

# SISTEMA DE INFORMACIÓN PARA LA GESTIÓN DE UN TORNEO DE FUTBOL EN LARAVEL

HERI FERNANDO LONDOÑO SALGADO

## Instrucciones

El presente instructivo consta de tres tipos de contenido que son la teoría, las actividades y las tareas.

- **La teoría.** Contextualiza al aprendiz sobre el conocimiento que va a adquirir.
- **Las actividades.** Corresponden a las líneas de código y comandos representados en imágenes, que el aprendiz debe realizar.
- **Las tareas.** Son ejercicios que se dejan para que el aprendiz realice con base en la teoría y las actividades.

## Que es Laravel

- Framework PHP de código abierto, robusto y fácil de entender.
- Sigue un patrón de diseño modelo-vista-controlador.
- Reutiliza los componentes existentes, creando aplicaciones más estructuradas y pragmáticas.
- Ofrece un conjunto de funcionalidades y características que aumentan la velocidad del desarrollo web.
- Permite crear aplicaciones seguras evitando ataques web.

## Ventajas de Laravel

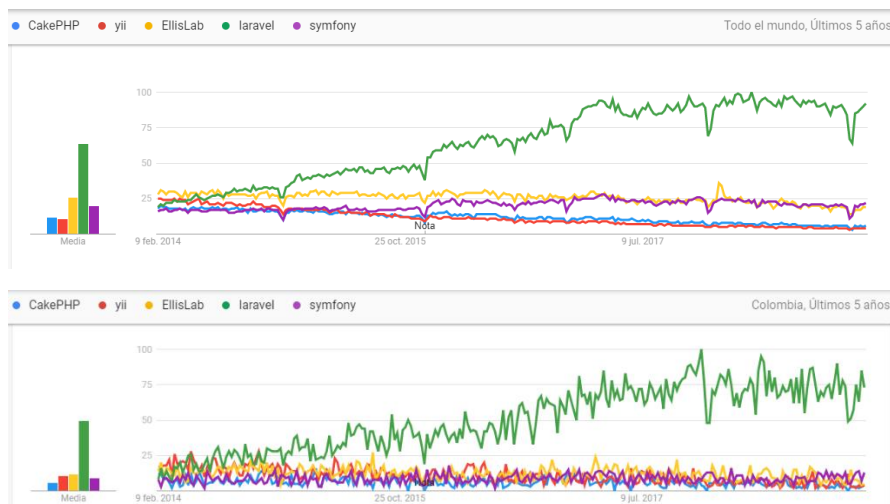
- La aplicación web es más escalable.
- Se ahorra mucho tiempo en el diseño de la aplicación por la reutilización de componentes.
- Incluye espacios de nombres e interfaces, por lo tanto, ayuda a organizar y administrar recursos.

## Características de Laravel

- **Modularidad.** 20 bibliotecas y módulos integrados con el administrador de dependencias de Composer, que ayudan a mejorar la aplicación.
- **Testeabilidad.** Incluye funciones para ejecución de pruebas a través de casos de prueba.
- **Enrutamiento.** Permite al usuario definir rutas en la aplicación, permitiendo escalar la aplicación y aumenta su rendimiento.
- **Gestión de la configuración.** Una aplicación diseñada en Laravel se ejecutará en diferentes entornos, con un manejo eficiente de la configuración.
- **Generador de consultas y ORM (Object Relational Mapper).** Incorpora un generador de consultas para acceso y consulta de bases de datos utilizando varios métodos de cadena simples.

- **Constructor de esquemas.** Mantiene la definición y esquema de la base de datos en código PHP, haciendo seguimiento de los cambios con respecto a las migraciones de la base de datos.
- **Motor de plantillas.** Usa el motor de plantillas Blade, un lenguaje de plantillas ligero para la creación de contenido dinámico.
- **Email.** Incluye una clase de correo que ayuda a enviar correo con contenido enriquecido y archivos adjuntos desde la aplicación web.
- **Autenticación.** Facilita el diseño de la autenticación, ya que incluye características como el registro, la contraseña olvidada y el envío de recordatorios de contraseña.
- **Redis.** Laravel usa Redis para conectarse a una sesión existente y caché de propósito general. Redis interactúa con la sesión directamente.
- **Colas.** Laravel incluye servicios de cola, como enviar por correo electrónico a un gran número de usuarios o una tarea cron específica. Estas colas ayudan a completar las tareas de una manera más fácil sin esperar a que se complete la tarea anterior.
- **Bus de Evento y Comando.** Incluye Command Bus, que ayuda a ejecutar comandos y enviar eventos de manera sencilla. Los comandos en Laravel actúan según el ciclo de vida de la aplicación.

## Tendencias entre los Frameworks de PHP



Fuente.

<https://trends.google.es/trends/explore?date=today%205-y&q=%2Fm%2F09t3sp,yii,%2Fm%2F02qgdj,laravel,symfony>

## Modelo Vista Controlador (MVC)

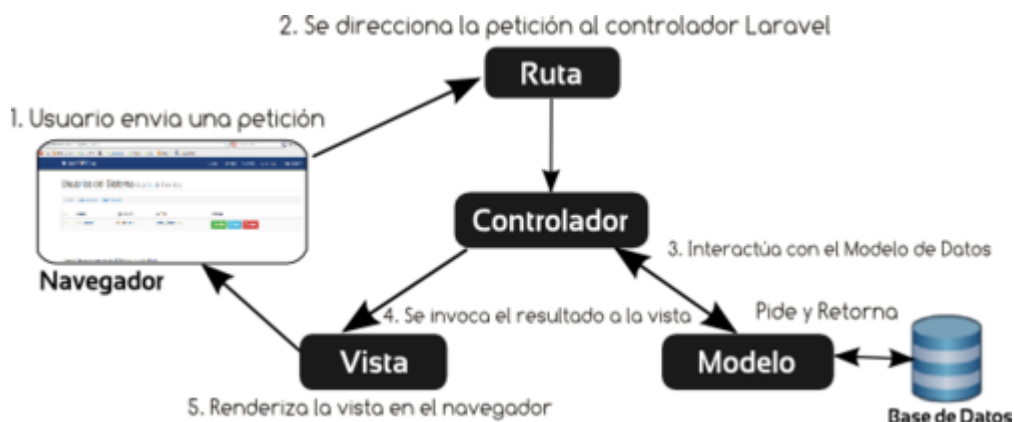
Es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. El estilo de llamada y retorno MVC, se ve frecuentemente en aplicaciones web, donde la vista es la página HTML y el código que provee de datos dinámicos a la página. El modelo

es el Sistema de Gestión de Base de Datos y la Lógica de negocio, y el controlador es el responsable de recibir los eventos de entrada desde la vista.

- **Modelo.** Es donde esta toda la lógica del negocio, la representación de todo el sistema incluido la interacción con una base de datos, si es que el programa así lo requiere.
- **Vista.** Representa la interfaz de usuario y todas las herramientas con las cuales el usuario hace uso del programa.
- **Controlador.** Es el que escucha los cambios en la vista y se los envía al modelo, el cual le regresa los datos a la vista, es un ciclo donde cada acción del usuario causa que se inicie de nuevo un nuevo ciclo. En cierta forma debe tener un registro de la relación entre ordenes que le pueden llegar y la lógica de negocio que le corresponde.

### Como funciona MVC

1. El usuario envía una petición
2. Se redirecciona la petición al controlador Laravel.
3. El controlador interactúa con el modelo de datos.
4. Se invoca el resultado a la vista.
5. La vista se renderiza en el navegador.



### Requisitos de Laravel

- PHP  $\geq$  7.3
- BCMath PHP Extension
- Ctype PHP Extension
- Fileinfo PHP Extension
- JSON PHP Extension
- Mbstring PHP Extension
- OpenSSL PHP Extension
- PDO PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

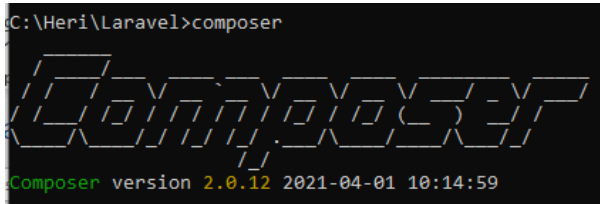
## Composer

Una forma de instalar Laravel es a través de Composer para la gestión de dependencias.

Es necesario descargar e instalar Composer desde la siguiente URL.

<https://getcomposer.org/download/>

Después de instalar Composer, se debe abrir CMD y comprobar la instalación tecleando el comando composer. Se debe mostrar el resultado de la figura.




---

### Actividad 1

Descargar, instalar y probar Composer.

---

## Instalar Laravel

Se utiliza Composer para instalar Laravel de esta manera:

```
composer create-project laravel/laravel example-app
```

Con este comando se crea un nuevo proyecto llamado mi-proyecto. Composer descarga los archivos del repositorio de distribución.

Para instalar Laravel en una versión diferente a la última, se especifica de esta manera:

```
composer create-project laravel/laravel miproyecto "6.*"
```

Para ver y navegar por la aplicación recién creada se ejecuta el comando:

```
php artisan serve.
```

Este comando ejecuta la aplicación en la url <http://localhost:8000>. Para detener la ejecución del servidor, se presiona Ctrl + C

---

### Actividad 2

Crear una carpeta en una ruta deseada del disco duro para alojar su proyecto.

Navegar en el CMD hasta esa ruta.

Instalar Laravel escribiendo y ejecutando el comando mostrado en la figura

```
composer create-project laravel/laravel TorneoV9
```

---

El proceso de descarga puede tardar unos minutos.

Para ver y navegar por la aplicación recién creada se debe ingresar a la carpeta del proyecto y ejecutar el comando:

```
php artisan serve
```

Este comando ejecuta la aplicación en la url `http://localhost:8000`.

Para detener la ejecución del servidor, se presiona `Ctrl + C`

---

### Actividad 3

Ingresar a la carpeta del proyecto

Cargar el servicio con el comando mostrado.

```
php artisan serve
```

Cuando el servicio esta cargado mostrara la URL donde se despliega

```
Laravel development server started: <http://127.0.0.1:8000>
```

Abrir el navegador e ir a esa URL para ver el proyecto en ejecución.

---

### Front Controller

Laravel funciona con un patrón de diseño Front Controller, este consiste en que un solo componente de la aplicación es el responsable de manejar de todas las peticiones HTTP que ésta recibe.

Todas las peticiones web entran a través del archivo `public/index.php`, quien comprueba en el archivo de rutas (`routes/web.php`) si la URL es válida y a que controlador tiene que hacer la petición.

Luego se llamará al método del controlador asignado para dicha ruta. Dependiendo de la petición, se obtienen los datos necesarios para pasarlos a la vista.

### Rutas

Son una capa muy importante en Laravel, posee su propia carpeta, (`routes`), donde se encuentran los archivos `web.php` y `api.php`.

Las rutas de la aplicación se deben definir en `web.php`, cualquier ruta no definida allí no será válida, generando una excepción (devuelve un error 404).

Las rutas, en su forma más sencilla, pueden devolver directamente un valor desde el propio archivo de rutas, pero también podrán generar la llamada a una vista o a un controlador.

Al definir la ruta, se debe definir el método HTTP, la URL de la petición y el llamado a una vista, a un controlador o el retorno de un dato.

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

```
Route::get('services', function () {
    return "Listado de servicios";
});
Route::get('services', function () {
    return view('services.index');
});
```

```
Route::get('/user/{id}', [MiControladorController::class, 'metodo']);
```

### Rutas con parámetros

Para añadir parámetros a una ruta, estos se indican entre llaves {} a continuación de la ruta. Es posible definir tantos parámetros como se requieran en la ruta. También es posible indicar que un parámetro es opcional, añadiendo el símbolo "?", y asignarle un valor por defecto.

```
Route::get('user/{id}', function($id)
{
    return 'User '.$id;
});
Route::get('user/{name?}', function($name = null)
{
    return $name;
});
Route::get('user/{name?}', function($name = 'Javi')
{
    return $name;
});
```

### Actividad 4

Escribir en el archivo de rutas, (web.php), el código mostrado en la imagen, para crear las rutas mostradas en la tabla. Probar en el navegador.

Ruta	Texto
/	Pantalla principal
equipos	Listado de equipos

equipos/create	Crear equipo
equipos/{id}	Detalle de equipo {id}
equipos/{id}/edit	Modificar equipo {id}

```
Route::get('/', function(){
    return "Pantalla principal";
});
Route::get('equipos', function(){
    return "Listado de equipos";
});
Route::get('equipos/create', function(){
    return "Crear equipo";
});
Route::get('equipos/{id}', function($id){
    return "Detalle de equipo $id";
});
Route::get('equipos/{id}/edit', function($id){
    return "Modificar equipo $id";
});
```

## Tarea

Escribir en el archivo de rutas, (web.php), el código necesario para crear las rutas mostradas en la tabla. Probar en el navegador.

Ruta	Texto
jugadores	Listado de jugadores
jugadores/create	Crear jugador
jugadores/{id}	Detalle de jugador {id}
jugadores/{id}/edit	Modificar jugador {id}

## Vistas

La vista se retorna desde una ruta o desde un controlador con el helper view, y se envía como primer parámetro el nombre del archivo, sin extensión, y de ser necesario, como segundo parámetro un array de datos que se le pasarán a la vista.

las vistas pueden estar anidadas dentro de cualquier carpeta dentro de resources/views. En este caso se usa dot notation para hacer referencia a vistas anidadas, donde las barras que separan las carpetas se sustituyen por puntos

### Pasar datos a una vista.

Para pasar datos a una vista se debe utilizar el segundo parámetro del método view, que acepta un array asociativo.

```
return view('greeting', ['name' => 'James']);
```

Laravel además ofrece una alternativa, donde en lugar de pasar un array como segundo parámetro se utiliza el método `with` para indicar una a una las variables o contenidos a enviar a la vista.

Dentro de la vista, se pueden acceder a estos valores utilizando su clave correspondiente.

Para imprimir una variable en la vista, se usa la sintaxis de dobles llaves `{{ $var }}` de Blade.

```
Route::get('/', function () {
    return view('home');
});
Route::get('catalog/show/{id}', function ($id) {
    return view('catalog.show', array('id' => $id));
});
Route::get('catalog/show/{id}', function ($id) {
    return view('catalog.show')->with('id', $id);
});
```

## Actividad 5

Crear los archivos, para las vistas dentro de la carpeta `resources/views` `resources/views/equipos` y agregar el contenido.

Archivo	Contenido	Ruta
inicio.blade.php	<code>&lt;h1&gt;Pantalla principal&lt;/h1&gt;</code>	/
equipos/index.blade.php	<code>&lt;h1&gt;Listado de equipos&lt;/h1&gt;</code>	equipos
equipos/create.blade.php	<code>&lt;h1&gt;Crear equipo&lt;/h1&gt;</code>	equipos/create
equipos/show.blade.php	<code>&lt;h1&gt;Detalle de equipo {{ \$id }}&lt;/h1&gt;</code>	equipos/{id}
equipos/edit.blade.php	<code>&lt;h1&gt;Modificar equipo {{ \$id }}&lt;/h1&gt;</code>	equipos/{id}/edit

Modificar en el archivo de rutas, (`web.php`), con el código mostrado en la imagen, para retornar las vistas dentro de las rutas. Probar en el navegador.

```
Route::get('/', function(){
    return view('inicio');
});
Route::get('equipos', function(){
    return view('equipos.index');
});
Route::get('equipos/create', function(){
    return view('equipos.create');
});
Route::get('equipos/{id}', function($id){
    return view('equipos.show')->with('id', $id);
});
Route::get('equipos/{id}/edit', function($id){
    return view('equipos.edit')->with('id', $id);
});
```



## Tarea

Crear una carpeta llamada “jugadores” dentro de “resources/views”, y dentro de esta crear los archivos para las vistas y agregar el contenido definido en la tabla. Crear las respectivas rutas. Probar en el navegador.

Archivo	Texto	Ruta
index.blade.php	Listado de jugadores	jugadores
create.blade.php	Crear jugador	jugadores/create
show.blade.php	Detalle de jugador {id}	jugadores/{id}
edit.blade.php	Modificar jugador {id}	jugadores/{id}/edit

## Plantillas Blade

Blade es un motor de plantillas incluido con Laravel. Las vistas en Blade usan la extensión blade.php y se almacenan en el directorio resources/views.

Blade permite la definición de layouts para crear una estructura HTML base con secciones que serán rellenadas por otras plantillas o vistas hijas.

Las vistas que extienden un layout deben sobrescribir las secciones del layout.

La directiva @section permite añadir contenido en las plantillas hijas, mientras que @yield será sustituido por el contenido que se indique.

El método @yield también permite establecer un contenido por defecto mediante su segundo parámetro:

@yield('section', 'Contenido por defecto')

```
<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show
    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

## Heredar una plantilla

Cuando se define una vista hija, se utiliza la directiva Blade `@extends` para especificar de qué layout debe "heredar". Las vistas que extienden un layout de Blade pueden inyectar contenido en las secciones mediante las directivas `@section`.

La directiva `@parent` será reemplazada por el contenido del layout cuando se procese la vista.

La directiva `@endsection` finaliza la sección.

```
@extends('layouts.app')
@section('title', 'Page Title')
@section('sidebar')
    @parent
    <p>This is appended to the master sidebar.</p>
@endsection
@section('content')
    <p>This is my body content.</p>
@endsection
```

This is the master sidebar.

This is appended to the master sidebar.

This is my body content.

## Incluir una plantilla

En Blade se puede incluir el contenido de una plantilla dentro de otra, con la instrucción `@include`.

```
@include('view_name')
```

Además, se puede pasar un array de datos a la plantilla a cargar usando el segundo parámetro del método `include`.

```
@include('view_name', array('some' => 'data'))
```

Esta opción es muy útil para crear vistas que sean reutilizables o para separar el contenido de una vista en varios archivos.

Si se intenta `@include` en una vista que no existe, Laravel lanzará un error. Si desea incluir una vista que puede o no estar presente, debe utilizar la directiva `@includeIf`.

```
@includeIf('view.name', ['some' => 'data'])
```

---

## Actividad 6

A continuación, se creará el layout base que utilizaran las vistas del sitio web y se incluirá la librería Bootstrap para utilizarla como estilo base.

Crear la carpeta layouts dentro de resources/views.

Dentro de esa carpeta layouts crear el archivo base.blade.php

Dentro del archivo base.blade.php se copia el contenido de la plantilla base que propone Bootstrap.

<https://getbootstrap.com/docs/4.6/getting-started/introduction/#starter-template>

y para incluir Blade, se modificará lo siguiente.

La etiqueta title será reemplazada por.

```
<title>Torneo - @yield('title')</title>
```

Dentro de la sección <body> se elimina el texto que viene de ejemplo y se incluye la referencia a la barra de navegación que se definirá más adelante.

```
@include('partials.navbar')
```

Luego se añade la sección donde aparecerá el contenido.

```
<div class="container">
```

```
    @yield('content')
```

```
</div>
```

Por lo tanto, la plantilla debe quedar como se muestra en la imagen.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css"
    integrity="sha384-B0vP5xmATw1+K9KRQjQERjvTumQW0nPEzvF6L/Z6nronJ3oU0FUFPcJEUQouq2+1" crossorigin="anonymous"
    ">
    <title>Torneo - @yield('title')</title>
  </head>
  <body>
    <div class="container">
      @include('partials.navbar')
      @yield('content')
    </div>
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/
    zy4C+0GpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/js/bootstrap.bundle.min.js" integrity="
    sha384-Piv4xVNRyMGPqkS2by6br4gNJ7DXjqk09RmUpJ8jgGtD7zP9yug3goQfGII0yAns" crossorigin="anonymous"></script>
  </body>
</html>
```

**Nota.** En caso de no querer usar Bootstrap de manera remota, sino local, se deben descargar los archivos css y js y guardarlos respectivamente en las carpetas public/css y public/js, y referenciarlos con la función asset, con la cual se ubica la ruta relativa a partir de la carpeta public.

```
<link rel="stylesheet" href="{{ asset('css/bootstrap.min.css') }}">
```

```
<script src="{{ asset('js/jquery-3.4.1.slim.min.js') }}"></script>
```

A continuación, se creará la plantilla con el menú.

Crear la carpeta partials dentro de resources/views.

Dentro de esa carpeta partials crear el archivo navbar.blade.php

Dentro del archivo se creará el un menú basado en Bootstrap que se muestra en la imagen.

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Torneo</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent"
    aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown"
          aria-haspopup="true" aria-expanded="false">
          Equipos
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="/equipos">Ver equipos</a>
          <a class="dropdown-item" href="/equipos/create">Nuevo equipo</a>
        </div>
      </li>
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown"
          aria-haspopup="true" aria-expanded="false">
          Jugadores
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="/jugadores">Ver jugadores</a>
          <a class="dropdown-item" href="/jugadores/create">Nuevo jugador</a>
        </div>
      </li>
    </ul>
  </div>
</nav>
```

Modificar las vistas para heredar del layout base, reemplazando el title y mostrar el contenido, de acuerdo con las imágenes.

resources/views/inicio.blade.php

```
@extends('layouts.base')
@section('title', 'Torneo')
@section('content')
  <h1>Pantalla principal</h1>
@endsection
```

resources/views/equipos/index.blade.php

```
@extends('layouts.base')
@section('title', 'Equipos')
@section('content')
  <h1>Listado de equipos</h1>
@endsection
```

resources/views/equipos/create.blade.php

```
@extends('layouts.base')
@section('title', 'Crear equipo')
@section('content')
    <h1>Crear equipo</h1>
@endsection
```

resoues/views/equipos/show.blade.php

```
@extends('layouts.base')
@section('title', 'Detalle equipo')
@section('content')
    <h1>Detalle de equipo {{ $id }}</h1>
@endsection
```

resoues/views/equipos/edit.blade.php

```
@extends('layouts.base')
@section('title', 'Modificar equipo')
@section('content')
    <h1>Modificar equipo {{ $id }}</h1>
@endsection
```

Probar en el navegador.

La imagen muestra el resultado para la pantalla principal.

Torneo Equipos ▼ Jugadores ▼

## Pantalla principal

### Tarea

Crear las vistas de los jugadores, (index, create, show y edit), de tal forma que hereden del layout base.

Probar en el navegador.

### Controladores

Un controlador es una clase que permite agrupar las acciones de una sección en particular. Normalmente se les añade el sufijo Controller, por ejemplo, ProductosController, ContactosController, RegistrosController. Y los métodos de ProductosController serían, por ejemplo, “crear”, “modificar”, “eliminar”, “listar”, etc.

Los controladores se almacenan en el directorio app/Http/Controllers.

Cada una de sus acciones se definen dentro de routes/web.php.

```
use App\Http\Controllers\MiControladorController;
Route::get('/user/{id}', [MiControladorController::class, 'metodo']);
```

Para crear un controlador se puede utilizar el comando de Artisan.

php artisan make:controller NombreControladorController

Este comando creará el controlador dentro de la carpeta app/Http/Controllers y lo completará con el código básico.

## Actividad 7

Crear el controlador InicioController, ejecutando el comando mostrado.

```
php artisan make:controller InicioController
```

**Nota.** Si el servidor está en ejecución, presione Ctrl + C

El archivo creado tendrá el contenido mostrado en la imagen.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class InicioController extends Controller
{
}
```

Al controlador creado se le agregará el método index, el cual retornará la vista relacionada

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class InicioController extends Controller
{
    public function index(){
        return view('inicio');
    }
}
```

Dentro del archivo de rutas, modificar la ruta '/' de tal forma que haga referencia al controlador y al método.

```
use App\Http\Controllers\InicioController;

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

## Resource Controller

Artisan permite crear un controlador que realiza todas las acciones del CRUD, y además de eso enrutarlas con una línea de código.

Para crear el controlador se usa el comando.

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

Este comando crea un controlador con los siguientes métodos

index: Mostrar registros existentes (GET)

create: Mostrar nuevos datos a crear (GET)

store: Crear nuevos datos (POST)

show: Mostrar un registro (GET)

edit: Mostrar un registro a modificar (GET)

update: Modificar un registro (PUT)

destroy: Eliminar un registro (DELETE)

En la tabla se presenta el verbo, la URI, y la acción por defecto generadas por el Resource Controller, y la forma como se manejan las rutas.

Verbo	URI	Acción	Vista
GET	equipos	index	equipos.index
GET	equipos/create	create	equipos.create
POST	equipos	store	
GET	equipos/{id}	show	equipos.show
GET	equipos/{id}/edit	edit	equipos.edit
PUT	equipos/{id}	update	
DELETE	equipos/{id}	destroy	

Para que este controlador pueda ser accedido, se debe hacer su llamado dentro del archivo de rutas, routes/web.php.

```
Route::resource('ruta', NombreControladorController::class);
```

Es posible registrar varios controladores recurso pasando un arreglo al método resources.

```
Route::resources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

---

## Actividad 8

A continuación, se creará el Resource Controller EquiposController, este generará automáticamente las acciones para el CRUD.

En la consola escribir el comando mostrado en la imagen.

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

El archivo creado, app/Http/Controllers/EquiposController, posee 7 métodos relacionados con el CRUD. A continuación, se agregará el contenido a los métodos index, create, show, y edit, como se muestra en la imagen, para el cargue de vistas.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class EquiposController extends Controller
{
    public function index()
    {
        return view('equipos.index');
    }
    public function create()
    {
        return view('equipos.create');
    }
    public function store(Request $request)
    {
    }
    public function show($id)
    {
        return view('equipos.show')
            ->with('id', $id);
    }
    public function edit($id)
    {
        return view('equipos.edit')
            ->with('id', $id);
    }
    public function update(Request $request, $id)
    {
    }
    public function destroy($id)
    {
    }
}
```

A continuación, en el archivo de rutas se define la ruta del recurso, y se pueden eliminar las rutas individuales hacia las acciones de equipos, por lo tanto, el archivo de rutas quedará como se muestra en la imagen.



```
<?php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\InicioController;
use App\Http\Controllers\EquiposController;

Route::get('/', [InicioController::class, 'index']);
Route::resource('equipos', EquiposController::class);
```

Probar en el navegador.

## Tarea

crear el Resource Controller JugadoresController,

agregar el contenido a los métodos index, create, show, y edit, para el cargue de vistas.

definir la ruta del recurso.

## Actividad 9

Con el fin de simular un comportamiento dinámico, se agregará un array de equipos con el nombre, el dt, el municipio y el escudo, en el controlador EquiposController.

Agregar el array mostrado como atributo de EquiposController

```
private $equipos = array(
    array(
        'nombre' => 'equipo 1',
        'dt' => 'D.T. 1',
        'municipio' => 'Municipio 1',
        'escudo' => 'http://ximg.es/200x300/000/fff&text=equipo1'
    ),
    array(
        'nombre' => 'equipo 2',
        'dt' => 'D.T. 2',
        'municipio' => 'Municipio 2',
        'escudo' => 'http://ximg.es/200x300/000/fff&text=equipo2'
    ),
    array(
        'nombre' => 'equipo 3',
        'dt' => 'D.T. 3',
        'municipio' => 'Municipio 3',
        'escudo' => 'http://ximg.es/200x300/000/fff&text=equipo3'
    ),
    array(
        'nombre' => 'equipo 4',
        'dt' => 'D.T. 4',
        'municipio' => 'Municipio 4',
        'escudo' => 'http://ximg.es/200x300/000/fff&text=equipo4'
    )
);
```

El index de equipos mostrará el listado de los equipos inscritos en el torneo. Los equipos están disponibles en el array creado como atributo, (más adelante se importarán desde la base de datos).

En el método index del controlador se modifica la generación de la vista para pasarle el array de equipos.

```
public function index()
{
    return view('equipos.index')
        ->with('equipos', $this->equipos);
}
```

En la vista index se agrega el código html en la sección content para que muestre el nombre y el escudo de cada equipo, y además se genere el enlace para mostrar su información detallada.

En este caso se utiliza un ciclo @foreach de Blade.

```
@extends('layouts.base')
@section('title', 'Torneo')
@section('content')
    <div class="row">
        @foreach ($equipos as $key=>$equipo)
            <div class="col-3 text-center">
                <a href="/equipos/{{ $key }}">
                    
                    <h4>{{ $equipo['nombre'] }}</h4>
                </a>
            </div>
        @endforeach
    </div>
@endsection
```

Probar en el navegador.

El resultado debe quedar como el mostrado en la imagen.





## Tarea

Crear un array de jugadores dentro de JugadoresController, con los campos foto, nombre, posición, número y equipo.

El campo posición puede ser Delantero, Volante, Defensa, Portero.

Modificar la vista index, de tal forma que muestre los datos de los jugadores en una tabla y a cada jugador le asigne un botón para ver detalle, otro para actualizar y otro para eliminar.

En la imagen se muestra un ejemplo de la distribución de los elementos.

Foto	Nombre	Posicion	Numero	Equipo	Accion
	James Rodriguez	Volante	3	Colombia	Ver Modificar Eliminar
	David Ospina	Portero	1	Colombia	Ver Modificar Eliminar

## Actividad 10

A continuación, se van a realizar las acciones necesarias para mostrar la vista detalle de un equipo.

Inicialmente se modifica el método show de EquiposController, de tal forma que reciba el equipo y la posición del equipo en el array.

```
public function show($id)
{
    return view('equipos.show')
        ->with('id', $id)
        ->with('equipo', $this->equipos[$id]);
}
```

En la vista show se agrega el código html en la sección content para que muestre el detalle del equipo seleccionado, y además tres botones, uno para eliminar, otro para modificar y otro para volver.

```
@extends('layouts.base')
@section('title', 'Detalle equipo')
@section('content')
    <div class="row">
        <div class="col-sm-4">
            
        </div>
        <div class="col-sm-8">
            <p class="h3">{{ $equipo['nombre'] }}</p>
            <p class="h5">{{ $equipo['dt'] }}</p>
            <p class="h5">{{ $equipo['municipio'] }}</p>
            <a href="#" class="btn btn-danger">Eliminar</a>
            <a href="/equipos/{{ $id }}/edit" class="btn btn-warning">Modificar</a>
            <a href="/equipos" class="btn btn-outline-dark">Ver equipos</a>
        </div>
    </div>
@endsection
```

Probar en el navegador.

El resultado debe ser similar al mostrado en la imagen.



## Tarea

Modificar el método y la vista show de jugadores para mostrar la información detallada del jugador seleccionado en la tabla.

## Formularios

Para crear un formulario se utilizan etiquetas de HTML, tanto para el formulario, como para sus elementos.

Es posible también utilizar directivas de Blade para construir el contenido HTML.

La acción del formulario apunta a una URL

HTML solo permite el uso de formularios de tipo GET o POST. Para realizar solicitudes de tipo PUT, PATCH o DELETE, se debe agregar un campo @method oculto para falsificar estos verbos HTTP. Esto se logra incluyendo la directiva de Blade @method, ejemplo. @method ('PUT')

## Protección CSRF

Laravel facilita la protección frente a ataques de falsificación de solicitudes entre sitios (CSRF), generando automáticamente un "token" CSRF para cada sesión de usuario activa. Este token se usa para verificar que el usuario autenticado es el que realiza las solicitudes a la aplicación.

Este token se debe incluir en todos los formularios de la aplicación, usando la directiva @csrf de Blade.

## Actividad 11

Modificar la vista create de equipos insertando el código que aparece en la imagen, de tal forma que muestre el formulario para la creación de un nuevo equipo.

```

@extends('layouts.base')
@section('title', 'Crear equipo')
@section('content')
    <form action="/equipos" method="post" enctype="multipart/form-data">
        @csrf
        <div class="form-group">
            <label for="nombre">Nombre</label>
            <input type="text" name="nombre" id="nombre" class="form-control">
        </div>
        <div class="form-group">
            <label for="dt">D.T.</label>
            <input type="text" name="dt" id="dt" class="form-control">
        </div>
        <div class="form-group">
            <label for="municipio">Municipio</label>
            <select class="form-control" id="municipio" name="municipio">
                <option value="1">Manizales</option>
                <option value="2">Pereira</option>
                <option value="3">Armenia</option>
            </select>
        </div>
        <div class="form-group">
            <label for="escudo">Escudo</label>
            <input type="file" name="escudo" id="escudo" class="form-control-file">
        </div>
        <button type="submit" class="btn btn-primary">Crear</button>
    </form>
@endsection

```

Probar en el navegador.

El resultado debe ser similar al mostrado en la imagen.

Torneo Equipos ▾ Jugadores ▾

Nombre

D.T.

Municipio

Manizales ▾

Escudo

Seleccionar archivo Ningún archivo seleccionado

Crear

## Tarea

Crear un formulario para insertar un nuevo jugador.

Este debe tener los campos nombre (text), posición (select), numero (number), equipo (select) y foto (file).

El campo posición debe ser de tipo select, ya que puede ser Delantero, Volante, Defensa, Portero.

En el select de equipos se deben cargar tres equipos.

---

## Base de datos

Laravel permite una interacción simple con bases de datos, utilizando el fluent query builder y Eloquent ORM. Actualmente, soporta MySQL, PostgreSQL, SQLite y SQLServer.

---

## Actividad

En cualquier gestor de bases de datos MySQL, crear una base de datos llamada torneo.

---

para trabajar con bases de datos se debe configurar la conexión.

La configuración se encuentra en config/database.php, donde aparece la línea.

'default' => env('DB\_CONNECTION', 'mysql'),

Este valor indica el tipo de base de datos a utilizar por defecto, obteniendo el valor de la variable DB\_CONNECTION del fichero .env, en caso que no esté definida devolverá por defecto mysql.

Los valores se deben modificar en el archivo .env de la raíz del proyecto.

---

## Actividad

Abrir el archivo .env de la raíz del proyecto, buscar los parámetros de configuración de mysql, y asignar los valores que se muestran en la imagen.

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

---

## Migraciones

Las migraciones son un control de versiones para la base de datos, permiten modificar y compartir el esquema de la base de datos de la aplicación.

Las migraciones se crean en archivos PHP con la descripción de la tabla a crear y posteriormente, si se quiere modificar dicha tabla se añadiría una nueva migración con los campos a modificar. Artisan incluye comandos para crear migraciones, para ejecutar las migraciones o para hacer rollback de las mismas.

## Crear migraciones

Para crear una nueva migración se utiliza el comando de Artisan `make:migration`, seguido del nombre del nombre del archivo.

```
php artisan make:migration create_users_table
```

```
php artisan make:migration add_votes_to_user_table
```

La migración se almacenará en la carpeta `database/migrations` con un timestamp que permite conocer el orden en el que se deben ejecutar.

No hay restricción con el nombre del archivo, pero se recomienda seguir el patrón, `create_<table-name>_table` o `<action>_<field>_to_<table-name>_table`. Si se utiliza este patrón, Laravel crea la plantilla básica de la migración.

## Actividad 11

Crear las migraciones para las tablas `municipios`, `equipos`, `posiciones` y `jugadores`. Debe ser en ese orden, ya que, de acuerdo con el modelo relacional mostrado en la tabla, la tabla `equipos` tiene una llave foránea hacia `municipios`, y `jugadores` se debe crear de ultima, ya que tiene llaves foráneas hacia `equipos` y `posiciones`.



Ejecutar los comandos mostrados en las imágenes para crear las migraciones. (Dar Ctrl + C en caso de que el servidor este en ejecución).

```
php artisan make:migration create_municipios_table
```

```
php artisan make:migration create_equipos_table
```

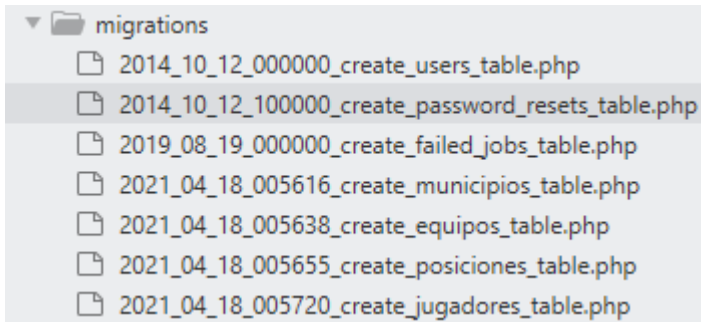
```
php artisan make:migration create_posiciones_table
```

```
php artisan make:migration create_jugadores_table
```

Una vez creada cada migración, se debe mostrar un mensaje confirmándolo.

```
Created Migration: 2020_04_29_173640_create_equipos_table
```

Las migraciones se pueden encontrar en la carpeta database/migrations.



Adicional a las migraciones creadas, se pueden encontrar tres migraciones adicionales.

### Estructura de la migración

Una migración contiene los métodos up y down.

El método up se usa para crear nuevas tablas, columnas, o índices en la base de datos, mientras que el método down debe revertir las operaciones realizadas por el método up.

En ambos métodos se puede usar Laravel schema builder para crear y modificar tablas.

```
class CreateEquiposTable extends Migration
{
    public function up()
    {
        Schema::create('equipos', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('equipos');
    }
}
```

### Tablas

Para crear una tabla se utiliza el método create de Schema. Este método recibe el nombre de la tabla y una función donde se definen los elementos.

Para renombrar una tabla se utiliza el método rename.

```
Schema::rename($from, $to);
```

Para borrar una tabla existente se utilizan los métodos drop o dropIfExists

```
Schema::drop('users');
```



```
Schema::dropIfExists('users');
```

## Tipos de columnas

La figura muestra los tipos de columnas que se pueden crear con Laravel.

<a href="#"><u>bigIncrements</u></a>	<a href="#"><u>lineString</u></a>	<a href="#"><u>string</u></a>
<a href="#"><u>bigInteger</u></a>	<a href="#"><u>longText</u></a>	<a href="#"><u>text</u></a>
<a href="#"><u>binary</u></a>	<a href="#"><u>macAddress</u></a>	<a href="#"><u>timeTz</u></a>
<a href="#"><u>boolean</u></a>	<a href="#"><u>mediumIncrements</u></a>	<a href="#"><u>time</u></a>
<a href="#"><u>char</u></a>	<a href="#"><u>mediumInteger</u></a>	<a href="#"><u>timestampTz</u></a>
<a href="#"><u>dateTimeTz</u></a>	<a href="#"><u>mediumText</u></a>	<a href="#"><u>timestamp</u></a>
<a href="#"><u>dateTime</u></a>	<a href="#"><u>morphs</u></a>	<a href="#"><u>timestampsTz</u></a>
<a href="#"><u>date</u></a>	<a href="#"><u>multiLineString</u></a>	<a href="#"><u>timestamps</u></a>
<a href="#"><u>decimal</u></a>	<a href="#"><u>multiPoint</u></a>	<a href="#"><u>tinyIncrements</u></a>
<a href="#"><u>double</u></a>	<a href="#"><u>multiPolygon</u></a>	<a href="#"><u>tinyInteger</u></a>
<a href="#"><u>enum</u></a>	<a href="#"><u>nullableMorphs</u></a>	<a href="#"><u>tinyText</u></a>
<a href="#"><u>float</u></a>	<a href="#"><u>nullableTimestamps</u></a>	<a href="#"><u>unsignedBigInteger</u></a>
<a href="#"><u>foreignId</u></a>	<a href="#"><u>nullableUuidMorphs</u></a>	<a href="#"><u>unsignedDecimal</u></a>
<a href="#"><u>geometryCollection</u></a>	<a href="#"><u>point</u></a>	<a href="#"><u>unsignedInteger</u></a>
<a href="#"><u>geometry</u></a>	<a href="#"><u>polygon</u></a>	<a href="#"><u>unsignedMediumInteger</u></a>
<a href="#"><u>id</u></a>	<a href="#"><u>rememberToken</u></a>	<a href="#"><u>unsignedSmallInteger</u></a>
<a href="#"><u>increments</u></a>	<a href="#"><u>set</u></a>	<a href="#"><u>unsignedTinyInteger</u></a>
<a href="#"><u>integer</u></a>	<a href="#"><u>smallIncrements</u></a>	<a href="#"><u>uuidMorphs</u></a>
<a href="#"><u>ipAddress</u></a>	<a href="#"><u>smallInteger</u></a>	<a href="#"><u>uuid</u></a>
<a href="#"><u>json</u></a>	<a href="#"><u>softDeletesTz</u></a>	<a href="#"><u>year</u></a>
<a href="#"><u>jsonb</u></a>	<a href="#"><u>softDeletes</u></a>	

## Modificadores

En la imagen se muestran los modificadores que se pueden usar con Laravel.

Modifier	Description
<code>-&gt;after('column')</code>	Place the column "after" another column (MySQL).
<code>-&gt;autoIncrement()</code>	Set INTEGER columns as auto-incrementing (primary key).
<code>-&gt;charset('utf8mb4')</code>	Specify a character set for the column (MySQL).
<code>-&gt;collation('utf8mb4_unicode_ci')</code>	Specify a collation for the column (MySQL/PostgreSQL/SQL Server).
<code>-&gt;comment('my comment')</code>	Add a comment to a column (MySQL/PostgreSQL).
<code>-&gt;default(\$value)</code>	Specify a "default" value for the column.
<code>-&gt;first()</code>	Place the column "first" in the table (MySQL).
<code>-&gt;from(\$integer)</code>	Set the starting value of an auto-incrementing field (MySQL / PostgreSQL).
<code>-&gt;nullable(\$value = true)</code>	Allow NULL values to be inserted into the column.
<code>-&gt;storedAs(\$expression)</code>	Create a stored generated column (MySQL).
<code>-&gt;unsigned()</code>	Set INTEGER columns as UNSIGNED (MySQL).
<code>-&gt;useCurrent()</code>	Set TIMESTAMP columns to use CURRENT_TIMESTAMP as default value.
<code>-&gt;useCurrentOnUpdate()</code>	Set TIMESTAMP columns to use CURRENT_TIMESTAMP when a record is updated.
<code>-&gt;virtualAs(\$expression)</code>	Create a virtual generated column (MySQL).
<code>-&gt;generatedAs(\$expression)</code>	Create an identity column with specified sequence options (PostgreSQL).
<code>-&gt;always()</code>	Defines the precedence of sequence values over input for an identity column (PostgreSQL).

## Crear índices

La figura muestra la forma en la cual Laravel permite crear índices.

Command	Description
<code>\$table-&gt;primary('id');</code>	Adds a primary key.
<code>\$table-&gt;primary(['id', 'parent_id']);</code>	Adds composite keys.
<code>\$table-&gt;unique('email');</code>	Adds a unique index.
<code>\$table-&gt;index('state');</code>	Adds an index.
<code>\$table-&gt;spatialIndex('location');</code>	Adds a spatial index (except SQLite).

## Longitud de índices

Laravel usa por defecto utf8mb4 que incluye soporte para almacenar emojis en la base de datos. si se está ejecutando una versión de MySQL inferior a la 5.7.7 o MariaDB inferior a 10.2.2 se debe modificar manualmente la longitud de strings.

Esto se puede hacer llamando el método `Schema::defaultStringLength(191)`; dentro del método `boot` en `app/Providers/AppServiceProvider`

## Actividad 12

Con el fin de evitar errores al momento de crear las tablas en la base de datos, se van a modificar manualmente la longitud de los strings.

`app/Providers/AppServiceProvider.php`

Agregar el código para incluir `Schema`.

```
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Schema;
```

En el método `boot` incluir la instrucción mostrada en la imagen.

```
public function boot()
{
    Schema::defaultStringLength(191);
}
```

## Llaves foráneas

Las imágenes mostradas a continuación muestran la forma en la cual se crear llaves foráneas.

```
Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

Esta sintaxis es muy detallada, Laravel proporciona métodos más breves que utilizan convenciones para proporcionar una mejor experiencia de desarrollador. El ejemplo anterior se puede reescribir así:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

El método `foreignId()` es un alias para `unsignedBigInteger`, mientras que el método `constrained()` usará convenciones para determinar la tabla y el nombre de la columna a la que se hace referencia. Si el nombre de su tabla no coincide con las

convenciones de Laravel, se puede especificar el nombre de la tabla pasándolo como un argumento al método `constrained()`.

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users');
});
```

También se puede especificar la acción deseada para las propiedades "al eliminar" y "al actualizar".

```
$table->foreignId('user_id')
    ->constrained()
    ->onUpdate('cascade')
    ->onDelete('cascade');
```

### Actividad 13

A continuación, se crearán las migraciones para las tablas municipios, equipos, posiciones y jugadores.

Abrir el archivo de migración de municipios ubicado en la carpeta database/migrations, y modificar el código como se muestra en la imagen.

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateMunicipiosTable extends Migration
{
    public function up()
    {
        Schema::create('municipios', function (Blueprint $table) {
            $table->id();
            $table->string('nombre', 100);
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('municipios');
    }
}
```

Abrir el archivo de migración de equipos ubicado en la carpeta database/migrations, y modificar el código como se muestra en la imagen.

```

<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateEquiposTable extends Migration
{
    public function up()
    {
        Schema::create('equipos', function (Blueprint $table) {
            $table->id();
            $table->string('nombre', 100);
            $table->string('dt', 100);
            $table->foreignId('municipio_id')->constrained();
            $table->string('escudo', 100);
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('equipos');
    }
}

```

Abrir el archivo de migración de posiciones ubicado en la carpeta database/migrations, y modificar el código como se muestra en la imagen.

```

<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreatePosicionesTable extends Migration
{
    public function up()
    {
        Schema::create('posiciones', function (Blueprint $table) {
            $table->id();
            $table->string('nombre', 100);
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('posiciones');
    }
}

```

Abrir el archivo de migración de jugadores ubicado en la carpeta database/migrations, y modificar el código como se muestra en la imagen.

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateJugadoresTable extends Migration
{
    public function up()
    {
        Schema::create('jugadores', function (Blueprint $table) {
            $table->id();
            $table->string('nombre', 100);
            $table->foreignId('posicion_id')->constrained('posiciones');
            $table->integer('numero');
            $table->string('foto', 100);
            $table->foreignId('equipo_id')->constrained();
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('jugadores');
    }
}
```

---

## Ejecutar migraciones

Para ejecutar todas las migraciones se utiliza el comando.

php artisan migrate

---

## Actividad 14

Ejecutar las migraciones para crear las tablas de la base de datos.

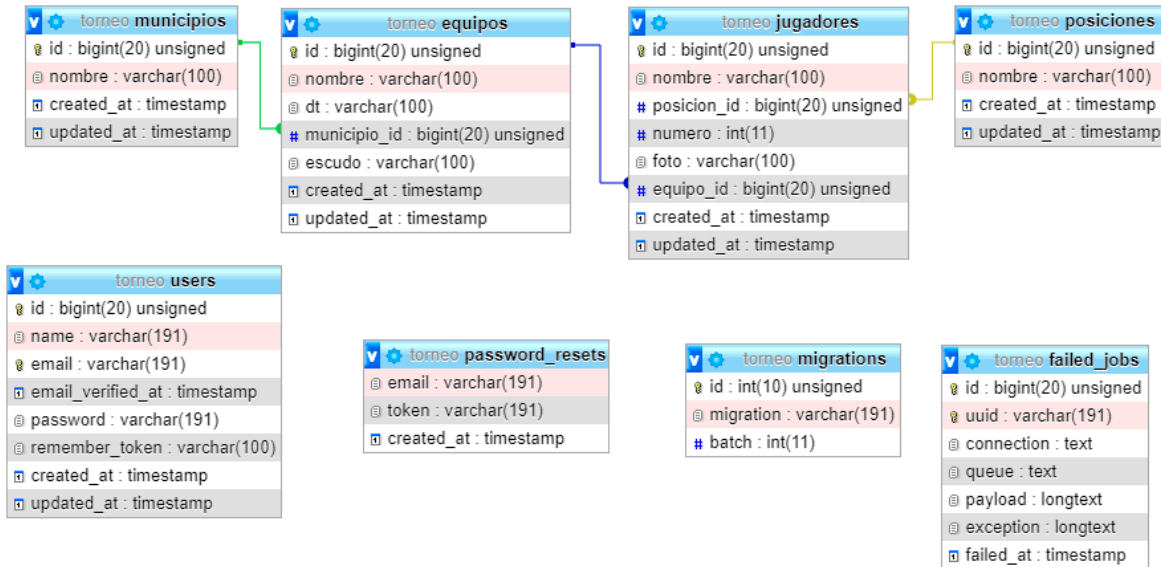
En cmd ejecutar el comando mostrado en la imagen.

```
php artisan migrate
```

Si todo salió bien, se muestra un mensaje similar al mostrado en la imagen.

```
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (704.46ms)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (520.09ms)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (517.93ms)
Migrating: 2021_04_18_005616_create_municipios_table
Migrated: 2021_04_18_005616_create_municipios_table (219.66ms)
Migrating: 2021_04_18_005638_create_equipos_table
Migrated: 2021_04_18_005638_create_equipos_table (1,396.19ms)
Migrating: 2021_04_18_005655_create_posiciones_table
Migrated: 2021_04_18_005655_create_posiciones_table (405.93ms)
Migrating: 2021_04_18_005720_create_jugadores_table
Migrated: 2021_04_18_005720_create_jugadores_table (4,406.74ms)
```

Al verificar la base de datos en el gestor de bases de datos, se puede verificar que se crearon las tablas municipios, equipos, posiciones y jugadores con sus respectivas relaciones, además se crearon cuatro tablas adicionales.



## Revertir migraciones

Para revertir la última migración, se usa el comando rollback, que regresa al último "batch" de migraciones.

```
php artisan migrate:rollback
```

El comando migrate:reset deshará todas las migraciones.

```
php artisan migrate:reset
```

El comando migrate:refresh revertirá todas tus migraciones y luego ejecutará el comando migrate

```
php artisan migrate:refresh
```

El comando migrate:fresh eliminará todas las tablas de la base de datos y luego ejecutará el comando migrate.

```
php artisan migrate:fresh
```

## Eloquent ORM

El mapeado objeto-relacional es una técnica de programación para convertir datos entre un lenguaje de programación orientado a objetos y una base de datos, permitiendo el uso de las características propias de la orientación a objetos.

Laravel incluye su propio ORM llamado Eloquent, que permite interactuar con la base de datos. A cada tabla se le define un modelo, para interactuar desde código con ella, permitiendo hacer CRUD sobre los datos.

### Definir modelos

Los modelos se ubican en `app\Models`.

Todos los modelos heredan de la clase

`Illuminate\Database\Eloquent\Model`

La forma más sencilla de crear un modelo es utilizando el comando.

```
php artisan make:model NombreModelo
```

Este comando creará el archivo dentro de la carpeta `app\Models` con el código básico de un modelo.

### Nombre del modelo

El nombre de los modelos corresponde al nombre de la tabla, pero en singular, y con la primera letra en mayúscula, mientras que el nombre de las tablas suele estar en plural. Al definir un modelo no es necesario indicar el nombre de la tabla asociada, sino que Eloquent automáticamente buscará la tabla, esto solo se da si el nombre de la tabla tiene una "s" adicional al nombre del modelo, en otro caso de debe hacer una modificación al modelo.

---

### Actividad 15

Crear los modelos Municipio, Equipo, Jugador y Posición, escribiendo en el cmd las instrucciones que se muestran en las imágenes.

```
php artisan make:model Municipio
```

```
php artisan make:model Equipo
```

```
php artisan make:model Jugador
```

```
php artisan make:model Posicion
```

Si todo ha salido bien. Se mostrará un mensaje como el de la imagen.

```
Model created successfully.
```

Y los archivos de los modelos estarán disponibles en la carpeta `app\Models`.

---

Laravel asumirá que los modelos creados anteriormente almacenaran datos en las tablas municipios, equipos, jugadores y posiciones.



Se puede especificar otra tabla definiendo la propiedad protegida `table`, en el modelo.

```
protected $table = 'my_flights';
```

Acción que se debe hacer en los modelos Jugador y Posición

---

### Actividad 16

Definir la tabla relacionada con el modelo Jugador, abriendo el archivo `app/Models/Jugador.php` y modificar el contenido con el código mostrado en la imagen.

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Jugador extends Model
{
    protected $table = 'jugadores';
}
```

Definir la tabla relacionada con el modelo Posicion, abriendo el archivo `app/Models/Posicion.php` y modificar el contenido con el código mostrado en la imagen.

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Posicion extends Model
{
    protected $table = 'posiciones';
}
```

---

### Primary Key

Eloquent asume que cada tabla tiene una llave primaria llamada `id`. Se puede cambiar ese valor sobrescribiendo la propiedad protegida `$primaryKey`.

```
protected $primaryKey = 'flight_id';
```

Eloquent supone que la llave primaria es un valor entero autoincrementable. Si se utiliza una llave primaria no autoincrementable o no numérica, se debe establecer la propiedad pública `$incrementing` a `false`.

```
public $incrementing = false;
```

Si la clave primaria no es un número entero, se debe establecer la propiedad protegida `$keyType` a "string"

```
protected $keyType = 'string';
```

Los modelos Eloquent no admiten claves primarias "compuestas".

## Timestamps

Por defecto Eloquent asume que todas las tablas contienen los campos `updated_at` y `created_at` (los cuales se añaden con `$table->timestamps()` en la migración). Estos campos se actualizan automáticamente cuando se cree o modifique un nuevo registro. Para no usarlos, se debe indicar en el modelo modificando el valor de la propiedad pública `$timestamps` a `false`.

```
public $timestamps = false;
```

## Relaciones

Las tablas de las bases de datos se relacionan a menudo unas con otras. Eloquent facilita la gestión y el trabajo con estas relaciones fácilmente soportando varios tipos de relaciones diferentes definiéndolas como métodos en los modelos.

- Uno a uno
- Uno a muchos
- Muchos a muchos

### Relación uno a uno

Para definir la relación entre un modelo `User` asociado con uno `Phone`, se crea un método `phone` en el modelo `User`, el cual llama al método `hasOne`, pasando como argumento el nombre del modelo relacionado y retornando el resultado.

El método `hasOne` está disponible en `Illuminate\Database\Eloquent\Model`

```
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{
    public function phone()
    {
        return $this->hasOne(Phone::class);
    }
}
```

Una vez la relación está definida, se puede acceder al registro usando las propiedades dinámicas de Eloquent, que permiten acceder a los métodos de la relación como si fueran propiedades del modelo.

```
$phone = User::find(1)->phone;
```

Eloquent determina la llave foránea de la relación basada en el nombre del modelo padre. En este caso se asume que el modelo `Phone` tiene una llave foránea `user_id`.

En caso de no ser así, se debe sobrescribir esta convención pasando un segundo argumento con el nombre del campo definido como llave foránea.

```
return $this->hasOne(Phone::class, 'foreign_key');
```

Adicionalmente, Eloquent asume que la llave primaria se llama id, en caso de no ser así, se debe pasar un tercer argumento, con el nombre de la llave primaria, al método hasOne.

```
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

### Relación uno a uno inversa

Para definir la relación inversa en la cual desde el modelo Phone se pueda acceder al User que posee el teléfono, se utiliza el método belongsTo.

```
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Phone extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Eloquent determina la llave foránea de la relación basada en el nombre del modelo padre. En este caso se asume que el modelo Phone tiene una llave foránea user\_id. En caso de no ser así, se debe sobrescribir esta convención pasando un segundo argumento con el nombre del campo definido como llave foránea.

```
return $this->belongsTo(User::class, 'foreign_key');
```

Adicionalmente, Eloquent asume que la llave primaria se llama id, en caso de no ser así, se debe pasar un tercer argumento, con el nombre de la llave primaria, al método belongsTo.

```
return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
```

### Relación uno a muchos

Una relación uno a muchos se usa para definir relaciones en las cuales un modelo único, que es el padre, posee cualquier cantidad de otros modelos. Por ejemplo, un blog puede tener un número infinito de comentarios. Esta relación se define con el método hasMany.

```
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Post extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Una vez que la relación ha sido definida, se puede acceder a la colección de comentarios desde la propiedad comments.

```
$comments = Post::find(1)->comments;
foreach ($comments as $comment) {
}
```

Como el método hasOne, se pueden sobrescribir la llave foránea y primaria pasando argumentos adicionales al método hasMany

```
return $this->hasMany(Comment::class, 'foreign_key');
return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

### Relación uno a muchos inversa

En la relación anterior es posible acceder a todos los comentarios del post, ahora se va a definir la relación para permitir que un comentario acceda a su publicación padre. Para definir el inverso de una relación hasMany, se define una función en el modelo hijo que llame al método belongsTo

```
use Illuminate\Database\Eloquent\Model;
class Comment extends Model
{
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Una vez la relación ha sido definida, se puede acceder a la publicación padre accediendo a la propiedad post.

```
$comment = Comment::find(1);
return $comment->post->title;
```

En caso de que la llave foránea y primaria no cumplan con la estructura de Laravel, se pueden sobrescribir estos valores como se explico anteriormente.

### Actividad 17

De acuerdo con el modelo relacional, crear las relaciones entre los modelos.

Modificar el modelo Municipio agregando la relación Municipio hasMany con Equipo.

```
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Municipio extends Model
{
    use HasFactory;
    public function equipos()
    {
        return $this->hasMany(Equipo::class);
    }
}
```

Modificar el modelo Equipo para agregar la relación, Equipo hasMany Jugador. Y Equipo belongsTo Municipio

```
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Equipo extends Model
{
    use HasFactory;
    public function jugadores()
    {
        return $this->hasMany(Jugador::class);
    }
    public function municipio()
    {
        return $this->belongsTo(Municipio::class);
    }
}
```

Modificar el modelo Jugador para agregar las relaciones, Jugador belongsTo Equipo y Jugador belongsTo Posición.

```
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Jugador extends Model
{
    use HasFactory;
    protected $table = 'jugadores';
    public function equipo()
    {
        return $this->belongsTo(Equipo::class);
    }
    public function posicion()
    {
        return $this->belongsTo(Posicion::class);
    }
}
```

Modificar el modelo Posicion agregando la relación Posición hasMany con Jugador.

```
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Posicion extends Model
{
    use HasFactory;
    protected $table = 'posiciones';
    public function Jugadores()
    {
        return $this->hasMany(Jugador::class);
    }
}
```

---

## Semillas

Laravel incluye una forma sencilla para agregar datos de prueba utilizando seed classes. Todas las semillas se almacenan en la carpeta database/seeder.

De manera predeterminada, se define una clase DatabaseSeeder, Desde la cual se puede llamar otras clases semillas, permitiendo controlar el orden de inicialización.

### Crear Seeders

Para la creación de las semillas se utiliza el comando

```
php artisan make:seeder UserSeeder
```

Una semilla solo contiene el método run, el cual se llama cuando se ejecuta el comando Artisan db:seed. Allí se pueden insertar los registros deseados, usando el generador de consultas o usando Eloquent model factories.

---

## Actividad 18

Crear las semillas MunicipioSeeder y PosicionSeeder, ejecutando los comandos mostrados en la imagen.

```
php artisan make:seeder MunicipioSeeder
```

```
php artisan make:seeder PosicionSeeder
```

Si todo salió bien, mostrara la confirmación.

```
Seeder created successfully.
```

---

A continuación, se crean los registros de las semillas a través de la creación de modelos.

---

## Actividad 19

Acceder al archivo database/seeder/MunicipioSeeder.php, y modificar su código para agregar los registros para los municipios Manizales, Pereira y Armenia, como se muestra en la imagen.

Tener en cuenta que es necesario agregar la referencia al modelo Municipio.

```
namespace Database\Seeders;
use Illuminate\Database\Seeder;
use App\Models\Municipio;
class MunicipioSeeder extends Seeder
{
    public function run()
    {
        $municipio = new Municipio();
        $municipio->nombre = "Manizales";
        $municipio->save();
        $municipio = new Municipio();
        $municipio->nombre = "Pereira";
        $municipio->save();
        $municipio = new Municipio();
        $municipio->nombre = "Armenia";
        $municipio->save();
    }
}
```

Acceder al archivo database/seeder/PosicionSeeder.php, y modificar su código para agregar los registros para las posiciones Portero, Defensa, Volante y Delantero, como se muestra en la imagen.

Tener en cuenta que es necesario agregar la referencia al modelo Posición.

```
namespace Database\Seeders;
use App\Models\Posicion;
use Illuminate\Database\Seeder;
class PosicionSeeder extends Seeder
{
    public function run()
    {
        $posicion = new Posicion();
        $posicion->nombre = "Portero";
        $posicion->save();
        $posicion = new Posicion();
        $posicion->nombre = "Defensa";
        $posicion->save();
        $posicion = new Posicion();
        $posicion->nombre = "Volante";
        $posicion->save();
        $posicion = new Posicion();
        $posicion->nombre = "Delfantero";
        $posicion->save();
    }
}
```

---

## Llamando semillas

Dentro de la clase DatabaseSeeder, se puede usar el método call para llamar otras seed classes. Como se muestra en la imagen.

```
public function run()
{
    $this->call([
        UserSeeder::class,
        PostSeeder::class,
        CommentSeeder::class,
    ]);
}
```

---

## Actividad 20

Acceder al archivo database/seeder/DatabaseSeeder.php, y modificar su código para agregar el llamado a MunicipiosSeeder y PosicionesSeeder, como se muestra en la imagen.

```
public function run()
{
    $this->call([
        MunicipioSeeder::class,
        PosicionSeeder::class
    ]);
}
```

---

## Ejecutar semillas

Una vez definidos los archivos de semillas, para agregar los datos semilla a las tablas, se debe usar el comando de Artisan:

```
php artisan db:seed
```

---

## Actividad 21

En cmd escribir y ejecutar el comando mostrado en la imagen.

```
php artisan db:seed
```

Se debe mostrar el mensaje de confirmación.

```
Database seeding completed successfully.
```



Al verificar en la tabla municipios, estos deben aparecer.

id	nombre	created_at	updated_at
1	Manizales	2021-04-25 02:52:58	2021-04-25 02:52:58
2	Pereira	2021-04-25 02:53:00	2021-04-25 02:53:00
3	Armenia	2021-04-25 02:53:00	2021-04-25 02:53:00

Al verificar en la tabla posiciones, estos deben aparecer.

id	nombre	created_at	updated_at
1	Portero	2020-04-30 01:13:00	2020-04-30 01:13:00
2	Defensa	2020-04-30 01:13:00	2020-04-30 01:13:00
3	Volante	2020-04-30 01:13:00	2020-04-30 01:13:00
4	Delantero	2020-04-30 01:13:00	2020-04-30 01:13:00

## Obtener modelos

Cada modelo Eloquent es un potente query builder que permite consultar la tabla de la base de datos asociada al modelo.

El método `all()` retornará todos los registros de la tabla asociada al modelo.

```
Flight::all()
```

Se puede usar el método `get` y añadir constraints a las consultas. Ambos retornan una colección.

```
$flights = Flight::where('active', 1)
    ->orderBy('name')
    ->take(10)
    ->get();
```

Esa colección se puede recorrer con un `foreach`.

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

## Actividad 22

A continuación, se van a cargar dinámicamente, desde la base de datos, los municipios que se mostrarán en el select de equipos.

En vista de que se va a trabajar con el modelo Municipio, es necesario incluir en la referencia al modelo Municipio en el controlador EquiposController.

```
use App\Models\Municipio;
```

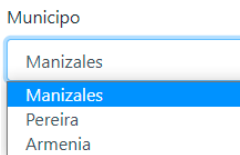
A continuación, se modifica el método create del controlador EquiposController, incluyendo el código mostrado en la imagen.

```
public function create()
{
    $municipios = Municipio::all();
    return view('equipos.create')
        ->with('municipios', $municipios);
}
```

Modificar la vista create de equipos reemplazando el código del select, de tal forma que cargue los option desde los datos enviados por el controlador.

```
<select class="form-control" id="municipio" name="municipio">
    @foreach($municipios as $municipio)
        <option value="{{ $municipio->id }}">{{ $municipio->nombre }}</option>
    @endforeach
</select>
```

Probar en el navegador.



## Tarea

Modificar el método create de JugadoresController y la vista create de jugadores, para que se carguen dinámicamente tanto los equipos como las posiciones de los jugadores.

El campo posición, debe mostrar la información de la tabla posiciones, cada option debe tener como value el id de la posición, y mostrar el nombre de la posición.

El campo equipo, debe mostrar la información de la tabla equipos, cada option debe tener como value el id del equipo, y mostrar el nombre del equipo.

## Request

Retomando el tema de formularios, para obtener una instancia de la solicitud HTTP actual se debe importar Illuminate\Http\Request en el controlador.

Para recuperar todos los datos de entrada se usa el método all

```
$request->all();
```

Con el método input es posible acceder al valor de un elemento específico del formulario.

```
$request->input("nombre");
```

También se puede acceder a los datos utilizando las propiedades dinámicas.

```
$nombre = $request->nombre;
```

### Archivos cargados

Se puede acceder a los archivos cargados desde una instancia Illuminate\Http\Request usando el método file que retorna una instancia de Illuminate\Http\UploadedFile.

Para determinar si un archivo está presente en la solicitud se utiliza el método hasFile.

También se puede verificar que no hubo problemas para cargar el archivo a través del método isValid

```
$file = $request->file('photo');
```

```
$file = $request->photo;
```

```
if ($request->hasFile('photo'))
```

```
if ($request->file('photo')->isValid())
```

### UploadedFile

La clase UploadedFile cuenta con otros métodos como.

getClientOriginalName(). Retorna el nombre original del archivo.

getClientOriginalExtension(). Retorna la extensión original del archivo.

getClientSize(). Retorna el tamaño del archivo.

move(\$directory, \$name = null). Mueve el archivo a una nueva posición.

### Insert

Para insertar un registro en la tabla de la base de datos asociada con un modelo simplemente se crea una instancia del modelo, asignar los valores y por último guardarlos con el método save().

Para usar un modelo, es necesario incluirlo en el controlador.

```
$user = new User;
```

```
$user->name = 'Juan';
```

```
$user->save();
```

Para obtener el identificador asignado en la base de datos después de guardar (cuando se trate de tablas con índice auto-incremental), se usa.

```
$insertedId = $user->id;
```

### Actividad 23

A continuación, se insertará un equipo en la base de datos con los datos y la imagen que llegan desde el formulario.

En vista de que se va a insertar un objeto de tipo Equipo, es necesario incluir en la referencia al modelo Equipo en el controlador EquiposController.

```
use App\Models\Equipo;
```

A continuación, se modifica el método store del controlador EquiposController, incluyendo el código mostrado en la imagen.

```
public function store(Request $request)
{
    if($request->hasFile('escudo')){
        $file = $request->file('escudo');
        $escudo = time() . $file->getClientOriginalName();
        $file->move("images/equipos", $escudo);
    }
    $equipo = new Equipo();
    $equipo->nombre = $request->nombre;
    $equipo->dt = $request->dt;
    $equipo->municipio_id = $request->municipio;
    $equipo->escudo = $escudo;
    $equipo->save();
    return "Guardado";
}
```

Insertar un equipo desde el formulario, con su respectivo escudo y verificar en la base de datos que el equipo se creó correctamente, y que en la carpeta public, exista una carpeta llamada images, dentro de ella una carpeta llamada equipos, y dentro de esta se encuentre el archivo cargado.

### Tarea

Modificar el método store de JugadoresController para que inserte el jugador en la base de datos y guarde la foto del jugador en una carpeta jugadores dentro de images.

## Validaciones

Laravel proporciona varias herramientas para validar los datos de entrada de la aplicación, la más común es el método `validate()`.

Para incluir la lógica de validación se utiliza el método `validate`. Si pasa la regla de validación, el código continuará ejecutándose, si no, se lanza una excepción y la respuesta se envía de vuelta al usuario.

Se pueden especificar múltiples reglas de validación con un carácter pipe, o pasar un arreglo con las reglas de validación deseadas.

```
'title' => 'required|unique:posts|max:255',
'title' => ['required', 'unique:posts', 'max:255'],
```

## Reglas de validación

La imagen muestra las reglas de validación que se pueden utilizar en Laravel

<a href="#">Accepted</a>	<a href="#">Distinct</a>	<a href="#">Nullable</a>
<a href="#">Active URL</a>	<a href="#">E-Mail</a>	<a href="#">Numeric</a>
<a href="#">After (Date)</a>	<a href="#">Exists (Database)</a>	<a href="#">Present</a>
<a href="#">After Or Equal (Date)</a>	<a href="#">File</a>	<a href="#">Regular Expression</a>
<a href="#">Alpha</a>	<a href="#">Filled</a>	<a href="#">Required</a>
<a href="#">Alpha Dash</a>	<a href="#">Greater Than</a>	<a href="#">Required If</a>
<a href="#">Alpha Numeric</a>	<a href="#">Greater Than Or Equal</a>	<a href="#">Required Unless</a>
<a href="#">Array</a>	<a href="#">Image (File)</a>	<a href="#">Required With</a>
<a href="#">Bail</a>	<a href="#">In</a>	<a href="#">Required With All</a>
<a href="#">Before (Date)</a>	<a href="#">In Array</a>	<a href="#">Required Without</a>
<a href="#">Before Or Equal (Date)</a>	<a href="#">Integer</a>	<a href="#">Required Without All</a>
<a href="#">Between</a>	<a href="#">IP Address</a>	<a href="#">Same</a>
<a href="#">Boolean</a>	<a href="#">JSON</a>	<a href="#">Size</a>
<a href="#">Confirmed</a>	<a href="#">Less Than</a>	<a href="#">Starts With</a>
<a href="#">Date</a>	<a href="#">Less Than Or Equal</a>	<a href="#">String</a>
<a href="#">Date Equals</a>	<a href="#">Max</a>	<a href="#">Timezone</a>
<a href="#">Date Format</a>	<a href="#">MIME Types</a>	<a href="#">Unique (Database)</a>
<a href="#">Different</a>	<a href="#">MIME Type By File Extension</a>	<a href="#">URL</a>
<a href="#">Digits</a>	<a href="#">Min</a>	<a href="#">UUID</a>
<a href="#">Digits Between</a>	<a href="#">Not In</a>	
<a href="#">Dimensions (Image Files)</a>	<a href="#">Not Regex</a>	

## Actividad 24

Modificar el método `store` de `EquiposController` para agregar las reglas para la validación de los campos, como se muestra en la imagen.

```
public function store(Request $request)
{
    $request->validate([
        'nombre' => 'required',
        'dt' => 'required',
        'municipio' => 'required',
        'escudo' => 'required|image'
    ]);
    if($request->hasFile('escudo')){
```

Al tratar de enviar el formulario sin alguno de los campos requeridos, o con un archivo que no sea de tipo imagen, este no se enviara.

---



---

## Tarea

Modificar el método store de JugadoresController para para agregar las reglas para la validación de los campos.

---

## Mostrar errores

Si no se cumplen las reglas de validación, Laravel redireccionará automáticamente al usuario a su ubicación anterior y todos los errores de validación se agregan automáticamente a la sesión, a través de la variable \$errors, que tiene diferentes métodos para obtener y mostrar los mensajes de error al usuario.

\$errors->any(). Devuelve verdadero si hay errores y falso si no.

\$errors->all(). Llamando a este método dentro de un foreach() se puede iterar a través de todos los mensajes.

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

## Directiva @error

Se puede usar la directiva de Blade @error para determinar si existen mensajes de validación para un atributo dado. dentro de la directiva @error se puede mostrar la variable \$message para mostrar el mensaje de error.

```
@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

## Actividad 25

Modificar la vista create, (resources/equipos/create.blade.php), para mostrar los errores, como se muestra en la imagen.

```
<div class="form-group">
    <label for="nombre">Nombre</label>
    <input type="text" name="nombre" id="nombre" class="form-control">
    @error('nombre')
        <small>{{ $message }}</small>
    @enderror
</div>
<div class="form-group">
    <label for="dt">D.T.</label>
    <input type="text" name="dt" id="dt" class="form-control">
    @error('dt')
        <small>{{ $message }}</small>
    @enderror
</div>
<div class="form-group">
    <label for="municipio">Municipio</label>
    <select class="form-control" id="municipio" name="municipio">
        @foreach($municipios as $municipio)
            <option value="{{ $municipio->id }}">{{ $municipio->nombre }}</option>
        @endforeach
    </select>
    @error('municipio')
        <small>{{ $message }}</small>
    @enderror
</div>
<div class="form-group">
    <label for="escudo">Escudo</label>
    <input type="file" name="escudo" id="escudo" class="form-control-file">
    @error('escudo')
        <small>{{ $message }}</small>
    @enderror
</div>
```

Al enviar el formulario sin algún campo, o con un archivo diferente a imagen, se mostrarán los mensajes indicando los errores de validación.

## Traducir los mensajes de error

Los mensajes de error de las reglas de validación están ubicados en el archivo resources/lang/en/validation.php.

Se puede crear otra carpeta y traducir los mensajes, cambiando la propiedad location en la carpeta config/app.php

---

### Actividad 26

Crear una copia de la carpeta `resources/lang/en` y cambiarle el nombre por `resources/lang/es`.

Ir al repositorio <https://github.com/Laraveles/spanish/tree/master/resources/lang/es> y reemplazar el contenido de los archivos, respectivamente.

Cambiar el valor de la propiedad local a 'es' en el archivo `config/app`.

```
'locale' => 'es',
```

Probar y verificar que los mensajes están en español.

---

### Repoblar formularios

Cuando se genera un redireccionamiento debido a un error de validación, Laravel envía los valores de los inputs de la solicitud a la sesión, con el fin de poblar el formulario con los datos que el usuario intentó enviar.

Para recuperar los datos de la solicitud anterior, se utiliza el método `old()` de `Illuminate\Http\Request`.

```
<input type="text" name="title" value="{{ old('title') }}">
```

---

### Actividad 27

Modificar la vista `create`, (`resources/equipos/create.blade.php`), para repoblar los campos `nombre` y `dt`, asignándoles los valores.

```
<input type="text" name="nombre" id="nombre" class="form-control" value="{{ old('nombre') }}">
```

```
<input type="text" name="dt" id="dt" class="form-control" value="{{ old('dt') }}">
```

---

### Tarea

Modificar la vista `create` de jugadores para repoblar los campos cuando se redirecciona el formulario.

---

### Form Request

Son clases que contienen la lógica de la validación. Se crean con el comando `Artisan make:request`.



```
php artisan make:request NombreFormRequest
```

La clase generada será guardada en `app/Http/Requests`.

Esta clase contiene un método `authorize`, donde se puede comprobar si el usuario autenticado tiene permiso para actualizar un recurso determinado.

Si la lógica de autorización se implementa en otra parte de la aplicación, se retorna `true`.

El método `rules` retorna un arreglo con las reglas de validación.

## Actividad 28

Crear el Form Request `StoreEquiposRequest`, ejecutando el comando mostrado en la imagen.

```
php artisan make:request StoreEquiposRequest
```

Si todo ha salido bien, se mostrará el mensaje de confirmación y habrá creado el archivo en la ruta `app/Http/Requests`

```
Request created successfully.
```

Al Form Request `StoreEquiposRequest` se le modificar el método `authorize` para que retorne `true`, y el método `rules`, para agregar las reglas de validación.

```
class StoreEquiposRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }
    public function rules()
    {
        return [
            'nombre' => 'required',
            'dt' => 'required',
            'pais' => 'required',
            'escudo' => 'required|image'
        ];
    }
}
```

Para utilizar el Form Request, se debe incluir en el controlador `EquiposController`.

```
use App\Http\Requests\StoreEquiposRequest;
```

También se debe reemplazar la clase `Request` que llega al método del controlador, por la clase creada, y eliminar las reglas de validación de este método.

```
public function store(StoreEquiposRequest $request)
{
    if($request->hasFile('escudo')){
```

Verificar que las reglas de validación continúen funcionando.

---

## Personalizar el mensaje

Se pueden personalizar los mensajes de error usados en la petición del formulario sobrescribiendo, en el Form Request, el método messages, el cual retorna una matriz de pares de atributo.regla y su correspondiente mensaje de error.

---

## Actividad 29

Modificar el Form Request, StoreEquiposRequest, creando el método messages, en el cual se incluirán los mensajes personalizados cuando hay errores de validación.

```
public function messages(){
    return[
        'nombre.required' => 'El nombre es obligatorio',
        'dt.required' => 'El D.T. es obligatorio',
        'pais.required' => 'El pais es obligatorio',
        'escudo.required' => 'El escudo es obligatorio',
        'escudo.image' => 'El escudo debe ser una imagen'
    ];
}
```

Verificar que los mensajes mostrados corresponden con los que se definieron cuando hay errores de validación.

---

## Redirect

Las redirecciones son instancias de la clase Illuminate\Http\RedirectResponse y contienen las cabeceras apropiadas para redirigir al usuario a otra URL, ruta o acción de un controlador.

Si la ruta tiene parámetros, se pasan como segundo argumento al método route.

Si el parámetro es un modelo, se pasa este.

```
return redirect('home/dashboard');
```

```
return redirect()->route('login');
```

```
return redirect()->route('profile', ['id' => 1]);
```

## Redirect con datos

La redirección a una nueva URL y flash de los datos a la sesión se hacen generalmente al mismo tiempo. Normalmente, después de realizar con éxito una acción como cuando se muestra un mensaje de éxito.

Después de redirigir al usuario, se puede mostrar el mensaje usando Sintaxis Blade.

```
return redirect('dashboard')->with('status', 'Profile updated!');
```

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

## Actividad 30

Modificar el método store de EquiposController, para que redirija a la ruta equipos.index, y envíe el mensaje del equipo creado. Como muestra la imagen.

```
$equipo->save();
return redirect()->route('equipos.index')->with('status', 'Equipo creado');
}
```

Modificar la vista index de equipos, (resources/views/Equipos/index.blade.php)

Para que muestre el mensaje, en caso de que identifique una variable de sesión llamada status.

```
@section('content')
    @if(session('status'))
        <div class="alert alert-success">
            {{ session('status') }}
        </div>
    @endif
    <div class="row">
```

Verificar que, al insertar un equipo, se muestra el listado de los equipos, y un mensaje confirmando su creación.

## Tarea

1. Crear un FormRequest para realizar las validaciones al formulario de creación de jugadores, para que todos los campos sean obligatorios.
2. Crear y mostrar mensajes de errores personalizados cuando no se cumplen todas las reglas de validación.
3. Redireccionar hacia el index de jugadores cuando se inserta un nuevo jugador, y mostrar el mensaje indicándolo.

### Actividad 31

A continuación, se cargarán los equipos desde la base de datos. Por lo tanto, se debe eliminar el arreglo `$arrayEquipos` del controlador `EquiposController`.

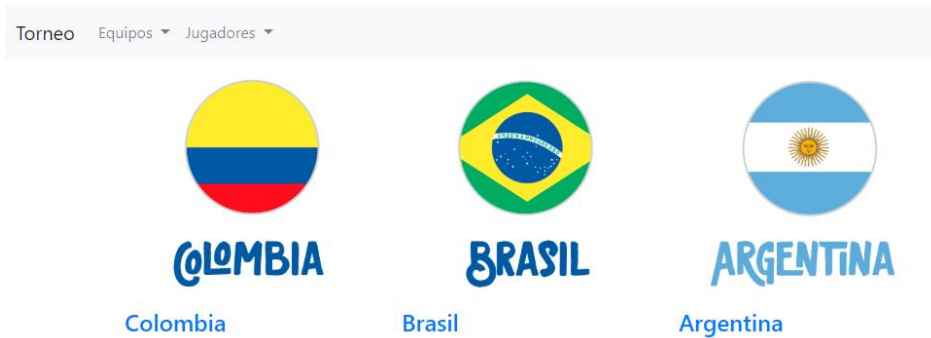
Modificar el método `index` de `EquiposController`, para solicitar los equipos y enviarlos a la respectiva vista, como se muestra en la imagen.

```
public function index()
{
    $equipos = Equipo::all();
    return view('Equipos.index')
        ->with('equipos', $equipos);
}
```

Modificar la vista `index` de `Equipos`, (`resources/views/Equipos/index.blade.php`), para cargar la información desde la colección que llego como parámetro.

```
@extends('layouts.base')
@section('title', 'Ver equipos')
@section('content')
    @if(session('status'))
        <div class="alert alert-success">
            {{ session('status') }}
        </div>
    @endif
    <div class="row">
        @foreach($equipos as $equipo)
            <div class="col-xs-6 col-sm-4 col-md-3 text-center">
                <a href="/equipos/{{ $equipo->id }}">
                    
                    <h4>{{ $equipo->nombre }}</h4>
                </a>
            </div>
        @endforeach
    </div>
@stop
```

Verificar en la aplicación que se cargan los equipos desde la base de datos, como se muestra en la imagen.



### Obtener modelo

Además de obtener todos los registros de una tabla, se pueden obtener registros únicos utilizando find. Este método busca y retorna el registro cuyo id se envía como parámetro.

```
$flight = App\Flight::find(1);
```

También se puede llamar al método find con un array de llaves primarias, que devolverá una colección de registros coincidentes.

```
$flights = App\Flight::find([1, 2, 3]);
```

Otro método es el first, Este retornan una instancia de un modelo único.

```
$flight = App\Flight::where('active', 1)->first();
```

### Actividad 32

A continuación, se va a agregar el código necesario para mostrar el detalle de cada equipo.

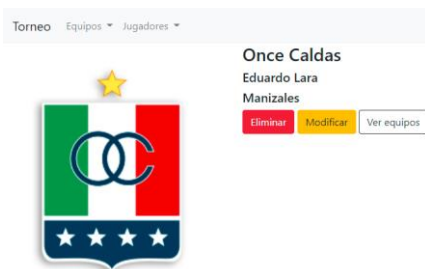
Lo primero que se debe hacer es modificar el método show de EquiposController, para solicitar, a la base de datos, el equipo, de acuerdo con el parámetro que llega por la URL, y ese equipo enviarlo a la vista.

```
public function show($id)
{
    $equipo = Equipo::find($id);
    return view('Equipos.show')
        ->with('equipo', $equipo);
}
```

Luego se debe modificar la vista show de Equipos, (resources/views/Equipos/show.blade.php), para que cargue la información del objeto que recibió como parámetro, como se muestra en la imagen.

```
@extends('layouts.base')
@section('title', 'Detalle equipo')
@section('content')
    <div class="row">
        <div class="col-sm-4">
            
        </div>
        <div class="col-sm-8">
            <p class="h3">{{ $equipo->nombre }}</p>
            <p class="h5">{{ $equipo->dt }}</p>
            <p class="h5">{{ $equipo->municipio->nombre }}</p>
            <a href="#" class="btn btn-danger">Eliminar</a>
            <a href="/equipos/{{ $equipo->id }}/edit" class="btn btn-warning">Modificar</a>
            <a href="/equipos" class="btn btn-outline-dark">Ver equipos</a>
        </div>
    </div>
@endsection
```



Verificar que, al dar clic en alguno de los equipos, se cargue el detalle de este, como muestra la imagen.



Además, verificar que al dar clic al botón Modificar, se muestre la vista relacionada, y al dar clic sobre Ver equipos, muestre de nuevo el listado de los equipos.

## Tarea

Mostrar el listado de los jugadores desde la base de datos. Como se muestra en la imagen.

Foto	Nombre	Posicion	Numero	Equipo	Accion
	James Rodriguez	Volante	3	Colombia	<a href="#">Ver</a> <a href="#">Modificar</a> <a href="#">Eliminar</a>
	David Ospina	Portero	1	Colombia	<a href="#">Ver</a> <a href="#">Modificar</a> <a href="#">Eliminar</a>

Mostrar el detalle de cada jugador al dar clic en el botón ver.

---

## Update

El método save también permite actualizar modelos existentes en la base de datos. Para actualizar un modelo, primero se debe recuperar, y luego modificar los atributos, para finalmente llamar el método save. Por su parte, el campo updated\_at se actualizará automáticamente.

```
$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();
```

---

## Actividad 33

Lo primero que se debe hacer es modificar el método edit de EquiposController para solicitar el equipo con el id recibido, para enviarlo a la vista edit, que muestra el formulario para modificar el equipo. Como se muestra en la imagen.

```
public function edit($id)
{
    $equipo = Equipo::find($id);
    $municipios = Municipio::all();
    return view('equipos.edit')
        ->with('equipo', $equipo)
        ->with('municipios', $municipios);
}
```

A continuación, se modifica la vista edit de equipos, (resources/views/Equipos/edit.blade.php), creando el formulario y cargando los datos que están almacenados en la base de datos y que llegan con el objeto.

Adicional a ello, el select de municipios debe dejar seleccionado el país que le corresponde al equipo.

En este caso se utiliza el método @method('PUT') para indicar que esa es la acción sobre el formulario. En la imagen se muestra el código de edit.blade.php.

```

@extends('layouts.base')
@section('title', 'Torneo')
@section('content')
<form action="/equipos/{{ $equipo->id }}" method="post" enctype="multipart/form-data">
    @method('PUT')
    @csrf
    <div class="form-group">
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" id="nombre" class="form-control" value="{{ $equipo->nombre }}">
    </div>
    <div class="form-group">
        <label for="dt">D.T.</label>
        <input type="text" name="dt" id="dt" class="form-control" value="{{ $equipo->dt }}">
    </div>
    <div class="form-group">
        <label for="municipio">Municipio</label>
        <select class="form-control" id="municipio" name="municipio">
            @foreach($municipios as $municipio)
                <option value="{{ $municipio->id }}" @if($municipio->id == $equipo->municipio_id) selected
                    @endif>{{ $municipio->nombre }}</option>
            @endforeach
        </select>
    </div>
    <div class="form-group">
        <label for="escudo">Escudo</label>
        <input type="file" name="escudo" id="escudo" class="form-control-file">
    </div>
    <button type="submit" class="btn btn-primary">Actualizar</button>
</form>
@endsection

```

A continuación, se modifica el método update del EquiposController, para que busque el equipo, obtenga el modelo y permita modificar sus atributos, además que actualice la imagen en la base de datos y guarde. El código se muestra en la imagen.

```

public function update(Request $request, $id)
{
    $equipo = Equipo::find($id);
    $equipo->nombre = $request->nombre;
    $equipo->dt = $request->dt;
    $equipo->municipio_id = $request->municipio;
    if($request->hasFile('escudo')){
        $file = $request->file('escudo');
        $escudo = $equipo->escudo;
        $file->move("images/equipos", $escudo);
    }
    $equipo->save();
    return redirect()->route('equipos.index')->with('status', 'equipo actualizado');
}

```

## Tarea

1. Crear un FormRequest para validar el nombre y el dt en el formulario de actualización, además los mensajes de error.
2. Hacer el mismo proceso para actualizar los jugadores. Es importante que el select del equipo y de la posición muestren seleccionado el equipo y la posición a la que pertenece cada jugador.



3. Crear un FormRequest y los mensajes de error de validaciones para el formulario actualizar jugador.

## Delete

Para eliminar un modelo, se debe llamar el método delete de su instancia.

Esto se hace desde el método destroy del controlador, pero para llegar a él, se debe hacer a través de un formulario con el método delete.

```
$flight = App\Flight::find(1);

$flight->delete();
```

## Actividad 34

A continuación, se realizará la eliminación de un equipo, lo primero que se debe hacer es modificar el método destroy de EquiposControler, para que cree el objeto con el equipo identificado con el id, luego se elimina con el método delete, y luego se redirecciona. En la imagen se muestra el código.

```
public function destroy($id)
{
    $equipo = Equipo::find($id);
    $equipo->delete();
    return redirect()->route('equipos.index')
        ->with('status', 'Equipo eliminado');
}
```

Teniendo en cuenta que se debe llegar al método destroy con una acción de tipo delete en la ruta, es necesario acceder a la ruta a través de un formulario con el método DELETE.

Por lo tanto, en la vista show de equipos, (resources/views/Equipos/show.blade.php), se debe reemplazar el botón Eliminar, que actualmente debe tener el siguiente código.

```
<a href="#" class="btn btn-danger">Eliminar</a>
```

Por el código que se muestra a continuación.

```
<form class="delete d-inline" action="/equipos/{{ $equipo->id }}" method="post">
    @method('DELETE')
    @csrf
    <button type="submit" class="btn btn-danger">Eliminar</button>
</form>
```

Comprobar que el equipo se eliminó.

En caso de querer eliminar el archivo, se puede utilizar el método delete del facade File. Por lo tanto, se hace referencia al facade File dentro del EquiposController, insertando la línea mostrada en la imagen.

```
use Illuminate\Support\Facades\File;
```

A continuación, se modifica el método destroy, insertando el código necesario para que elimine la imagen.

```
public function destroy($id)
{
    $equipo = Equipo::find($id);
    $equipo->delete();
    File::delete("images/equipos/".$equipo->escudo);
    return redirect()->route('equipos.index')
        ->with('status', 'Equipo eliminado');
}
```

Para incluir un mensaje de confirmación en JavaScript, se debe.

1. Crear el archivo script.js en public/js, con el siguiente código.

```
$(".delete").on("submit", function(){
    return confirm("¿Desea eliminar este equipo?");
});
```

2. Incluir el script al final de base.blade.php, antes de cerrar el body, y después de los scripts que incluyen Bootstrap.

```
<script type="text/javascript" src="{{ asset('js/script.js') }}"></script>
```

Comprobar el funcionamiento.



## Tarea

Realizar las modificaciones necesarias para eliminar los jugadores.

