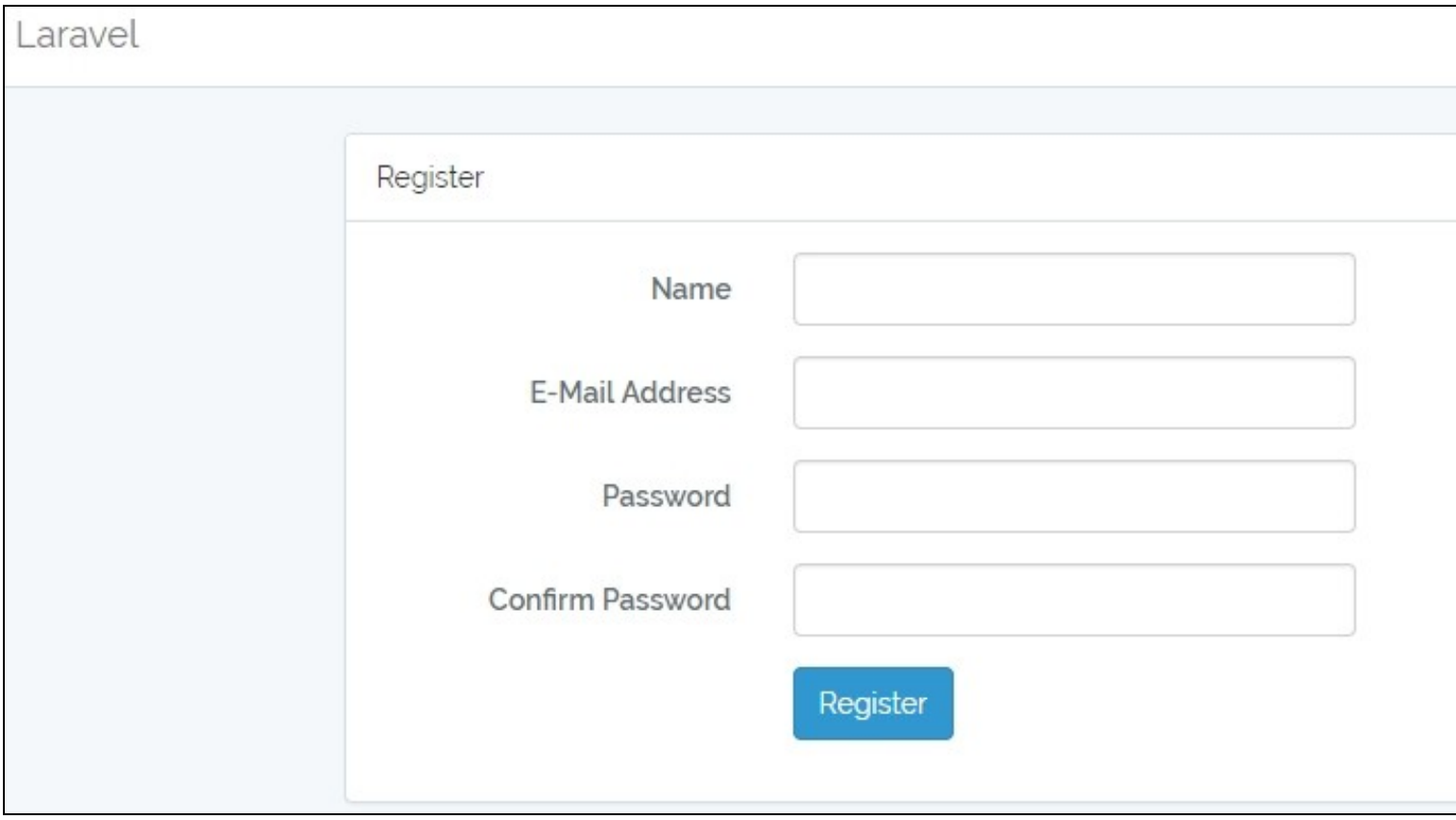
Autenticación de Usuarios en Laravel

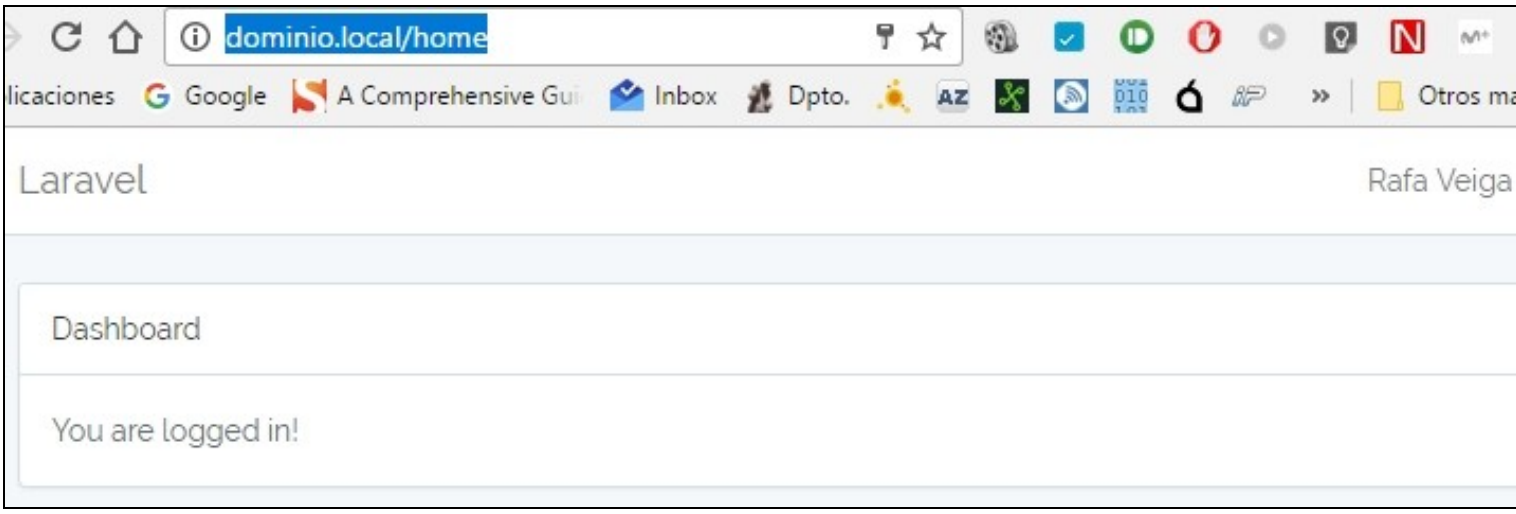
- Vamos a ver como podemos configurar la autenticación para que solamente los usuarios autenticados puedan usar nuestra API REST.
- La **documentación oficial sobre Autenticación** en Laravel: **https://laravel.com/docs/7.x/authentication**
- Laravel hace que la implementación de la autenticación sea muy simple. De hecho, casi todo está prácticamente configurado. El archivo de configuración de autenticación se encuentra en **config/auth.php**, que contiene varias opciones para ajustar el comportamiento de los servicios de autenticación.
- En su núcleo, las instalaciones de autentificación de Laravel están formadas por "guards" y "providers". Los "guards" definen la forma en que los usuarios se autentican para cada solicitud. Por ejemplo, Laravel cuenta con un "guard" de sesión que mantiene el estado utilizando el almacenamiento de sesiones y las cookies.
- Los "providers" definen cómo se recuperan los usuarios de su almacenamiento persistente. Laravel cuenta con soporte para recuperar usuarios mediante Eloquent y el constructor de consultas de base de datos. Sin embargo, puede definir "providers" adicionales según sea necesario para su aplicación.
- Laravel incluye un modelo para gestionar los usuarios a través de Eloquent. Se encuentra en **App\User**

```
# Para crear las rutas de autenticación y las vistas necesarias para la autenticación utilizamos el siguiente comando:
php artisan make:auth

# Ejecutamos las migraciones para que se generen las tablas en la base de datos:
php artisan migrate

# A partir de aquí dispondremos de las siguientes rutas: /home /login /register /password/reset /logout
```

Ejemplos de la **interfaz totalmente programada para gestionar el acceso a la aplicación**.







Bloquear el acceso a la API solamente para usuarios registrados

- Una vez que ya tenemos la interfaz funcionando y las tablas creadas, vamos a ver como podemos bloquear el acceso a los recursos de la API solamente a usuarios que se hayan logueado previamente.
- Lo podremos hacer de varias formas:
- A **nivel de ruta**

Por ejemplo si tenemos una ruta sencilla podríamos bloquear el acceso a dicha ruta a través del middleware auth:

```
# Al acceder a /rutasencilla ejecutará el middleware auth para comprobar si el usuario está logueado.
Route::get('/rutasencilla', 'FabricanteController@index')->middleware('auth');

# Si se trata de una ruta de tipo resource, debido a que una ruta resource no devuelve nada, sino que simplemente
# registra un montón de rutas, tendremos que hacerlo de la siguiente forma:

Route::group(['middleware' => ['web', 'auth']], function () {
        Route::resource('fabricantes','FabricanteController');
});
```

- Dentro del propio **controlador** (para todas las rutas que estén programadas dentro del controlador:

```
# También podríamos hacerlo dentro del controlador en el constructor, en lugar de programarlo en la ruta:
public function __construct()
{
$this->middleware('auth');
}
```

- Dentro de un **método de un controlador** (para bloquear solamente rutas específicas dentro de un controlador):

```
# Si quisiéramos solamente proteger algún método dentro de nuestro controlador, por ejemplo el método store de FabricanteController.
# Tenemos que usar la clase Auth al principio del controlador.

...
use Auth;
...

public function store()
{
if (Auth::check())
{
                // Código para usuarios logueados
}
else
return response('Unauthorized.', 401);
}
```

JSON Web Tokens en Laravel

https://github.com/tymondesigns/jwt-auth/wiki

API RATE Limits

https://mattstauffer.co/blog/api-rate-limiting-in-laravel-5-2

Validación de Usuarios con OAuth2

Introducción a OAuth2

Instalación de Servidor OAuth2 en Laravel

- Instalaremos el siguiente componente para crear un servidor OAuth2 en Laravel.
- Instrucciones aquí: https://github.com/lucadegasperi/oauth2-server-laravel
- Documentación de instalación: https://github.com/lucadegasperi/oauth2-server-laravel/wiki

Middleware para validar con OAuth2

Usaremos un middleware creado para el servidor OAuth2 anteriormente citado:

- https://github.com/tpavlek/oauth2-server-laravel
- Código del middleware en: https://github.com/tpavlek/oauth2-server-laravel/blob/l5/src/Middleware/OAuthMiddleware.php

Pruebas con OAuth2

Uso de Git para control de Versiones

- Control de versiones con Git y GitHub - Guía Rápida
- Trabajo Colaborativo con Git y GitHub

Guía de Diseño de APIs HTTP

- https://github.com/jmnavarro/http-api-design

Veiga (discusión) 23:06 4 mar 2020 (CET)