CODEV

Prácticas de Código	
Lenguaje	PHP
Framework	Laravel
Aprobado por	Fabián Portillo

- Todo se maneja por GitHub
- Cada task o feature que se haga sobre el backend, debe tener su propia rama en GitHub
- Cuando se termine cada feature se debe hacer un pr y debe ser aprobado por todos los que estén trabajando en los proyectos
- Se tendrán dos ramas principales, la de develop y la de producción
- Pull request a develop y una vez aprobado se pasa a producción
- (diagrama de flujo de aprobación con el respectivo encargado)
- Subir a git diario o cada vez que se termine funcionalidad (feature)
- Usa la última versión de laravel la lts - 1 - toca revisar las dependencias
- Utilizar transformadores para respuestas http
- Utilizar form request scope
- La base de datos siempre normalizada en 3 forma normal
- Todas las variables en inglés (transformadores, modelos y demás)
- Documentación del código swager (puede ser en español)
- Por cada modelo de datos una carpeta de controladores y tienen un archivo (index, post, show y update), cada cosa debe tener su funcionalidad, por cada módulo si es necesario
- Si se afecta más de una tabla, tiene que ir con una transacción que reporte el error si lo hay sino hace el commit
- Sistema de autenticación con JWT
- Cada controlador protegido por los middleware

- Para los form request, una carpeta por controlador o por modelo
- Inyección implícita de dependencias
- Construcción de objetos se hace desde el constructor, inyección implícita
- Trade que funciona con los transformadores
- Datos paginados desde backend
- Control de debug telescop, si ven necesarios políticas o validaciones dentro del controlador, arrojando excepciones
- Todo dentro de bloques de tra catch
- nosotros pasamos el archivos de trate y el de excepciones
- Tabla polimórfica para los resources (archivos meda) y la url en la base de datos e guarda
- Antes de iniciar, hacer la base de datos y una vez aprobada pueden iniciar desarrollo

Estructura del proyecto:

A continuación, se explica el scaffold que debe seguir un proyecto Backend realizado en Laravel, siguiendo todas las recomendaciones mencionadas anteriormente:

1. **Scaffold:** es importante saber que Laravel genera una estructura de directorios predeterminados una vez que se crea un proyecto, los siguientes directorios son aquellos donde más se realizarán cambios en el proyecto:

```
app
    Http
      Controllers
      Middleware
      Requests
    Jobs
    Mail
    Models
config
database
    factories
    migrations
    seeds
resources
    views
routes
    api.php
    web.php
.env
```

2. **Estructura de una API:** El siguiente ejemplo muestra los pasos que se deben de seguir para crear un proyecto que expone recursos por medio De API Rest y con autenticación con JWT tenga en cuenta de que si se encuentra en un proyecto existente no es necesario que realice los siguientes pasos, pero sin embargo esto le ayudará a entender cómo funciona la estructura API del proyecto.

   Nota: Para el siguiente ejemplo se usará la versión 8 de Laravel, pero estos mismos pasos se pueden realizar con cualquier versión con la que se esté trabajando.

- Un proyecto de Laravel trabaja con guardias de autenticación, de los cuales podemos usar para WEB o API en este caso se usará el guardia api para generar la autenticación de los usuarios en el sistema.

  En App/Providers/RouteServiceProvider.php en la versión 8 de Laravel encontrará un método boot con el siguiente contenido:

```php
public function boot()
{
    $this->configureRateLimiting();

    $this->routes(function () {
        Route::prefix('api')
            ->middleware('api')
            ->namespace($this->namespace)
            ->group(base_path('routes/api.php'));

        Route::middleware('web')
            ->namespace($this->namespace)
            ->group(base_path('routes/web.php'));
    });
}
```

  En este caso el Route::profix(‘api’) indica el prefijo por donde se expondrán las rutas tipo API de la aplicación.

  Y el método ->group(base_path(‘routes/api.php’)) le indica al middleware de que archivo se leerán las rutas de la aplicación.

En versiones anteriores de Laravel puede que en App/Providers/RouteServiceProvider.php encuentre un método como este en lugar del anterior:

```
protected function mapApiRoutes()
{
    Route::prefix('api')
         ->middleware('api')
         ->namespace($this->namespace)
         ->group(base_path('routes/api.php'))
}
```

En este caso aplicarían los mismos métodos y el mismo prefijo, con esto Laravel nos indica que el prefijo para acceder a las rutas API de la aplicación es api http://localhost:8000/api y que las rutas declaradas para el uso de la API se encontrarán en routes/api.php. pero sabiendo esto se puede cambiar la configuración si así se desea, pero lo más recomendable es dejarlo como esta.

Ahora que ya se sabe dónde se declaran las rutas para el api en routes/api.php ingresaremos el siguiente código solo para el ejemplo:

```
Route::get('user', function(){
    return ['user' => 'user'];
});
```

Luego ejecutar el siguiente comando desde la terminal: php artisan serve

Y desde el navegador ingresar a http://localhost:8000/api/user y la respuesta del navegador debería verse así:

```
{"user":  "user"}
```

3. **Instalación autenticación JWT**: Es necesario instalar el paquete tymon/jwt-auth para poder generar y administrar el contenido de los JWT para cada usuario autenticado en el sistema.

- ejecute el siguiente comando para instalar el paquete: composer require tymon/jwt-auth
- Agregue el proveedor de servicios a la providers matriz en el config/app.php

```
'providers' => [
    ...
    Tymon\JWTAuth\Providers\LaravelServiceProvider::class,
]
```

- Ejecute el siguiente comando para publicar el archivo de configuración del paquete:

  php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"

- Ahora deberá tener un config/jwt.php archivo donde podrá realizar las configuraciones de este paquete, cambie la propiedad ttl para prolongar el tiempo de vida del token de forma que quede de esta manera:

```
'ttl' => env('JWT_TTL', 999999),
```

- Ejecute el siguiente comando desde su terminal para generar la JWT_SECRET propiedad en su archivo .env

  php artisan jwt:secret

- Esto actualizará su .env archivo con algo como JWT_SECRET=foobar es la clave que se utilizará para firmar sus tokens.

- En primer lugar, debe implementar el Tymon\JWTAuth\Contracts\JWTSubject contrato en su modelo de Usuario, lo que requiere que implemente los 2 métodos getJWTIdentifier() y getJWTCustomClaims().

  Ejemplo en Laravel 8:

```php
<?php

namespace App\Models;

use Laravel\Sanctum\HasApiTokens;
use Tymon\JWTAuth\Contracts\JWTSubject;
use Illuminate\Notifications\Notifiable;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements JWTSubject
{
    use HasApiTokens, HasFactory, Notifiable;

    public $table = 'users';

    public function getJWTIdentifier()
    {
        return $this->getKey();
    }

    public function getJWTCustomClaims()
    {
        return [];
    }

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

Ejemplo en Laravel 7:

```php
<?php

namespace App;

use Tymon\JWTAuth\Contracts\JWTSubject;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements JWTSubject
{
    use Notifiable;

    public function getJWTIdentifier()
    {
        return $this->getKey();
    }

    public function getJWTCustomClaims()
    {
        return [];
    }
}
```

- Dentro del config/auth.php archivo, deberá realizar algunos cambios para configurar Laravel para que use la jwt protección para activar la autenticación de su aplicación.

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
        'hash' => false,
    ],
],
```

4. **Configuración del CORS:** por defecto nuestra API tendrá deshabilitado el intercambio de recursos de origen cruzado, para habilitar orígenes desde cliente específicos es necesario instalar el paquete fruitcake/laravel-cors ejecuta el siguiente comando desde la terminal:

   composer require fruitcake/laravel-cors

   Luego agrega el siguiente middleware en el archivo App\Http\Kernel.php en la propiedad $middleware.

```
protected $middleware = [
    \Fruitcake\Cors\HandleCors::class,
    ...
];
```

   en la versión 8 de Laravel normalmente este paquete ya está instalado e incluido en los middlewares de la aplicación pero es importante verificar igualmente el archivo App\Http\Kernel.php.

- En el archivo config/cors.php puedes modificar la configuración para permitir acceso únicamente a los orígenes deseados o dejar la configuración por defecto para permitir acceso a cualquier cliente:

```
'paths' => ['api/*', 'sanctum/csrf-cookie'],

'allowed_methods' => ['*'],

'allowed_origins' => ['*'],

'allowed_origins_patterns' => [],

'allowed_headers' => ['*'],

'exposed_headers' => [],

'max_age' => 0,

'supports_credentials' => false,
```

5. **Traits API Response**: Una buena practica en el desarrollo es asignar la responsabilidad única, en este caso usaremos un trait para administrar el formato y los métodos de respuesta de la API y otro encargado de administrar el proceso de autenticación de usuarios.

- Cree un archivo app/Traits/ApiResponse.php con el siguiente contenido: https://github.com/Gabriel1777/practices/blob/master/backend/ApiResponse.php

- Cree un archivo app/Traits/AuthManager.php con el siguiente contenido: https://github.com/Gabriel1777/practices/blob/master/backend/AuthManager.php

- Cree un archivo app/Serializers/NoDataSerializer.php con el siguiente contenido: https://github.com/Gabriel1777/practices/blob/master/backend/NoDataSerializer.php

- Cree un archivo app/Http/Controllers/ApiController.php con el siguiente contenido:

```php
<?php

namespace App\Http\Controllers;

use App\Traits\ApiResponse;
use App\Traits\AuthManager;

class ApiController extends Controller
{
    use AuthManager;
    use ApiResponse;
}
```

6. **Manejo de excepciones:** en una API es muy importante el manejo global de los errores, de esto depende el formato de repuesta una vez que el Backend arroja un error 500, 400, 404, 401, etc …

- para esto modifique el app/Exceptions/Handler.php con el siguiente contenido: https://github.com/Gabriel1777/practices/blob/master/backend/Handler.php

- Con esto se manejará los mensajes que devolverá la API una vez ocurra cada tipo de excepción especificado, utilizando el app/Traits/ApiResponse.php creado anteriormente se maneja el formato de respuesta.

7. **Transformadores:** Un transformador le ayudara a transformar un modelo de Laravel en un formato JSON que luego se retornara como respuesta de la API.

- Para esto es necesario instalar el paquete de laravel-fractal, ejecute el siguiente comando desde la terminal ubicado en la raíz del proyecto: composer require spatie/laravel-fractal

- Solo en la versión 5.8 o menor de laravel es necesario instalar el paquete de metricloop para hacer funcionar los transformadores, si se encuentra en una versión superior simplemente ignore este paso.

  Para instalarlo ejecute desde la terminal: composer require metricloop/laravel-transformer-maker

- Ejecute desde la terminal el siguiente comando para crear un transformador de ejemplo: php artisan make:transformer UserTransformer

- Esto le debió crear un archivo app/Transformers/UserTransformer.php archivo

- Dentro del archivo que acaba de crear copie y pegue el siguiente contenido:

  https://github.com/Gabriel1777/practices/blob/master/backend/UserTransformer.php

- En este caso el arreglo que retorna el método transform será el json que se devolverá para cada objeto en la API ya sea un solo modelo de usuario o una colección, no olvide que puede modificar este arreglo según sus necesidades.

- El método includeRole permitirá llamar el rol del usuario por URL con un include=role este usa el RoleTransformer para anidarlo en la respuesta.

- Agrega la propiedad transformer en su modelo en este caso el de Usuario para indicarle a la ApiResponse clase que transformador debe usar para crear la colección de datos del modelo y darle formato en la devolución de la API.

  Importa la clase del transformador en el modelo:

```
use App\Transformers\UserTransformer;
```

  Agregue la propiedad transformer:

```
public $transformer = UserTransformer::class;
```

8. **Controladores:** Los controladores de la aplicación siguen la siguiente estructura, para este ejemplo primero ejecute el siguiente comando desde la terminal:

   php artisan make:controller User/UserIndexController

- Esto debió crear un app/Http/Controllers/User/UserIndexController.php archivo.

- el scaffold en los controladores utiliza un archivo por cada método y todos deben estar dentro de una carpeta padre ejemplo:

```
app
  Http
    Controllers
      User
        UserIndexController.php
        UserCreateController.php
        UserUpdateController.php
        UserDeleteController.php
```

- Dentro de cada archivo existe la lógica que le corresponde en el caso de UserCreateController la lógica de creación de usuarios.

- UserUpdateController la lógica de actualización de usuarios
- UserDeleteController la lógica de eliminación de usuarios
- UserIndexController la lógica para devolver colecciones o modelos singulares de usuario.

- También puede optar por utilizar un controlador resource por ejemplo UserController.php que administre la lógica de todos los métodos pero si lo hace asegúrese de que cada método tenga líneas cortas y simples de código para mejor legibilidad del código.

- El siguiente ejemplo muestra la estructura de un controlador que devuelve colecciones de datos como por ejemplo UserIndexController.php:

  https://github.com/Gabriel1777/practices/blob/master/backend/UserIndexController.php

- En este ejemplo el controlador hereda de ApiController que es la clase que hereda todos los métodos de ApiResponse por lo cual podemos llamar el método showAll para devolver una colección de usuarios.

- También se llama al middleware auth:api desde el constructor de la clase de esta manera y en todos los controladores se deben llamar el middleware para las rutas privadas de la aplicación.

- Las clases, modelos, servicios etc. que se utilicen desde el controlador deben ser instanciadas desde el constructor como en este caso lo hace la clase del modelo User.

- Todo método de un controlador debe estar encerrado por un bloque try cath y utilizar el método errorResponse heredado de la ApiResponse clase para el manejo de excepciones.

- El siguiente es un ejemplo con UserCreateController.php:

  https://github.com/Gabriel1777/practices/blob/master/backend/UserCreateController.php

- Como se puede dar cuenta este controlador es prácticamente igual al anterior con la diferencia de que usa un showOne para devolver el modelo creado en lugar de una colección de datos, usa el DB::commit() para autorizar el guardado del registro si todo ha marchado bien y el DB::rollback() para cancelar el guardado del registro en la base de datos en caso de que ocurra una excepción.

- El siguiente es un ejemplo con UserUpdateController.php:

  https://github.com/Gabriel1777/practices/blob/master/backend/UserUpdateController.php

- Este controlador es prácticamente igual al de UserCreateController con la diferencia de que este recibe un usuario por parámetro que es el que se va a actualizar.

- Cabe resaltar que es importante el uso del implicit binding para obtener los modelos a actualizar desde el controlador

```
    public function update(UserUpdateRequest $request, User $user)
    {
        DB::beginTransaction();
        try{
            $this->user = $user->setData($request->all());
            $this->user->save();
            DB::commit();
            return $this->showOne($this->user->refresh(), 200);
        } catch (\Exception $exception){
            DB::rollback();
            return $this->errorResponse($exception->getMessage(), 400);
        }
    }
}
```

- Por último es importante resaltar que los controladores no pueden tener lógica de negocio, esto le corresponde a los modelos como en este ejemplo se llama el método setData del modelo User para actualizar los datos en lugar de tener esa lógica en el controlador.

9. **Modelos**: Los modelos administran la lógica de negocio de la aplicación, básicamente cada modelo representa una entidad de la base de datos y de la misma forma la lógica de negocio que se ejecuta sobre esta.

- En cada modelo debe existir un método especializado para cada operación que se realice sobre la entidad de la base de datos con la que está asociado.

- Los controladores solo deben llamar métodos con lógica preparada de los modelos. Nunca un controlador ejecuta directamente lógica sobre la base de datos.

- Los modelos declaran los valores constantes de su entidad en la base de datos. Por ejemplo el modelo State que administra la tabla states puede contener una constante con la lista de estados que se almacenaran de fabrica sobre la base de datos.

```
const STATES = [
    [
        "id" => "enabled",
        "name" => "Enabled",
        "type" => "general"
    ],
    [
        "id" => "disabled",
        "name" => "Disabled",
        "type" => "general"
    ]
];
```

- Luego puede crear un seeder que itere este arreglo y guarde la información sobre la base de datos:

```php
<?php

namespace Database\Seeders;

use App\Models\State;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class StateSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        foreach (State::STATES as $state) {
            DB::table('states')->insert([
                'id' => $state['id'],
                'name' => $state['name'],
                'type' => $state['type']
            ]);
        }
    }
}
```

- Para las relaciones de base de datos se recomienda usar el belongsTo, hasOne, hasMany, belongsToMany relaciones que provee eloquent

10. **Servicios:** Comúnmente las aplicaciones API deben comunicarse con servicios que proveen terceros.

- un ejemplo de esto pueden ser integración con pasarelas de pago, servicios de Google , Firebase, Servicios de Amazon, SMS o algún software como servicio como lo podría ser una plataforma de firmas digitales, de facturación entre otras.

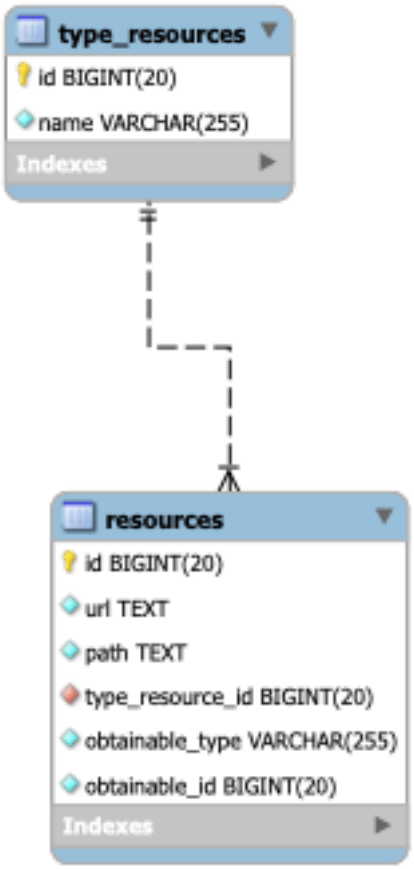
- En estos casos es importante organizar esta lógica de integración en un archivo dentro de la carpeta app/Services un ejemplo podría ser una integración del servicio de Dynamic links de Firebase en app/Services/DynamicLinkService.php el cual administra la lógica de integración con dicho servicio:

  https://github.com/Gabriel1777/practices/blob/master/backend/DynamicLinkService.php

11. **Base de datos:** A la hora de estructurar una base de datos pueden salir muchas soluciones para un solo problema y todas estas ser validas, sin embargo hay ciertos parámetros a tener en cuenta a la hora de crear una base de datos relacional:

- Todas las tablas de la base de datos deben tener una clave única ID.
- Las relaciones deben ser coherentes según los requerimientos de cada proyecto.
- Para las tablas constantes que almacenan datos y estos no son administrables desde código o desde una aplicación preferiblemente su ID debe ser de tipo string para que este siempre sea constante y no cambie.

- Para las imágenes se debe utilizar una tabla polimórfica resources y una type_resources con el fin de almacenar las imágenes de cualquier otro tipo de entidad en la base de datos.



- Ya se ha programado una lógica para el almacenamiento de archivos que preferiblemente se debería de reutilizar, modelo Resource:

- Este modelo usa la librería intervention/image para administrar el tamaño de las imágenes ejecute el siguiente comando desde la terminal:

  composer require intervention/image

- en app/Resource.php agregue el siguiente código:

  https://github.com/Gabriel1777/practices/blob/master/backend/Resource.php

en app/TypeResorce.php agregue el siguiente código y ajústelo según sus necesidades:

https://github.com/Gabriel1777/practices/blob/master/backend/TypeResource.php

- en database/sedes/TypeResourceSeeder.php agregue el siguiente código:

  https://github.com/Gabriel1777/practices/blob/master/backend/TypeResourceSeeder.php

- ejecute los siguientes comandos para crear las migraciones

  php artisan make:migration create_type_resources_table

  php artisan make:migration create_resources_table

- en database/migrations/…..create_type_resources_table.php agregue el siguiente código:

  https://github.com/Gabriel1777/practices/blob/master/backend/2021_10_14_221348_create_type_resources_table.php

- en database/migrations/…..create_resources_table.php agregue el siguiente código:

  https://github.com/Gabriel1777/practices/blob/master/backend/2021_10_14_221409_create_resources_table.php

- Y con esto ya tendría toda la lógica de archivos reutilizada , ahora solo tiene que ejecutar las migraciones y seeders desde la terminal para hacer efectivos los cambios.

  php artisan migrate --seed

12. **Documentación:** Toda API debe tener una documentación para cada endpoint y recurso que esta expone para esto se recomienda usar el paquete l5-swagger.

- Instalación Laravel 7.x y 8.x

  composer require "darkaonline/l5-swagger"

- Laravel 7.x

  composer require "darkaonline/l5-swagger:7.*"

- esta librería proporciona documentación para los endpoints de su api, diseño, requests, parameters, response etc... con solo agregar unos comentarios en cualquier parte de su proyecto. sin embargo la forma más recomendada es en cada controlador donde se maneja el endpoint este su propia documentación.

- para empezar agregue este comentario en su app\Http\Controllers\ApiController.php o app\Http\Controllers\Controller.php dependiendo si estas usando una api o una web.

- Ejemplo:

  https://github.com/Gabriel1777/practices/blob/master/backend/ApiController.php

- en la propiedad @OA\Contact email puede poner su correo como desarrollador y en @OA\Server url es importante que ponga la url completa del dominio de la api correctamente ya que l5-swagger la usara para que los usuarios que visiten la documentación puedan hacer pruebas.

- El siguiente es un ejemplo de un controlador que usa comentarios de l5-swagger para generar la documentación de la API:

  https://github.com/Gabriel1777/practices/blob/master/backend/SwaggerController.php

  para hacer efectiva la documentación ejecute el siguiente comando desde la terminal:

  php artisan l5-swagger:generate

php artisan serve

y luego ingrese a http://localhost:8000/api/documentation y debería ver la documentación de su API.

13. **Inicio de sesión:** Anteriormente se creó un app/Traits/AuthManger.php archivo con la lógica de credenciales y de validación del usuario autenticado en el sistema.

- El siguiente es un ejemplo de un controlador que hereda del ApiController clase y esta hereda del AuthManager clase, de esta forma se pueden llamar los métodos de la AuthManager trait para realizar operaciones de autenticación desde el controlador.

  https://github.com/Gabriel1777/practices/blob/master/backend/AuthLoginController.php

14. **Recuperación de contraseña**: La mayoría de sistemas usan la recuperación de contraseña, si bien esta funcionalidad puede realizarse de muchas maneras hay ciertas recomendaciones que se deben tener en cuenta.

- En la web normalmente se usa una ruta de recuperación de contraseña y un token que se envía a través de la url para validar el acceso, esta url se envía a través de un correo electrónico al usuario.

- En la api se envía un código de autorización al correo electrónico del usuario que posteriormente el usuario ingresara desde una aplicación externa para validar el cambio de contraseña.

- Si bien siguiendo esta funcionalidad pueden existir diferentes formas de llegar al mismo objetivo cabe mencionar que la lógica de recuperación de contraseña ya está programada y que puede reutilizar este código en su proyecto o guiarse de las practicas que se utilizan aquí para lograr la funcionalidad.

- Cree un archivo mail ejecutando php artisan make:mail ResetPasswordUser de manera que obtenga un archivo en app/Mail/ResetPasswordUser.php con el siguiente contenido:

  https://github.com/Gabriel1777/practices/blob/master/backend/ResetPasswordUser.php

- Cree un archivo en resources/views/emails/resetPassword.blade.php con el siguiente contenido:

  https://github.com/Gabriel1777/practices/blob/master/backend/resetPassword.blade.php

- Asegúrese de tener la migración de password_resets_table o si no créela ejecutando desde la terminal php artisan make:migration create_password_resets_table y agregándole el siguiente contenido:

  https://github.com/Gabriel1777/practices/blob/master/backend/2014_10_12_100000_create_password_resets_table.php

- Ahora cree el modelo de password resets ejecutando php artisan make:model PasswordReset esto debió crear un app/Models/PasswordReset.php archivo, ahora agregue el siguiente contenido sobre el:

  https://github.com/Gabriel1777/practices/blob/master/backend/PasswordReset.php

- En app/Http/Requests/RecoverPassword/RecoverPasswordEmail.php agregue el siguiente contenido:

  https://github.com/Gabriel1777/practices/blob/master/backend/RecoverPassword/RecoverPasswordEmail.php

- En app/Http/Requests/RecoverPassword/RecoverPasswordReset.php agregue el siguiente contenido:

  https://github.com/Gabriel1777/practices/blob/master/backend/RecoverPassword/RecoverPasswordReset.php

- En app/Http/Requests/RecoverPassword/RecoverPasswordToken.php agregue el siguiente contenido:

  https://github.com/Gabriel1777/practices/blob/master/backend/RecoverPassword/RecoverPasswordToken.php

- Cree el siguiente controlador ejecutando php artisan make:controller RecoverPassword/RecoverPasswordController esto le genera un archivo en app/Http/Controllers/RecoverPassword/RecoverPasswordController.php modifique el archivo de tal forma que quede de la siguiente manera:

  https://github.com/Gabriel1777/practices/blob/master/backend/RecoverPasswordController.php

- Por ultimo puede crear las rutas para la recuperación de contraseña en el archivo routes/api.php y agregando las siguientes líneas de código:

```
use App\Http\Controllers\RecoverPassword\RecoverPasswordController;
```

```
//Routes for register
Route::post('register', [RegisterController::class, 'register'])
    ->name('register');

//Routes for recover password
Route::post('forgotPassword', [RecoverPasswordController::class,
    'forgotPassword'])->name('api.forgot.password');
Route::post('password/reset/verifyToken', [
    RecoverPasswordController::class, 'verifyTokenResetPassword']
    )->name('api.passwordReset.verifyToken');
Route::post('password/reset', [RecoverPasswordController::class,
    'resetPassword'])->name('api.password.reset');
```

- No olvide ejecutar migraciones y seeders para hacer efectivas las migraciones de recuperación de contraseña.

- Tampoco olvide que debe realizar la configuración de correo electrónico desde el archivo .env para poder enviar correos electrónicos de recuperación de contraseña a los usuarios.

RESPONSABLE DE APROBACIONES.

Por CODEV	**Ricardo Ramos Cuervo** **312 449 3543** **comercial@codev.com.co**