Tu primera aplicación en 7 pasos

Ahora que ya tenemos nuestro entorno de desarrollo preparado, crearemos una aplicación web con Laravel paso por paso. Al finalizar los 7 pasos que encontrarás en este capítulo, obtendremos como resultado una revista online a la que hemos llamado RevistApp. Esta aplicación mostrará los artículos de una revista almacenados en una base de datos.md-button.

¿Comenzamos?

Paso 1 - Crea tu primer proyecto

Este paso **no es necesario realizarlo si has optado por utilizar Laravel Sail**, ya que Sail creará automáticamente una nueva aplicación junto con el entorno.

Si has decidido utilizar Laravel Homestead o Vagrant como tu entorno de desarrollo, entonces deberás realizar las siguientes acciones:

Crea la aplicación dentro de la máquina virtual

Accede a tu máquina virtual utilizando el comando `vagrant ssh` y posicionate en la carpeta donde almacenarás tus proyectos (la carpeta que has indicado en el archivo de configuración `Homestead.yaml`, por ejemplo: `/home/vagrant/proyects`). Ejecuta el siguiente comando para crear un nuevo proyecto:

```
composer create-project laravel/laravel revistapp
```

Si recibes un error, probablemente sea porque todavía no tienes Composer instalado en tu máquina virtual. Para ello, ejecuta el siguiente comando:

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

Una vez instalado vuelve a ejecutar el comando `create-project` de Composer. Este comando inicializará un nuevo proyecto creando en el directorio `revistapp`. Puedes acceder a la aplicación entrando a http://aplicacion1.test (o el dominio que hayas indicado en la configuración) desde tu navegador favorito.

De forma alternativa también puedes utilizar el comando `laravel new` que también creará un nuevo proyecto de Laravel en la carpeta especificada:

```
laravel new revistapp
```

Puedes entrar a la nueva carpeta del proyecto creada, `revistapp`, para ver los archivos que se han generada. A partir de ahora siempre trabajaremos dentro de este directorio.

Ten en cuenta que en este caso, al haber creado una aplicación llamada `revistapp`, deberás tener una entrada en tu archivo de configuración `Homestead.yaml` que referencia a la carpeta `/public` del proyecto recién creado:

```
sites:
    - map: articulos.test
      to: /home/vagrant/code/revistapp/public
```

Generar la clave de la aplicación (Application Key)

Laravel utiliza una clave para securizar tu aplicación. La clave de aplicación es un string de 32 caracteres utilizado para encriptar datos como la sesión de usuario. Cuando se instala Laravel utilizando Composer o el instalador de Laravel, la clave se genera automáticamente, por lo que no es necesario hacer nada. Comprueba que existe un valor para `APP_KEY` en el fichero de configuración `.env`. En caso de no tener una clave generada, créala utilizando el siguiente comando:

```
php artisan key:generate
```

Establecer los permisos de directorio

Homestead realiza este paso por nosotros, por lo que si estás utilizando Homestead los permisos deberían estar correctamente establecidos. Si no estás utilizando Homestead o quieres desplegar tu aplicación en un servidor, no olvides establecer permisos de escritura para el servidor web en los directorios `storage` y `bootstrap/cache`.

Hands on! (1/7)

Si no lo has hecho ya, crea una aplicación llamada `revistapp`. Esta será la aplicación que iremos creando paso a paso.

Paso 2 - Crear un Router

Las rutas son los puntos de entrada a nuestra aplicación. Cada vez que un usuario hace una petición a una de las rutas de la aplicación, Laravel trata la petición mediante un Router definido en el directorio `routes`, el cual será el encargado de direccionar la petición a un Controlador. Las rutas accesibles para navegadores estarán definidas en el archivo `routes/web.php` y aquellas accesibles para servicios web (webservices) estarán definidas en el archivo `routes/api.php`. A continuación se muestra un ejemplo:

```
<?php

Route::get('/articulos', function () {
    return '¡Vamos a leer unos articulos!';
});
```

El código anterior muestra cómo se define una ruta básica. En este caso, cuando el usuario realice una petición sobre `/articulos`, nuestra aplicación ejecutará la función anónima definida. En este caso enviará una respuesta al usuario con el string '¡Vamos a leer unos articulos!'.

Podemos especificar tantas rutas como queramos:

```
<?php

Route::get('/articulos', function () {
    return '¡Vamos a leer unos articulos!';
});

Route::get('/usuarios', function () {
    return 'Hay muchos usuarios en esta aplicación';
});
```

También es posible responder a peticiones de tipo POST, por ejemplo para recibir datos de formularios:

```
<?php

Route::get('/articulos', function () {
    return '¡Vamos a leer unos articulos!';
});

Route::post('/articulos', function () {
    return '¡Vamos a insertar un artículo!';
});
```

Aparte de ejecutar las acciones definidas para cada ruta, Laravel ejecutará el middlewere específico en función del Router utilizado (por ejemplo, el middlewere relacionado con las peticiones web proveerá de funcionalidades como el estado de la sesión o la protección CSRF).

Ver las rutas creadas

La utilidad Artisan de Laravel nos provee de comandos muy útiles (por ejemplo, para crear controladores o migraciones). Disponemos de un comando concreto para mostrar todas las rutas de una aplicación de forma rápida. Basta con ejecutar el siguiente comando en la consola:

```
php artisan route:list
```

Recuerda que si estás utilizando Laravel Sail, puedes ejecutar el comando directamente los comandos de artisan incluyendo `sail` al princicio:

```
sail artisan route:list
```

Si quieres que Laravel no muestre las rutas creadas por paquetes de terceros (vendor) puedes añadir el flag `except-vendor` al final:

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

Devolver un JSON

También es posible devolver un JSON. Laravel convertirá automáticamente cualquier array a JSON:

```php
<?php

Route::get('/articulos', function () {
    $articulos = [
        [
            "id" => 1,
            "titulo" => "Primer artículo..."
        ],
        [
            "id" => 1,
            "titulo" => "Segundo artículo..."
        ]
    ];
    return $articulos;
});
```

Parámetros en la ruta

Una URL puede contener información de nuestro interés. Laravel permite acceder a esta información de forma sencilla utilizando los parámetros de ruta:

```php
<?php

Route::get('articulos/{id}', function ($id) {
    return 'Vas a leer el artículo: '.$id;
});
```

Los parámetros de ruta vienen definidos entre llaves `{}` y se inyectan automáticamente en las callbacks. Es posible utilizar más de un parámetro de ruta:

```php
<?php

Route::get('articulos/{id}/usuario/{name}', function ($id, $name) {
    return 'Vas a leer el artículo: '.$id. ' del usuario' .$name;
});
```

Para indicar un parámetro como opcional, le tenemos que añadir el símbolo `?` al final del parámentro. Le tendremos que añadir un valor por defecto para asegurarnos su correcto funcionamiento:

```php
<?php

Route::get('articulos/{id?}', function ($id = 0) {
    return 'Vas a leer el artículo: '.$id;
});
```

Acceder a la información de la petición

También es posible acceder a la información enviada en la petición mediante el método `request()`. Por ejemplo, el siguiente código devolverá el valor enviado para el parámetro 'fecha' de la URL `/articulos?fecha=hoy`:

```php
<?php

Route::get('/articulos', function () {
    $fecha = request('fecha');
    return $fecha;
});
```

Rutas con nombre

Es posible asignar nombres a las rutas que sirvan para referirnos a ellas. De esta forma, en caso de que la URL de una ruta cambie, únicamente tendremos que cambiarlo en el router y no en todos los ficheros HTML donde estemos enlazando a dicha ruta.

Para especificar el nombre a una ruta debemos utilizar la función `name()`, la cual deberá recibir como parámetro el nombre que se desea asignar a la ruta:

```php
<?php

Route::get('/articulos', function () {
    return "Ruta con nombre!";
})->name('articulos.index');
```

En un futuro veremos cómo generar las URLs a partir de su nombre. Por ejemplo, en lugar de utilizar:

```
<a href="/articulos">Ver artículos</a>
```

utilizaremos:

```
<a href="{{ route('articulos.index') }}">Ver artículos</a>
```

Hands on! (2/7)

Añade a tu aplicación `revistapp` dos nuevas rutas.

- `articulos/` : Devolverá un array de artículos en formato JSON. Asigna el nombre `articulos.index` a la ruta utilizando la función `name()` .
- `articulos/{id}` : Devolverá la siguiente frase: **"Gracias por leer el artículo con id: {id}"**. Asigna el nombre `articulos.show` a la ruta utilizando la función `name()` .

Solución

```
<?php

Route::get('/articulos', function () {
    $articulos = [
        ["id" => 1, "titulo" => "Primer artículo..."],
        ["id" => 2, "titulo" => "Segundo artículo..."],
        ["id" => 3, "titulo" => "Tercer artículo..."],
    ];
    return $articulos;
})->name('articulos.index');

Route::get('articulos/{id}', function ($id) {
    $frase = "Gracias por leer el artículo con id: " . $id;
    return $frase;
})->name('articulos.show');
```

Paso 3 - Crear una vista

Definiendo una vista sencilla

Las vistas son plantillas que contienen el HTML que enviará nuestra aplicación a los usuarios. Se almacenan en el directorio `/resources/views` de nuestro proyecto.

```
<!-- vista almacenada en /resources/views/articulos.blade.php -->
<html>
    <body>
        <h1>¡Vamos a leer unos artículos!</h1>
    </body>
</html>
```

Tendrán la extensión `.blade.php` ya que Laravel utiliza el motor de plantillas Blade, como veremos más adelante.

Devolver una vista

Cargar y devolver una vista al usuario es tan sencillo como utilizar la función global (helper) `view()`:

```
<?php

Route::get('/articulos', function () {
    return view('articulos');
})->name('articulos');
```

Al indicarle el nombre de la vista como parámetro no es necesario indicar la ruta completa de la vista ni la extensión `.blade.php`. Laravel asume que las vistas estarán en la carpeta `/resources/views` y que tendrán la extensión `.blade.php`.

Acceder a datos desde la vista

Laravel utiliza el motor de plantillas Blade por defecto. Un motor de plantillas permite crear vistas empleando código HTML junto con código específico del motor empleado. De esta forma podremos mostrar información almacenada en variables, crear condiciones if/else, estructuras repetitivas, etc.

En Blade mostrar datos almacenados en variables es muy sencillo:

```
<?php $nombre = "Nora"; ?>
<html>
    <body>
    <h1>Vamos a leer al escritor {{ $nombre }}</h1>
        </ul>
    </body>
</html>
```

Tal y como se puede ver en el ejemplo anterior, basta con escribir el nombre de la variable entre llaves `{{ }}`. Es una buena práctica evitar mezclar el código PHP con nuestras vistas, por lo que toda la información que necesitemos en las vistas la ubicaremos fuera de ellas. Existen distintas formas de pasarle variables a las vistas:

La primera opción sería utilizando el método `with()`, pasándole como parámetros el nombre de la variable y su valor:

```
<?php

Route::get('/', function () {
    $nombre = "Nora";
    return view('saludo')->with('nombre', $nombre);
});
```

Otra forma sería enviándolo como array:

```
<?php

Route::get('/', function () {
    $nombre = "Nora";
    return view('saludo')->with(['nombre' => $nombre]);
});
```

También podríamos pasar el array como segundo parámetro de la función `view()` y no utilizar `with()`:

```
<?php

Route::get('/', function () {
    $nombre = "Nora";
    return view('saludo',['nombre' => $nombre]);
});
```

Blade permite iterar por los datos de una colección o array. El siguiente ejemplo muestra cómo iterar por un array de strings de forma rápida:

```
<!-- Vista almacenada en resources/views/articulos.blade.php -->
<html>
    <body>
    <h1>Vamos a leer al escritor {{ $nombre }}</h1>
        <h2>Estos son sus últimos artículos:</h2>
        <ul>
            @foreach ($articulos as $articulo)
                <li>{{ $articulo }}</li>
            @endforeach
        </ul>
    </body>
</html>
```

En el caso anterior, la vista muestra el valor de la variable `nombre` e itera por el array `articulos`, por lo que será necesario proporcionarle dichos datos en la llamada al método `view()`:

```
<?php

Route::get('/articulos', function () {
    $articulos = array('Primero', 'Segundo','Tercero', 'Último');
    return view('articulos', [
        'nombre' => 'Ane Aranceta',
        'articulos' => $articulos
    ]);
})->name('articulos');
```

El motor de plantillas Blade permite el uso de todo tipo de estructuras:

```
@for ($i = 0; $i < 10; $i++)
    El valor actual es {{ $i }}
@endfor

@foreach ($users as $user)
    <p>El usuario: {{ $user->id }}</p>
@endforeach

@forelse($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>Eso es un bucle infinito.</p>
@endwhile

@if (count($articulos) === 1)
    Hay un artículo.
@elseif (count($articulos) > 1)
    Hay varios artículos.
@else
    No hay ninguno.
@endif

@unless (Auth::check())
    No estas autenticado.
@endunless
```

Puedes encontrar toda la información acerca de Blade en la documentación oficial.

Hands on! (3/7)

Actualiza las rutas de tu aplicación para que comiencen a devolver vistas al usuario:

- `/articulos`: Devolverá una vista que muestre los artículos en una tabla. La primera columna tendrá un enlace a la ruta del artículo, utilizando su `id`. La segunda columna contendrá el título del artículo.
- `articulos/{id}`: Devolverá una vista que contenga un párrafo con la siguiente frase: **"Gracias por leer el artículo con id {id}"**. También tendrá un enlace para voler a cargar ruta que muestra todos los arículos.

Solución

`/resources/views/articulos/index.blade.php`:

```
<html>
<head>
    <title>RevistApp</title>
</head>
<body>
    <h1>Revistapp</h1>
    <h2>Listado artículos:</h2>
    <table>
        <tr><th>Enlace</th><th>Título</th></tr>
        @foreach ($articulos as $articulo)
        <tr>
            <td><a href="{{ route('articulos.show', $articulo['id'])
}}">Ver</a></td>
            <td>{{ ($articulo['titulo']) }}</td>
        </tr>
        @endforeach
    </ul>
</body>
</html>
```

`/resources/views/articulos/show.blade.php`:

```
<html>
<head>
    <title>RevistApp</title>
</head>
<body>
    <h1>Revistapp</h1>
    <h2>Detalle del artículo:</h2>
    <p>Gracias por leer el artículo con id: {{ $id }}</p>
    <a href="{{ route('articulos.index') }}">Volver</a>
</body>
</html>
```

`routes/web.php`:

```php
<?php

Route::get('/articulos', function () {
    $articulos = [
        ["id" => 1, "titulo" => "Primer artículo..."],
        ["id" => 2, "titulo" => "Segundo artículo..."],
        ["id" => 3, "titulo" => "Tercer artículo..."],
    ];
    return view('articulos.index', [
        'articulos' => $articulos
    ]);
})->name('articulos.index');

Route::get('articulos/{id}', function ($id) {
    return view('articulos.show', [
        'id' => $id
    ]);
})->name('articulos.show');
```

Paso 4 - Crear un Controlador

Los controladores **contienen la lógica** para atender las peticiones recibidas. En otras palabras, un Controlador es una clase que agrupa el comportamiento de todas las peticiones **relacionadas con una misma entidad**. Por ejemplo, el controlador `ArticuloController` será el encargado de definir el comportamiento de acciones como: creación de un artículo, modificación de un artículo, búsqueda de artículos, etc.

Creando un Controller

Existen dos formas de crear un controlador:

- Crear manualmente una clase que extienda de la clase `Controller` de Laravel dentro del directorio `app/Http/Controllers`.
- Utilizar el comando de Artisan `make:controller`. Artisan es una herramienta que nos provee Laravel para interactuar con la aplicación y ejecutar instrucciones.

En este caso escogeremos la segunda opción y ejecutaremos el siguiente comando:

```
php artisan make:controller ArticuloController
# recuerda que si estás utilizado Sail el comando será: sail artisan ...
```

De este modo Laravel creará automáticamente un controlador vacío, que vendrá a ser una subclase de la clase `Controller`:.

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ArticuloController extends Controller
{
    //
}
```

Como hemos comentado antes los **controladores son los responsables de procesar las peticiones entrantes y devolver al cliente la respuesta**. Es decir, el router únicamente tendrá que invocar al controlador correspondiente para que atienda la petición entrante.

En el siguiente ejemplo se muestra cómo añadirle métodos que devuelvan vistas (como se puede ver en el caso de la función de nombre `show()`). Tal y como se puede apreciar, moveremos la lógica de la aplicación del router al controlador.

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\User;

class ArticuloController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return View
     */
    public function show($id)
    {
        return view('articulos.show', [
            'id' => $id
        ]);
    }
}
```

Es posible añadir más opciones al comando `make:controller`, aunque el único obligatorio es el nombre del controlador. por ejemplo el flag `--resource` al comando anterior, Artisan incluirá en el controlador creado los siete métodos REST más comunes: `index()`, `create()`, `store()`, `show()`, `edit()`, `update()`, `destroy()`:

```
php artisan make:controller ArticuloController --resource
```

Cada método tiene su función:

Tipo	URI	Método	Acción
GET	/articulos	index	Muestra todos los artículos
GET	/articulos/create	create	Muestra el formulario para crear un artículo
POST	/articulos	store	Guarda un nuevo artículo a partir de la información recibida
GET	/articulos/{id}	show	Muestra la información de un artículo específico
GET	/articulos/{id}/edit	edit	Muestra el formulario de edición de un artículo que ya existe
PUT/PATCH	/articulos/{id}	update	Guardar los cambios del artículo indicado a partir de la información recibida
DELETE	/articulos/{id}	destroy	Elimina el artículo con el ID indicado

Enrutar el Controlador

El siguiente paso es incluir en el Router las llamadas a los métodos del Controlador. En este caso crearemos las siguientes como ejemplo:

```php
<?php

use App\Http\Controllers\ArticuloController;

Route::get('articulos/', [ArticuloController::class, 'index'])-
>name('articulos.index');
Route::get('articulos/{id}', [ArticuloController::class, 'show'])-
>name('articulos.show');
Route::get('articulos/create', [ArticuloController::class, 'create'])-
>name('articulos.create');
Route::post('articulos/', [ArticuloController::class, 'store'])-
>name('articulos.store');
```

De esta forma direccionaremos las peticiones a los métodos de los controladores. Recuerda que el router no debe incluir ninguna lógica de la aplicación, únicamente redireccionar las peticiones a los controladores.

Existe también otra forma más rápida para generar automáticamente las rutas a todos los métodos de nuestro controlador:

```php
Route::resource('articulos', ArticuloController::class);
```

Si ejecutamos el comando `php artisan route:list` podemos comprobar cómo ya disponemos de todas las rutas a nuestro recurso y que cada una apunta al método correspondiente en el controlador.

Esta opción de `Route::resource` también ofrece la posibilidad de generar únicamente las rutas que le indiquemos. El siguiente ejemplo muestra como generar únicamene las rutas index y create utilizando el método `only()`:

```php
<?php

Route::resource('articulos', ArticuloController::class)->only([
    'index', 'create'
]);
```

Hands on! (4/7)

Crea un controlador llamado `ArticuloController.php` y mueve la lógica de las dos rutas del router `articulos/` y `articulos/{id}` al nuevo controlador.

Solución

`/App/Http/Controllers/ArticuloController.php`:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;

class ArticuloController extends Controller
{

    public function index()
    {
        $articulos = [
            ["id" => 1, "titulo" => "Primer artículo..."],
            ["id" => 2, "titulo" => "Segundo artículo..."],
            ["id" => 3, "titulo" => "Tercer artículo..."],
        ];
        return view('articulos.index', [
            'articulos' => $articulos
        ]);
    }

    public function show($id)
    {
        return view('articulos.show', [
            'id' => $id
        ]);
    }
}
```

El router pasará a indicar el controlador y el método que se encargará de cada petición:

```
<?php

use App\Http\Controllers\ArticuloController;

Route::get('articulos', [ArticuloController::class, 'index'])-
>name('articulos.index');
Route::get('articulos/{id}', [ArticuloController::class, 'show'])-
>name('articulos.show');
```

Paso 5 - Configurar la base de datos

Es muy difícil de imaginar una aplicación web que no haga uso de una base de datos para almacenar la información. A continuación veremos como preparar la base de datos para nuestra aplicación.

Configuración de la base de datos

El fichero `.env` de Laravel contiene la configuración relacionada con la aplicación y el entorno, como por ejemplo la configuración de la base de datos. Abre el fichero `.env` para visualizar las credenciales de la base de datos:

```
# Ejemplo de configuración creada por Laravel Sail
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=revistapp
DB_USERNAME=sail
DB_PASSWORD=password
```

Todas estas variables de configuración serán referenciadas desde el archivo de configuración `database.php`.

(Solo en Homestead) Creación de la base de datos

Este paso **solo será necesario si estamos utilizando Laravel Homestead** y no le hemos indicado a Homestead en su archivo de configuración `Homestead.yaml` que cree una base de datos. Para crear la base de datos accede al cliente de MySQL como root:

```
mysql -u root
```

Crea la base de datos si no está creada:

```
CREATE DATABASE revistapp;
```

Crear un usuario de base de datos

Es recomendable utilizar un usuario de base de datos diferente a `root`. Si deseas crear un nuevo usuario con permisos de acceso únicamente para la base de datos de la aplicación, ejecuta lo siguiente desde MySQL:

```
CREATE USER 'nombre_usuario'@'localhost' IDENTIFIED BY 'tu_contrasena';
```

El comando anterior crea el usuario con la contraseña indicada. El siguiente paso es otorgar permisos al usuario:

```
GRANT ALL PRIVILEGES ON * . * TO 'nombre_usuario'@'localhost';
```

Para que los cambios surjan efecto ejecuta lo siguiente:

```
FLUSH PRIVILEGES;
```

No olvides actualizar los datos de acceso a base de datos en el fichero de configuración `.env`.

Paso 6 - Crear la Migración (Migration)

Las Migraciones (Migrations) se utilizan para construir el esquema de la base de datos, es decir, crear y modificar las tablas que utilizará nuestra aplicación. Ejecuta el siguiente comando de Artisan para crear una nueva Migración para una tabla que llamaremos "articulos".

```
php artisan make:migration create_articulos_table --create=articulos
```

Laravel creará una nueva migración automáticamente en el directorio `database/migrations`. El nombre del archivo creado será el indicado en el comando anterior, en este caso `create_articulos_table`, precedido por un timestamp, por ejemplo: `2022_12_21_162755_create_articulos_table.php`.

El contenido de la clase creada será el siguiente:

```php
<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('articulos', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

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

Tal y como puedes deducir del código anterior, una migración contiene 2 métodos:

- `up()`: se utiliza para crear nuevas tablas, columnas o índices a la base de datos.
- `down()`: se utiliza para revertir operaciones realizadas por el método `up()`.

El siguiente paso es implementar el método `up()` para que cree las columnas tal y como queremos:

```php
<?php

public function up()
{
    Schema::create('articulos', function (Blueprint $table) {
        // Completar con los campos que queremos que contenta la tabla
'articulos':
        $table->id(); # Columna de tipo integer autoincremental (equivalente a
UNSIGNED INTEGER)
        $table->string('titulo'); # Columna de tipo string (equivalente a
VARCHAR)
        $table->text('contenido'); # Columna de tipo text (equivalente a TEXT)
        $table->timestamps(); # Crea las columnas created_at y updated_at de
tipo TIMESTAMP.
    });
}

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

Aparte de `id()`, `string()` o `integer()`, existen una gran variedad de tipos de columnas disponibles para definir las tablas. Puedes encontrarlas en la documentación oficial.

Una vez tenemos definida una migración, solo quedará ejecutarla para que así se ejecute en nuestra base de datos y aplique los cambios indicados. Para ejecutar las migraciones simplemente lanza el comando `migrate` de Artisan:

```
php artisan migrate
```

Hands on! (5/7)

Crea una migración para una tabla llamada `articulos` siguiendo los pasos anteriormente descritos. Completa la función `up()` para definir la tabla y lanza la migración.

Paso 7 - Crear un Modelo

Laravel incluye por defecto Eloquent ORM, el cual hace de la **interacción con la base de datos** una tarea fácil. Tal y como dice la documentación oficial:

> Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

En otras palabras, cada tabla de la base de datos corresponde a un Modelo, el cual permite ejecutar operaciones sobre esa tabla (insertar o leer registros por ejemplo). Este patrón es conocido como Active Record Pattern.

El nombre del modelo será la forma singular del nombre asignado a la tabla de la base de datos. Por ejemplo: `User` será el modelo correspondiente a la tabla `users`.

Creando un Modelo

Vamos a crear un Modelo:

```
php artisan make:model Articulo
```

El comando anterior ha creado una clase llamada Articulo en el directorio `app/Models` (es el directorio por defecto para los modelos de Eloquent a partir de Laravel 8).

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Articulo extends Model
{
    //
}
```

Por defecto un modelo de Eloquent almacena los registros en una tabla con el mismo nombre pero en plural. En este caso, `Articulo` interactuará con la tabla de la base de datos llamada `articulos`.

Laravel protege por defecto los modelos de forma que no se puedan generar registros de forma "masiva", es decir, en una única petición. De esta forma nos protegemos por ejemplo de que un usuario al editar su perfil pueda cambiar el valor de una propiedad llamada `is_admin`. Para que Laravel permita crear un artículo desde un formulario común, debemos indicarle los

campos que podrán ser completados y procesados. Para ello incluimos los nombres de los campos en una propiedad llamada `fillable`.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
class Articulo extends Model
{
    use HasFactory;
    protected $fillable = [
        'titulo',
        'contenido',
    ];
}
```

Consultando datos de la base de datos

Los modelos de Eloquent se pueden utilizar para recuperar información de las tablas relacionadas con ese modelo. Proporcionan métodos como los siguientes:

```php
<?php

use App\Models\Articulo;

// Recupera todos los modelos
$articulos = Articulo::all();

// Recupera un modelo a partir de su clave
$articulo = Articulo::find(1);

// Recupera el primer modelo que cumpla con los criterios indicados
$articulos = Articulo::where('active', 1)->first();

// Recupera los modelos que cumplan con los criterios indicados y de la forma
indicada:
$articulos = Articulo::where('active', 1)
               ->orderBy('titulo', 'desc')
               ->take(10)
               ->get();
```

Es posible iterar por la colección que devuelven los métodos `all()`, `get()` o `first()` de la siguiente forma:

```php
<?php

foreach ($articulos as $articulo) {
    echo $articulo->titulo;
}
```

Es importante mencionar que los métodos como `all()` o `get` no devuelven arrays típicos de PHP si no que devuelven una instancia de la clase `Collection` de Laravel. La diferencia es que estas colecciones tienen métodos adicionales que pueden ayudarnos en distintas situaciones, como por ejemplo: `last()`, `count()`, `sort()`, `merge()`, `filter()`, ... Puedes encontrar la lista de estos métodos de ayuda aquí: https://laravel.com/docs/9.x/collections#available-methods

Insertar información en la base de datos

El flujo de interacción que seguirá un usuario para insertar nuevos registros (artículos en nuestro caso) será el siguiente:

1. Acceder a la página con el formulario para enviar los datos. La ruta a la que deberá acceder será la siguiente: `articulos/create/`
2. Enviar los datos del formulario. La ruta que recibirá los datos será la siguiente: `articulos/` (POST).
3. Una vez enviados los datos, si todo ha ido bien nuestra aplicación le mostrará una nueva página.

Por lo tanto, será necesario crear dos nuevas rutas que invoquen a los métodos `create()` y `store()` del controlador.

El fichero `/routes/web.php` quedará así:

```php
<?php

use App\Http\Controllers\ArticuloController;

Route::get('articulos', [ArticuloController::class, 'index'])-
>name('articulos.index');
Route::get('articulos/create', [ArticuloController::class, 'create'])-
>name('articulos.create');
Route::get('articulos/{id}', [ArticuloController::class, 'show'])-
>name('articulos.show');
Route::post('articulos/', [ArticuloController::class, 'store'])-
>name('articulos.store');
```

La ruta empleada para mostrar el formulario será de tipo `GET` y la encargada de almacenar los datos será de tipo `POST`.

A continuación será necesario implementar los métodos del controlador:

```php
<?php

namespace App\Http\Controllers;

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

class ArticuloController extends Controller
{

    public function create()
    {
        return view('articulos.create');
    }

    public function store(Request $request)
    {
        //Validar la petición:
        $validated = $request->validate([
            'titulo' => 'required|string|max:255',
            'contenido' =>'required|string'
        ]);
        /* Si la validación falla se redirigirá al usuario
        a la página previa. Si pasa la validación, el controlador
        continuará ejecutándose.
        */

        // Insertar el artículo en la BBDD tras su validación.
        Articulo::create($validated);

        return redirect(route('articulos.index'));
    }
}
```

En el método `store()` se ha incluido una validación de los datos. Para conocer más acerca de las validaciones automáticas que Laravel puede hacer por nosotros, puedes visitar este enlace.

Si no se quiere validar los datos (no recomendado) se podría haber creado el nuevo artículo de la siguiente forma:

```php
<?php

    ...

    public function store(Request $request)
    {

        $articulo = new Articulo;
        $articulo->titulo = request('titulo');
        $articulo->contenido = request('contenido');
        $articulo->save();

        // Otra alternativa para la inserción:
        $articulo = Articulo::create([
            'titulo' => request('titulo'),
            'contenido' => request('contenido')
         ]);

        return redirect(route('articulos.index'));
    }
}
```

En el ejemplo anterior se puede apreciar que el método `request()` permite acceder a los datos enviados en la petición.

Por último, quedaría crear la vista que muestre el formulario:

`/resources/views/articulos/create.blade.php`:

```
<html>
<head>
    <title>RevistApp</title>
</head>
<body>
    <h1>Revistapp</h1>
    <h2>Crear un artículo:</h2>

    <form method="POST" action="{{ route('articulos.store') }}">
        @csrf
        <p><label>Titulo: </label><input type="text" name="titulo"></p>
        <p><label>Contenido: </label><input type="text" name="contenido"></p>
        <button type="submit">Crear</button>
    </form>
    <a href="{{ route('articulos.index') }}">Volver</a>
</body>
</html>
```

La directiva @csrf agrega un campo oculto con el Token de usuario para que Laravel nos proteja automáticamente de ataques XSS o de suplantación de identidad. Es necesario agregar siempre esta directiva.

Valores por defecto de un modelo

Al crear una instancia nueva de un modelo, los atributos de la instancia no tendrán ningún valor establecido. Si queremos definir valores por defecto, es posible hacerlo de la siguiente forma:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Articulo extends Model
{
    /**
     * The model's default values for attributes.
     *
     * @var array
     */
    protected $attributes = [
        'publicado' => false,
    ];
}
```

El ejemplo anterior muestra cómo establecer el atributo `publicado` como `false` cada vez que creemos un nuevo objeto de la clase `Articulo`.

Alternativas a Eloquent ORM

Laravel también permite interactuar con la base de datos mediante otras técnicas distintas a Eloquent ORM. Las alternativas disponibles son:

- Raw SQL: se trata de ejecutar sentencias SQL directamente contra la base de datos. Tienes toda la información disponible aquí.
- Query Builder: es una interfaz de comunicación con la base de datos que permite lanzar prácticamente cualquier consulta. A diferencia de la anterior, no es tan eficiente en cuanto a rendimiento pero aporta otras ventajas como la seguridad y abstracción de base de datos. Tienes toda la información disponible aquí.

Tinker: un potente REPL para Laravel

Tinker es un potente REPL o **consola interactiva** que viene por defecto en Laravel. Resulta muy útil durante el desarrollo ya que permite interactuar con nuestra aplicación Laravel y probar cantidad de cosas: eventos, acceso a datos, etc. Para iniciar Tinker hay que ejecutar el siguiente comando:

```
php artisan tinker
```

A partir de ese momento se puede comenzar a interactuar con nuestra aplicación, como muestra el ejemplo a continuación:

```
>>> $articulo = new App\Models\Articulo
=> App\Articulo {#3014}
>>> $articulo->titulo="AA";
=> "AA"
>>> $articulo->contenido="BBBB";
=> "BBBB"
>>> $articulo
=> App\Models\Articulo {#3014
     titulo: "Articulo numero 2",
     contenido: "Lorem ipsum...",
   }
>>> $articulos->save();
>>> App\Models\Articulo::count();
=> 2
>>> $articulos = App\Models\Articulo::all();
=> Illuminate\Database\Eloquent\Collection {#3035
     all: [
       App\Models\Articulo {#3036
         id: 1,
         titulo: "Articulo numero 1",
         contenido: "Lorem ipsum...",
         created_at: null,
         updated_at: null,
       },
       App\Models\Articulo {#3046
         id: 2,
         titulo: "Articulo numero 2",
         contenido: "Lorem ipsum...",
         created_at: "2019-12-15 15:32:04",
         updated_at: "2019-12-15 15:32:04",
       },
     ],
   }
```

Para salir se ejecuta el comando `exit`.

Hands on! (6/7)

Crea una vista para crear nuevos artículos y los métodos `create()` y `store()` en los controladores. De esta forma tu aplicación ya podrá crear artículos sin problemas. Tienes las soluciones en los códigos proporcionados junto con la explicación.

Hands on! (7/7)

- Actualiza el método `index()` para que utilice los datos almacenados en la base de datos. Puedes utilizar el método `all()` para recoger todos los artículos de la base de datos.
- Actualiza el método `show()` para que muestre el título y el contenido del artículo seleccionado.
- Añade también en la página incial un enlace a la página de creación de artículos.

Solución

`/App/Http/Controllers/ArticuloController.php`:

```
<?php

namespace App\Http\Controllers;

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

class ArticuloController extends Controller
{

    public function index()
    {
        $articulos = Articulo::all();
        return view('articulos.index', [
            'articulos' => $articulos
        ]);
    }

    public function show($id)
    {
        $articulo = Articulo::find($id);
        return view('articulos.show', [
            'articulo' => $articulo
        ]);
    }

    ...

}
```

Actualiza la vista `index.blade.php` para que utilice los datos correctamente y muestre el nuevo enlace:

```
<!DOCTYPE html>
<html>
<head>
    <title>RevistApp</title>
</head>
<body>
    <h1>Revistapp</h1>
    <h2>Listado artículos:</h2>
    <a href="{{ route('articulos.create') }}">Crear nuevo</a>
    <table>
        <tr><th>Enlace</th><th>Título</th></tr>
        @foreach ($articulos as $articulo)
        <tr>
            <td><a href="{{ route('articulos.show', $articulo->id) }}">Ver</a>
</td>
            <td>{{ ($articulo->titulo) }}</td>
        </tr>
        @endforeach
    </ul>
</body>
</html>
```

Actualiza la vista `show.blade.php` para que utilice los datos reales del artículo seleccionado:

```
<!DOCTYPE html>
<html>
<head>
    <title>RevistApp</title>
</head>
<body>
    <h1>Revistapp</h1>
    <h2>Detalle del artículo:</h2>
    <p>Titulo: {{ $articulo->titulo }}</p>
    <p>Contenido: {{ $articulo->contenido }}</p>
    <a href="{{ route('articulos.index') }}">Volver</a>
</body>
</html>
```

Bonus - Opciones (flags) de Artisan

Existen opciones muy útiles para generar archivos relacionados con los modelos. El siguiente ejemplo **crea un modelo junto con su controlador y migración utilizando un único comando**:

```
php artisan make:model Articulo –mcr
```

- -m indica la creación de una migración
- -c indica la creación de un controlador
- -r indica que el controlador creado será "resourceful" (incializado con los métodos).

Siguientes pasos

Ahora que ya tienes creada una aplicación de Laravel capaz de mostrar y almacenar datos desde la base de datos, puedes seguir avanzando con funcionalidades algo más avanzadas.