Manual de programador

Responsable:
Jairo Alejandro Castañeda Pedraza.
Hans Arevalo Barrera.
Jimmy Alejandro plazas López.
Karen Juliana Cano Vega.
Miler Yesid Menjure Barrera.

Contenido

INTRODUCCIÓN

En este anexo se presenta la explicación detallada del código fuente del aplicativo , a partir del modelo de la base de datos para una contextualización clara y después la estructura del proyecto , implementación de migraciones ,modelos , Controladores, rutas y vistas, todo con el soporte y definiciones que brinda la documentación oficial del framework utilizado para el back-end el cual fue LARAVEL en su versión 5.6 , el gestor de bases de datos MYSQL y VueJs como framework de Font-end, también especificar la manera con la que se trabajan con diferentes dependencias externas para funcionalidades especificas del aplicativo como lo son la generación de reportes PDF, generación de Gráficos ,entre otros.

OBJETIVOS

Para la realización de este documento se trazaron ciertos objetivos con el fin de cumplir en todo sentido con la documentación del proyecto, en este caso elaborado por parte del desarrollador quien entiende todo el código inmerso en el aplicativo.

OBJETIVO GENERAL

Presentar un documento realizado por el mismo desarrollador donde se explique de la mejor manera la implementación de las tecnologías utilizadas para el desarrollo del aplicativo con el fin de que en un futuro pueda ser cambiado el código ya sea por una actualización de versiones, cambio de requisitos, inclusión de nuevas funcionalidades u optimización de algunos métodos.

OBJETIVOS ESPECÍFICOS

- Documentar las principales líneas de código para que quien revise el código entienda su razón de ser y la forma en que se utilizó.
- Aclarar el procedimiento (paso a paso) que realiza un método desde cuando recibe los parámetros, consulta y hace operaciones hasta que retorna un resultado.
- Hacer saber en cierta forma el orden en el que se realizó el proyecto principalmente la parte de la lógica (back-end) y funcionamiento.
- Mostrar cómo se hizo efectiva la reutilización de código en este proyecto.

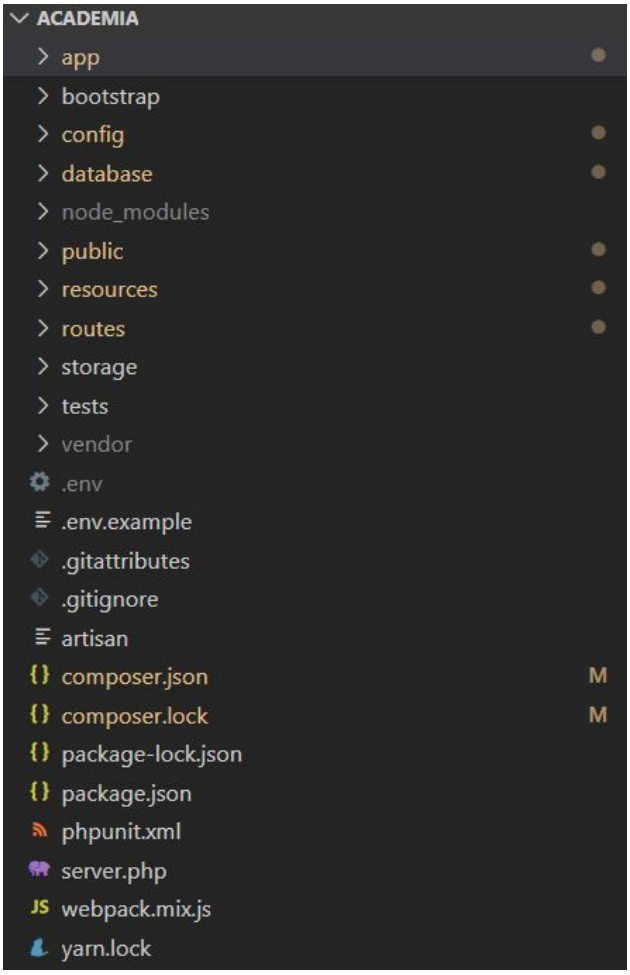
MODELO DE LA BASE DE DATOS

ESTRUCTURA DEL PROYECTO

El proyecto se creó según la documentación oficial de laravel, en donde se utilizó el gestor de paquetes ***compose***r y se ejecutó el siguiente comando con el fin de usar la versión 5.6 del framework la cual es muy estable: ***composer create-project --prefer-dist laravel/laravel academia "5.6.*"***

Para más información revisar la documentación oficial de la estructura de un proyecto de laravel en el siguiente link: https://documentacion-laravel.com/structure.html.

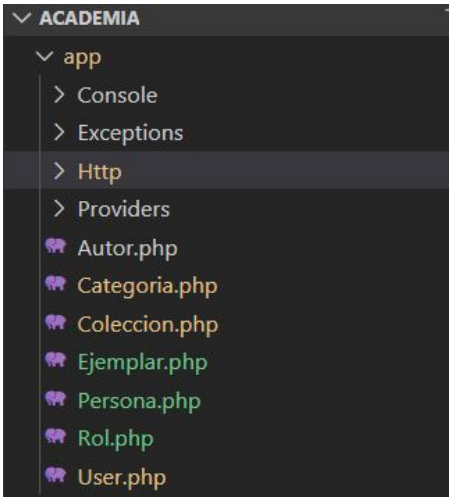
La estructura generada por el framework es la siguiente:



Se generan carpetas y ficheros cada una con un contexto particular en donde van diferentes tipos de componentes y particularidades que usa el framework, en su mayoría con archivos finales de ejemplo y a su vez una guía de en dónde encontrarlos.

DIRECTORIO APP

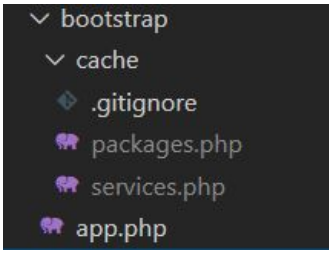
App es usado para ofrecer un hogar por defecto a todo el código personal de tu proyecto. Eso incluye clases que puedan ofrecer funcionalidad a la aplicación, archivos de configuración y más. Es considerado el directorio más importante de nuestro proyecto ya que es en el que más trabajaremos.



El directorio app tiene a su vez otros subdirectorios importantes pero uno de los más utilizados es el directorio **Http** en el cuál se ubican los ***Controllers, Middlewares y Requests*** en sus carpetas correspondientes, también a nivel de la raíz del directorio app se encuentran los modelos.

DIRECTORIO BOOTSTRAP

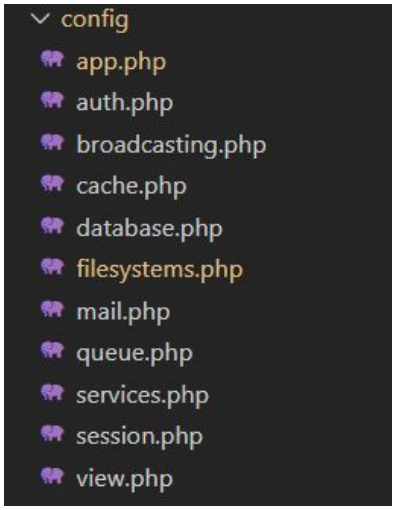
El directorio ***bootstrap*** contiene el archivo ***app.php*** que maqueta el framework.



Este directorio también almacena un directorio cache que contiene archivos generados por el framework para optimización de rendimiento como los archivos de cache de rutas y servicios.

DIRECTORIO CONFIG

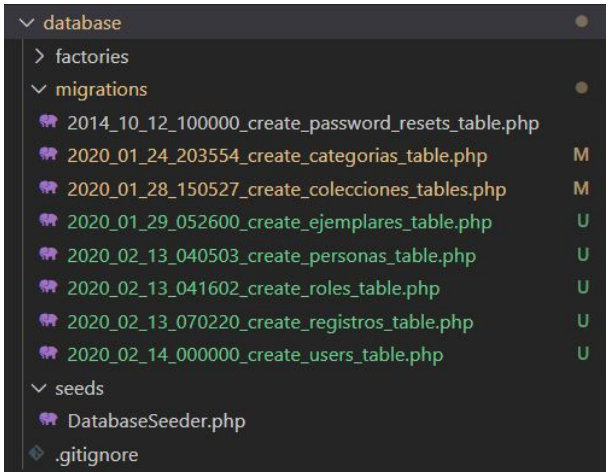
Este directorio contiene todos los archivos de configuración de la aplicación. Es una buena idea leer todos estos archivos y familiarizarte con todas las opciones disponibles.



- ***app.php***: En este archivo nos puede interesar configurar el lenguaje de nuestra aplicación, la zona horaria, los providers y aliases de las clases más comunes.
- ***database.php*** : En este archivo podemos configurar principalmente el motor de base de datos al cuál deseamos conectarnos.

DIRECTORIO DATABASE

Aquí se encuentran los archivos relacionados con el manejo de la base de datos, también se recomienda usar este directorio para almacenar una base de datos SQLite si es necesario.

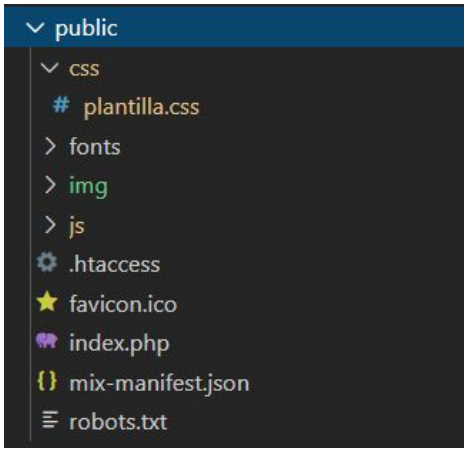


Dentro de este directorio se encuentran los subdirectorios:

- ***factories***: Aquí se escriben los model factories.
- ***Migrations***: Todas las migraciones que creamos se ubican en este subdirectorio, las cuales se nombran según la fecha de creación para cuando se ejecuten todas se haga de forma cronológica.
- ***seeds***: Contiene todas las clases de tipo seed.

DIRECTORIO PUBLIC

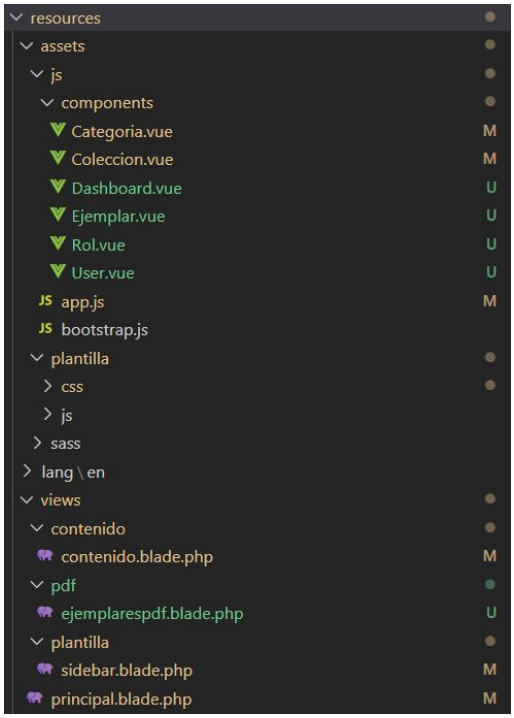
Dentro de este directorio van todos los recursos estáticos de la aplicación, tales como JavaScript, CSS e imágenes y fuentes.



Las buenas prácticas recomiendan crear una carpeta por cada tipo de recurso.

DIRECTORIO RESOURCES

Contiene las vistas así como también tus assets sin compilar tales como LESS, Sass o JavaScript.



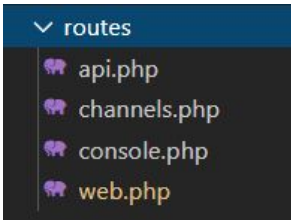
Dentro de este directorio se encuentran los subdirectorios:

- ***assets***: Aquí se ubican todos los archivos less de la aplicación.

- **lang**: Aquí se encuentran todos los archivos de internacionalización, es decir, los archivos para poder pasar nuestro proyecto de un idioma a otro. Normalmente habrá una carpeta por cada idioma, ejemplo:
- ***views***: Aquí se ubican las vistas en formato php o php.blade, es recomendable crear una carpeta por cada controlador, además agregar una carpeta **templates** para las plantillas. Una plantilla es una vista general, que tiene segmentos que pueden ser reemplazados mediante la herencia de plantillas, más adelante se hablará de este tema.

DIRECTORIO ROUTES

Este directorio contiene todas las definiciones de rutas para la aplicación. Por defecto, algunos archivos de rutas son incluidos con Laravel:



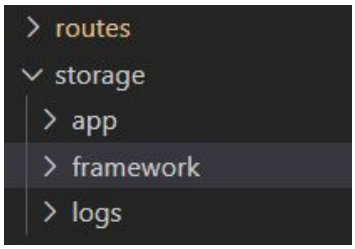
Las rutas son una de las funcionalidades que Laravel ofrece como carta de presentación y sin duda alguna es muy llamativa. Las rutas se definen con distintos métodos HTTP, grupos de rutas, dividirlas, etc.

El archivo ***web.php*** contiene rutas que ***RouteServiceProvider*** coloca en el grupo de middleware web, que proporciona estado de sesión, protección CSRF y encriptación de cookies. En la aplicación todas las rutas con definidas en el archivo ***web.php***.

El archivo ***api.php*** contiene rutas que ***RouteServiceProvider*** coloca en el grupo de middleware api, que proporcionan limitación de velocidad. Estas rutas están pensadas para no tener estado, así que las solicitudes que llegan a la aplicación a través de estas rutas están pensadas para ser autenticadas mediante tokens y no tendrán acceso al estado de sesión.

DIRECTORIO STORAGE

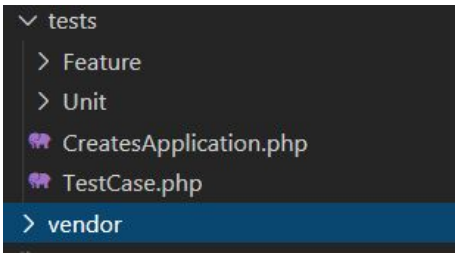
Cuando Laravel necesita escribir algo en el disco, lo hace en el directorio storage. Por este motivo, el servidor web debe poder escribir en esta ubicación.



También contiene las plantillas compiladas de ***Blade***, sesiones basadas en archivos, archivos de caches y otros archivos generados por el framework.

DIRECTORIO TESTS Y VENDOR

En el directorio ***tests*** se escriben los archivos de pruebas que serán ejecutadas posteriormente por ***phpunit*** y en el directorio ***vendor*** se almacenan las dependencias de Composer.



MIGRACIONES

Las migraciones son como el control de versiones de su base de datos, lo que permite a su equipo modificar y compartir fácilmente el esquema de la base de datos de la aplicación. Las migraciones generalmente se combinan con el generador de esquemas de Laravel para construir fácilmente el esquema de la base de datos de su aplicación. Para más información revisar la documentación oficial en el link que puede observar a continuación: https://laravel.com/docs/5.6/migrations.

Para la creación de las migraciones se utilizó la convención que recomienda laravel para nombrar cada migración con el fin de seguir el estándar y las buenas prácticas. En la

siguiente tabla se exponen los comandos usados durante la implementación de las migraciones.

Acción	Comando
Crear migración	***php artisan make:migration create_categorias_table***
Ejecutar migraciones	***php artisan migrate***
Borrar las tablas creadas y ejecutar de nuevo todas las migraciones	***php artisan migrate:refresh***

MIGRACIÓN TABLA CATEGORÍAS

En la siguiente imagen se observa la migración ya terminada con los atributos correspondientes a la entidad *'Categoria'* los cuales forman parte de la tabla '*categorias*' que se creara al momento de ejecutar la migración.

```
class CreateCategoriasTable extends Migration
{
    public function up()
    {
        Schema::create('categorias', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nombre',45);
            $table->string('descripcion',256);
            $table->boolean('condicion')->default(1);
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('categorias');
    }
}
```

La comando ***timestamps()*** hace referencia a una marca de tiempo que nos ofrece el framework en donde se crean los campos ***created_at*** que guarda la fecha de creación del registro y ***updated_at*** que guarda la fecha de la última modificación del registro.

MIGRACIÓN TABLA COLECCIONES

En la siguiente imagen se observa la migración ya terminada con los atributos correspondientes a la entidad *'Coleccion'* los cuales forman parte de la tabla '*colecciones*' que se creara al momento de ejecutar la migración.

```php
class CreateColeccionesTables extends Migration
{
    public function up()
    {
        Schema::create('colecciones', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nombre',45);
            $table->string('descripcion',256);
            $table->boolean('condicion')->default(1);
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('colecciones');
    }
}
```

MIGRACIÓN TABLA EJEMPLARES

En la siguiente imagen se observa la migración ya terminada con los atributos correspondientes a la entidad *'Ejemplar'* los cuales forman parte de la tabla '*ejemplares*' que se creara al momento de ejecutar la migración.

```php
class CreateEjemplaresTable extends Migration
{
    public function up()
    {
        Schema::create('ejemplares', function (Blueprint $table) {
            $table->increments('id');
            $table->string('titulo')->unique();
            $table->string('descripcion');
            $table->boolean('elaborado')->default(0);
            $table->string('editorial')->nullable();;
            $table->string('fecha_publicacion')->nullable();;

            $table->string('cantidad');
            $table->string('imagen',300);
            $table->boolean('condicion')->default(1);
            $table->string('autor',300);

            $table->integer('idcategoria')->unsigned();
            $table->foreign('idcategoria')->references('id')->on('categorias')->onDelete('cascade');

            $table->integer('idcoleccion')->unsigned();
            $table->foreign('idcoleccion')->references('id')->on('colecciones');

            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('ejemplares');
    }
}
```

Esta migración es más compleja que las anteriores debido a que trabaja bajo la cardinalidad uno a muchos con la tabla categorías y de la misma manera con la tabla colecciones.

MIGRACIÓN TABLA ROL

En la siguiente imagen se observa la migración ya terminada con los atributos correspondientes a la entidad *'Rol'* los cuales forman parte de la tabla '*roles*' que se creara al momento de ejecutar la migración.

```php
class CreateRolesTable extends Migration
{
    public function up()
    {
        Schema::create('roles', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nombre', 30)->unique();
            $table->string('descripcion', 200)->nullable();
            $table->boolean('condicion')->default(1);
        });
        DB::table('roles')->insert(array('id'=>'1','nombre'=>'Administrador', 'descripcion'=>'Maneja el CRUD de categorías
        ,colecciones ,ejemplares y usuarios, además tiene acceso a la tabla de roles y reportes , puede generar informes y ver
        gráficos estadísticos'));
        DB::table('roles')->insert(array('id'=>'2','nombre'=>'Secretario'   , 'descripcion'=>'Maneja el CRUD de categorías
        ,colecciones y ejemplares, a su vez puede ver la tabla de roles y ver gráficos estadísticos'));
    }
    public function down()
    {
        Schema::dropIfExists('roles');
    }
}
```

Como se pudo observar la tabla ya tiene predefinidos dos registros los cuales son ***Administrador*** y ***Secretario***, por ende esta tabla no necesita más registros porque solo se va a trabajar con estos dos roles, en definitiva la tabla quedaría estática.

MIGRACIÓN TABLA PERSONAS

En la siguiente imagen se observa la migración ya terminada con los atributos correspondientes a la entidad *'Persona'* los cuales forman parte de la tabla '*personas*' que se creara al momento de ejecutar la migración.

```php
class CreatePersonasTable extends Migration
{
    public function up()
    {
        Schema::create('personas', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nombres', 100);
            $table->string('apellidos', 70)->nullable();
            $table->string('celular',10)->nullable();
            $table->string('email', 20)->nullable();
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('personas');
    }
}
```

MIGRACIÓN TABLA USUARIOS

En la siguiente imagen se observa la migración ya terminada con los atributos correspondientes a la entidad *'**Usuario**'* según el modelo establecido de la base de datos , pero como laravel ya tiene esta migración se utilizó la que ya estaba, llamada '***Users***' la que al ser ejecutada crea la tabla '***users***'.

```php
class CreateUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->integer('id')->unsigned();
            $table->foreign('id')->references('id')->on('personas')->onDelete('cascade');

            $table->string('usuario')->unique();
            $table->string('password');
            $table->boolean('condicion')->default(1);

            $table->integer('idrol')->unsigned();
            $table->foreign('idrol')->references('id')->on('roles');

            $table->rememberToken();
        });
    }
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Como se observó en la imagen anterior es necesario acceder a las llaves primarias de las tablas ***'personas'*** y ***'roles'*** debido a las relaciones que se tienen entre estas.

MIGRACIÓN TABLA REGISTROS

En la siguiente imagen se observa la migración ya terminada con los atributos correspondientes a la entidad *'Registro'* los cuales forman parte de la tabla '*registros*' que se creará al momento de ejecutar la migración.

```php
class CreateRegistrosTable extends Migration
{
    public function up()
    {
        Schema::create('registros', function (Blueprint $table) {
            $table->increments('id');

            $table->integer('idpersona')->unsigned();
            $table->foreign('idpersona')->references('id')->on('personas')->onDelete('cascade');

            $table->integer('idejemplar')->unsigned();
            $table->foreign('idejemplar')->references('id')->on('ejemplares')->onDelete('cascade');

            $table->string('tema');

            $table->rememberToken();
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('registros');
    }
}
```

MODELOS

El ORM Eloquent incluido con Laravel proporciona una implementación de ActiveRecord fácil de entender y simple para trabajar con la base de datos. Cada tabla de base de datos tiene un "Modelo" correspondiente que se utiliza para interactuar con esa tabla. Los modelos le permiten consultar datos en sus tablas, así como insertar nuevos registros en la tabla. Para comenzar es vital configurar una conexión de base de datos en el archivo ***'.env'*** del proyecto. Como existen diferentes variantes según cada proyecto es necesario obtener más información sobre la configuración de su base de datos, consulte la documentación oficial de laravel en el siguiente link: https://laravel.com/docs/5.6/eloquent#defining-models.

.

Para la creación de los modelos se utilizó la convención que recomienda laravel para nombrar cada modelo con el fin de seguir el estándar y las buenas prácticas. En la siguiente tabla se exponen los comandos usados durante la implementación de los modelos.

Acción	Comando
Crear modelo	***php artisan make:model Categoria***

El modelo hace referencia a una clase php en donde van los atributos de la entidad y ciertos parámetros que son o no necesarios de inicializar según se requiera.

MODELO CATEGORÍA

Como en el modelo de la base de datos está estipulado que una categoría puede tener muchos ejemplares aparte de los atributos de esta entidad se debe implementar un método para que el framework entienda esta relación, el método 'ejemplares' hace referencia a dicha relación.

```php
class Categoria extends Model
{
    protected $fillable = ['nombre','descripcion','condicion'];

    public function ejemplares(){
        return $this->hasMany('App\Ejemplar');
    }
}
```

También es pertinente aclarar que no se declara el atributo 'id' porque como es un campo tipo entero auto incremental siempre el gestor de base de datos sabrá cuál es su valor.

MODELO COLECCION

Como en el modelo de la base de datos está estipulado que una colección puede tener muchos ejemplares aparte de los atributos de esta entidad se debe implementar un método para que el framework entienda esta relación, el método 'ejemplares' hace referencia a dicha relación.

```php
class Coleccion extends Model
{
    protected $table = 'colecciones';
    protected $fillable = ['nombre','descripcion','condicion'];

    public function ejemplares(){
        return $this->hasMany('App\Ejemplar');
    }
}
```

También es pertinente aclarar que no se declara el atributo 'id' porque como es un campo tipo entero auto incremental siempre el gestor de base de datos sabrá cuál es su valor. Por otra parte como en esta entidad no se usó la convención de laravel la cual indica que la tabla debe ser el mismo nombre del modelo en plural, hay que especificarle con la propiedad ***'$table'*** que la tabla es 'colecciones'.

MODELO EJEMPLAR

Como en el modelo de la base de datos está estipulado que un ejemplar tiene una categoría y una colección aparte de los atributos de esta entidad se deben solicitar los campos (id) de las llaves foráneas para poder relacionar las tres tablas.

```php
class Ejemplar extends Model
{
    protected $table = 'ejemplares';
    protected $fillable = [
        'autor',
        'titulo',
        'descripcion',
        'elaborado',
        'editorial',
        'fecha_publicacion',
        'cantidad',
        'imagen',
        'condicion',
        'idcategoria',
        'idcoleccion'
    ];
    public function categoria(){
        return $this->belongsTo('App\Categoria');
    }
    public function coleccion(){
        return $this->belongsTo('App\Coleccion');
    }
```

En este modelo se especifica con los métodos 'categoría' y 'colección' que se relaciona con estas entidades y que estas pertenecen a (***belongsTo***) los modelos especificados en cada método.

También es pertinente aclarar que no se declara el atributo 'id' porque como es un campo tipo entero auto incremental siempre el gestor de base de datos sabrá cuál es su valor. Por otra parte como en esta entidad no se usó la convención de laravel la cual indica que la tabla debe ser el mismo nombre del modelo en plural, hay que especificarle con la propiedad ***'$table'*** que la tabla es 'colecciones'.

MODELO ROL

Como en el modelo de la base de datos está estipulado que un rol puede tener muchos usuarios aparte de los atributos de esta entidad se debe implementar un método para que el framework entienda esta relación, el método '***users***' hace referencia a dicha relación.

```php
class Rol extends Model
{
    protected $table ='roles';
    protected $fillable=['nombre','descripcion','condicion'];

    public $timestamps =false;

    public function users(){
        return $this->hasMany('App\User');
    }
}
```

Cabe aclarar que no se declara el atributo ‘id’ porque como es un campo tipo entero auto incremental siempre el gestor de base de datos sabrá cuál es su valor. Por otra parte como en esta entidad no se usó la convención de laravel la cual indica que la tabla debe ser el mismo nombre del modelo en plural, hay que especificarle con la propiedad ***‘$table’*** que la tabla es ‘roles’ y como no se usan marcas de tiempo se debe especificar que la propiedad timestamps es falsa.

MODELO PERSONA

Según el modelo de la base de datos una persona solo puede tener un usuario aparte de los atributos de esta entidad se debe implementar un método para que el framework entienda esta relación, el método ‘***user***’ hace referencia a dicha relación.

```php
class Persona extends Model
{
    protected $fillable = ['nombres','apellidos','celular','email'];

    public function user(){
        return $this->hasOne('App\User');
    }
}
```

En este caso no se tuvo que especificar la tabla debido a que se cumple la convención que trabaja laravel.

MODELO USUARIO

Para este modelo se declaró cada atributo de la entidad y como se va a utilizar la llave primaria del rol se estableció la relación rol con el método ‘***rol’***, algo particular de este modelo es que se va a utilizar el mismo ‘***id’*** con el ‘***id’*** de la tabla ‘***personas’*** pero igualmente para acceder a este campo se implementó el método ‘***persona’***

```php
class User extends Authenticatable
{
    use Notifiable;

    protected $fillable = [
        'id',
        'usuario',
        'password',
        'idrol'
    ];
    public $timestamps =false;

    protected $hidden = [
        'password', 'remember_token',//protegiendo el password
    ];
    public function rol(){
        return $this->belongsTo('App\Rol');
    }
    public function persona(){
        return $this->belongsTo('App\Persona');
    }
}
```

El rasgo ***Notifiable*** es utilizado por defecto en el modelo y contiene un método que puede ser utilizado para enviar notificaciones, recomendado su uso por laravel para un proyecto escalable con usuarios. Para más información visitar el siguiente link: https://laravel.com/docs/5.8/notifications.

Como no se utilizó ninguna marca de tiempo para este registro la propiedad timestamps se dejó falsa, también se implementó la propiedad '***hidden'*** la cual se encarga de no mostrar de ninguna manera la contraseña (password).

CONTROLADORES

Son un mecanismo que nos permite agrupar la lógica de peticiones HTTP relacionadas y de esta forma organizar mejor nuestro código.

Acción	Comando
Crear controlador	***php artisan make:controller Categoria***

En lugar de definir en su totalidad la lógica de las peticiones en el archivo ***routes.php***, es posible que desee organizar este comportamiento usando clases tipo Controller. Los Controladores pueden agrupar las peticiones HTTP relacionada con la manipulación lógica en una clase. Los Controladores normalmente se almacenan en el directorio de aplicación app/Http/Controllers/.

Un ***controller*** usualmente trabaja con las peticiones:

GET, POST, PUT, DELETE y PATCH.

Asociando los métodos de la siguiente forma para los métodos más comunes:

- GET: index, create, show y edit.
- POST: store.
- PUT: update.
- DELETE: destroy.
- PATCH: update.

Como los métodos que están siendo implementados son trabajados por medio de Ajax, lo que le llega a cada método es un objeto 'Request' el cual contiene todos los parámetros que se le han enviado desde la vista, a cada método se le envían los parámetros correspondientes.

En cada controlador se deben incluir los modelos que se van a utilizar, por ejemplo en el '***CategoriaController'*** se van a instanciar objetos de tipo Categoria, por ende es necesario tener una referencia del modelo , ejemplo:

```
use App\Categoria;
```

Las Facades proveen una interfaz "estática" a las clases disponibles en el contenedor de servicios de la aplicación. Laravel viene con numerosas facades, las cuales brindan acceso a casi todas las características de Laravel. Las facades de Laravel sirven como "proxies estáticas" a las clases subyacentes en el contenedor de servicios, brindando el beneficio de una sintaxis tersa y expresiva, manteniendo mayor verificabilidad y flexibilidad que los métodos estáticos tradicionales .En algunos controladores es necesario importar este servicio, se implementa mediante este código:

```
use Illuminate\Support\Facades\DB;
```

Para más información visitar la documentación oficial en el siguiente link: https://documentacion-laravel.com/facades.html.

Para obtener una instancia de la solicitud HTTP actual a través de la inyección de dependencia, se debe instanciar la clase en el método de controlador mediante el siguiente código:

```
use Illuminate\Http\Request;
```

Por seguridad e integridad de la información almacenada en la base de datos fue necesario la implementación de una línea de código en todos los controladores, la cual valida la funcionalidad de cada método solo y exclusivamente si se hace desde una petición Ajax, de lo contrario redireccionará a la vista principal, la línea de código es la siguiente:

```
if (!$request->ajax()) return redirect('/');
```

Probablemente se van a utilizar más instancias en otros controladores pero las más comunes son estas y están presentes en todos.

Es muy importante aclarar que todas las validaciones para el ingreso de campos de un registro se validan desde el **FRONT-END** por ende no va a existir ningún tipo de manejo de cadenas, comparaciones ni evaluación de entradas en los controladores, los datos cuando llegan a los controladores corresponden al formato que se tiene en la base de datos.

CONTROLADOR CATEGORÍA

Para este controlador fue necesaria la implementación de seis métodos para el óptimo funcionamiento del CRUD y parte visual de objetos tipo '***Categoría'*** en otras vistas.

MÉTODO MOSTRAR

Este método se encarga de consultar todos los registros de la tabla ***'categorías'*** en donde se le envía con Ajax como parámetro un campo a buscar y un criterio de búsqueda el cual hace referencia a un campo de la tabla.

```
class CategoriaController extends Controller
{
    public function index(Request $request)
    {
        if (!$request->ajax()) return redirect('/');

        $buscar = $request->buscar;
        $criterio = $request->criterio;

        if ($buscar==''){
            $categorias = Categoria::orderBy('id', 'desc')->paginate(10);
        }
        else{
            $categorias = Categoria::where($criterio, 'like', '%'. $buscar . '%')->orderBy('id', 'desc')->paginate(10);
        }
        return [
            'pagination' => [
                'total'        => $categorias->total(),
                'current_page' => $categorias->currentPage(),
                'per_page'     => $categorias->perPage(),
                'last_page'    => $categorias->lastPage(),
                'from'         => $categorias->firstItem(),
                'to'           => $categorias->lastItem(),
            ],
            'categorias' => $categorias
        ];
    }
```

El método retorna un arreglo de tipo ***Categoria*** con todos los registros en grupos de a diez ordenados por ***'id'*** de manera descendente utilizando el método ***pagínate*** para que se facilite mostrar, todo si en el parámetro buscar no se le envía nada, si el parámetro buscar contiene algo se hará una búsqueda con ***Eloquent-ORM*** con doble comodín ***('%'***) para buscar coincidencias en cualquier parte del registro.

En el retorno se envían dos parámetros necesarios a la vista, el primero es '***pagination***' el cual tiene varios elementos necesarios para el funcionamiento de la paginación y el segundo es el arreglo '***categorias'***.

MÉTODO ALMACENAR

Este método recibe todos los campos requeridos para crear un nuevo registro, se crea un objeto de tipo 'Categoria' y se le asigna valor a cada uno de sus atributos, la condición por defecto es '1' debido a que lo más lógico es que cuando se ingrese un registro es porque está disponible.

```
public function store(Request $request)
{
    if (!$request->ajax()) return redirect('/');
    $categoria = new Categoria();
    $categoria->nombre = $request->nombre;
    $categoria->descripcion= $request->descripcion;
    $categoria->condicion = '1';
    $categoria->save();
}
```

Luego mediante ***Eloquent-ORM*** se guarda en la base de datos con el método ***save.***

MÉTODO ACTUALIZAR

Para la actualización de un registro se utiliza el método '***findOrFail'*** el cual busca el registro por el 'id', si lo encuentra creará un objeto con sus datos y se le asignarán los datos enviados desde Ajax, si no generará una excepción de tipo ***ModelNotFoundException*** .

```
public function update(Request $request)
{
    if (!$request->ajax()) return redirect('/');
    $categoria = Categoria::findOrFail($request->id);
    $categoria->nombre = $request->nombre;
    $categoria->descripcion= $request->descripcion;
    $categoria->condicion = '1';
    $categoria->save();
}
```

Si encontró el registro se guardarán los nuevos valores de sus parámetros con el método ***save*** de ***Eloquent-ORM.***

MÉTODO ACTIVAR Y DESACTIVAR

Estos métodos funcionan de manera similar, cambiando de estado el atributo condición de la Categoría.

```
public function activar(Request $request){
    if (!$request->ajax()) return redirect('/');
    $categoria = Categoria::findOrFail($request->id);
    $categoria->condicion = '1';
    $categoria->save();
}
public function desactivar(Request $request){
    if (!$request->ajax()) return redirect('/');
    $categoria = Categoria::findOrFail($request->id);
    $categoria->condicion = '0';
    $categoria->save();
}
```

El funcionamiento es primero buscando con el método '***findOrFail'*** y guardando con el método ***save*** de ***Eloquent-ORM.***

MÉTODO SELECCIONAR CATEGORÍA

Este método no hace parte del CRUD pero es necesario en el CRUD de ejemplares debido a que se necesita escoger la categoría a la cual pertenece un libro cuando se ingresa, esto mediante un campo tipo '***select'*** en la vista de ejemplares en donde se pueden escoger las categorías por nombre.

Primero se hace la consulta de las categorías que estén activas y se pide el 'id' y el 'nombre' de la categoría ordenados por el nombre de forma ascendente como se muestra en la siguiente imagen.

```
public function selectCategoria(Request $request){
    if (!$request->ajax()) return redirect('/');
    $categorias = Categoria::where('condicion','=','1')
    ->select('id','nombre')->orderBy('nombre', 'asc')->get();
    return ['categorias' => $categorias];
}
```

El método finalmente retorna el arreglo 'categorias' con todos los registros encontrados a la vista de ejemplares.

CONTROLADOR COLECCIÓN

Para este controlador fue necesaria la implementación de seis métodos para el óptimo funcionamiento del CRUD y parte visual de objetos tipo '***Colección'*** en otras vistas.

MÉTODO MOSTRAR

Este método se encarga de consultar todos los registros de la tabla ***'colecciones'*** en donde se le envía con Ajax como parámetro un campo a buscar y un criterio de búsqueda el cual hace referencia a un campo de la tabla.

```php
public function index(Request $request)
{
    if (!$request->ajax()) return redirect('/');

    $buscar = $request->buscar;
    $criterio = $request->criterio;

    if ($buscar==''){
        $colecciones = Coleccion::orderBy('id', 'desc')->paginate(10);
    }
    else{
        $colecciones = Coleccion::where($criterio, 'like', '%'. $buscar . '%')->orderBy('id', 'desc')->paginate(10);
    }
    return [
        'pagination' => [
            'total'        => $colecciones->total(),
            'current_page' => $colecciones->currentPage(),
            'per_page'     => $colecciones->perPage(),
            'last_page'    => $colecciones->lastPage(),
            'from'         => $colecciones->firstItem(),
            'to'           => $colecciones->lastItem(),
        ],
        'colecciones' => $colecciones
    ];
}
```

El método retorna un arreglo de tipo ***Coleccion*** con todos los registros en grupos de a diez ordenados por ***'id'*** de manera descendente utilizando el método ***pagínate*** para que se facilite mostrar, todo si en el parámetro buscar no se le envía nada, si el parámetro buscar contiene algo se hará una búsqueda con ***Eloquent-ORM*** con doble comodín ***('%'***) para buscar coincidencias en cualquier parte del registro.

En el retorno se envían dos parámetros necesarios a la vista, el primero es '***pagination***' el cual tiene varios elementos necesarios para el funcionamiento de la paginación y el segundo es el arreglo '***colecciones'***.

MÉTODO ALMACENAR

Este método recibe todos los campos requeridos para crear un nuevo registro, se crea un objeto de tipo 'Colección' y se le asigna valor a cada uno de sus atributos, la condición por defecto es '1' debido a que lo más lógico es que cuando se ingrese un registro es porque está disponible.

```php
public function store(Request $request)
{
    if (!$request->ajax()) return redirect('/');
    $coleccion = new Coleccion();
    $coleccion->nombre = $request->nombre;
    $coleccion->descripcion= $request->descripcion;
    $coleccion->condicion = '1';
    $coleccion->save();
}
```

Luego mediante ***Eloquent-ORM*** se guarda en la base de datos con el método ***save.***

MÉTODO ACTUALIZAR

Para la actualización de un registro se utiliza el método '***findOrFail'*** el cual busca el registro por el 'id', si lo encuentra creara un objeto con sus datos y se le asignarán los datos enviados desde Ajax, si no generará una excepción de tipo ***ModelNotFoundException*** .

```php
public function update(Request $request)
{
    if (!$request->ajax()) return redirect('/');
    $coleccion = Coleccion::findOrFail($request->id);
    $coleccion->nombre = $request->nombre;
    $coleccion->descripcion= $request->descripcion;
    $coleccion->condicion = '1';
    $coleccion->save();
}
```

Si encontró el registro se guardarán los nuevos valores de sus parámetros con el método ***save*** de ***Eloquent-ORM.***

MÉTODO ACTIVAR Y DESACTIVAR

Estos métodos funcionan de manera similar, cambiando de estado el atributo condición de la Categoría

```php
public function activar(Request $request){
    if (!$request->ajax()) return redirect('/');
    $coleccion = Coleccion::findOrFail($request->id);
    $coleccion->condicion = '1';
    $coleccion->save();
}
public function desactivar(Request $request){
    if (!$request->ajax()) return redirect('/');
    $coleccion = Coleccion::findOrFail($request->id);
    $coleccion->condicion = '0';
    $coleccion->save();
}
```

El funcionamiento es primero buscando con el método '***findOrFail'*** y guardando con el método ***save*** de ***Eloquent-ORM.***

MÉTODO SELECCIONAR COLECCIÓN

Este método no hace parte del CRUD pero es necesario en el CRUD de ejemplares debido a que se necesita escoger la colección a la cual pertenece un ejemplar cuando se ingresa, esto mediante un campo tipo '***select'*** en la vista de ejemplares en donde se pueden escoger la colección por nombre.

Primero se hace la consulta de las colecciones que estén activas, se pide el 'id' y el 'nombre' de la colección ordenados por el nombre de forma ascendente como se muestra en la siguiente imagen:

```
public function selectColeccion(Request $request){
    if (!$request->ajax()) return redirect('/');
    $colecciones = Coleccion::where('condicion','=','1')
    ->select('id','nombre')->orderBy('nombre', 'asc')->get();
    return ['colecciones' => $colecciones];
}
```

El método finalmente retorna el arreglo 'colecciones' con todos los registros encontrados a la vista de ejemplares.

CONTROLADOR ROL

Para este controlador fue necesaria la implementación de solo dos métodos debido a que es una tabla estática la cual contiene solo dos registros (dos roles), ambos métodos son para la parte visual, uno para mostrar los dos registros en una table y el otro para seleccionar el rol al registrar el usuario.

MÉTODO MOSTRAR

Este método se encarga de consultar todos los registros de la tabla ***'roles'*** en donde se le envía con Ajax como parámetro un campo a buscar y un criterio de búsqueda el cual hace referencia a un campo de la tabla.

```php
class RolController extends Controller
{
    public function index(Request $request)
    {
        if (!$request->ajax()) return redirect('/');

        $buscar = $request->buscar;
        $criterio = $request->criterio;

        if ($buscar==''){
            $roles = Rol::orderBy('id', 'desc')->paginate(3);
        }
        else{
            $roles = Rol::where($criterio, 'like', '%'. $buscar . '%')->orderBy('id', 'desc')->paginate(3);
        }
        return [
            'pagination' => [
                'total'        => $roles->total(),
                'current_page' => $roles->currentPage(),
                'per_page'     => $roles->perPage(),
                'last_page'    => $roles->lastPage(),
                'from'         => $roles->firstItem(),
                'to'           => $roles->lastItem(),
            ],
            'roles' => $roles
        ];
    }
```

El método retorna un arreglo de tipo ***Rol*** con todos los registros en grupos de a 3(por si luego se necesita un rol adicional) ordenados por ***'id'*** de manera descendente utilizando el método ***pagínate*** para que se facilite mostrar, todo si en el parámetro buscar no se le envía nada, si el parámetro buscar contiene algo se hará una búsqueda con ***Eloquent-ORM*** con doble comodín ***('%'***) para buscar coincidencias en cualquier parte del registro.

En el retorno se envían dos parámetros necesarios a la vista, el primero es '***pagination***' el cual tiene varios elementos necesarios para el funcionamiento de la paginación y el segundo es el arreglo '***roles'***.

MÉTODO SELECCIONAR ROL

Este método no hace parte del CRUD pero es necesario en el CRUD de **usuarios** debido a que se necesita escoger el rol que tiene un usuario al ser creado, esto mediante un campo tipo '***select'*** en la vista de **usuarios** en donde se puede escoger el rol por su nombre.

Primero se hace la consulta de las colecciones que estén activas, se pide el 'id' y el 'nombre' de la colección ordenados por el nombre de forma ascendente como se muestra en la siguiente imagen:

```php
public function selectRol(Request $request){
    $roles = Rol::where('condicion','=','1')
    ->select('id','nombre')
    ->orderBy('nombre','asc')->get();

    return [ 'roles' =>$roles];
}
```

El método finalmente retorna el arreglo '***roles'*** con todos los registros encontrados a la vista de ***usuarios***.

CONTROLADOR USUARIO

MÉTODO MOSTRAR

```
public function index(Request $request)
{
    if (!$request->ajax()) return redirect('/');

    $buscar = $request->buscar;
    $criterio = $request->criterio;

    if ($buscar==''){
        $personas = User::join('personas','users.id','=','personas.id')
        ->join('roles','users.idrol','=','roles.id')
        ->select('personas.id','personas.nombres','personas.celular',
        'personas.apellidos','personas.email','users.usuario','users.password',
        'users.condicion','users.idrol','roles.nombre as rol')
        ->orderBy('personas.id', 'desc')->paginate(3);
    }
    else{
        $personas = User::join('personas','users.id','=','personas.id')
        ->join('roles','users.idrol','=','roles.id')
        ->select('personas.id','personas.nombres','personas.celular',
        'personas.apellidos','personas.email','users.usuario','users.password',
        'users.condicion','users.idrol','roles.nombre as rol')
        ->where('personas.'.$criterio, 'like', '%'. $buscar . '%')
        ->orderBy('personas.id', 'desc')->paginate(3);
    }
    return [
        'pagination' => [
            'total'        => $personas->total(),
            'current_page' => $personas->currentPage(),
            'per_page'     => $personas->perPage(),
            'last_page'    => $personas->lastPage(),
            'from'         => $personas->firstItem(),
            'to'           => $personas->lastItem(),
        ],
        'personas' => $personas
    ];
}
```