**Máster web full Stack | Api Rest Full | Laravel – Angular**

**Instalación del entorno:**

Se debe preparar el entorno de desarrollo local de la siguiente manera:

* Servidor local de aplicaciones (XAMPP).
* Instalación Laravel:

<https://laravel.com/docs/8.x/installation>

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

* Crear un host virtual XAMPP (Recomendado).

**Diseño de la base de datos:**

De acuerdo con la necesidad del cliente se debe realizar el diseño de la base de datos, se recomienda el programa DIA o cualquiera que permita generar una imagen del modelo relacional de la base de datos.

**Creación de la base de datos:**

Se toma el diseño anterior y se programa por medio de lenguaje SQL en la base de datos cada una de las tablas que contendrá el software. Es importante que en cada tabla crees los siguientes campos:

create\_at: datetime

update\_at: datetime

remember\_token: varchar

**Conectar la base de datos con LARAVEL:**

Luego de crear el proyecto de Laravel y tener todo listo, se debe realizar la creación de la base de datos y configurar en el archivo. ENV toda la información para realizar la conexión, ya sea a MariaBD o Postgres.

El archivo debe ir configurado de la siguiente manera, dependiendo cual base de datos se vaya a utilizar:

|  |  |
| --- | --- |
| **María BD** | **Postgresql** |
| DB\_CONNECTION=mysql  DB\_HOST=127.0.0.1  DB\_PORT=3306  DB\_DATABASE=[nombreBD]  DB\_USERNAME=[usuario]  DB\_PASSWORD=[clave] | DB\_CONNECTION=pgsql  DB\_HOST=127.0.0.1  DB\_PORT=5432  DB\_DATABASE=[NombreDB]  DB\_USERNAME=[username]  DB\_PASSWORD=[Clave] |

**Nota:** en caso de elegir trabajar con PostgreSQL es importante activar algunas opciones del archivo php.ini del XAMPP

extension=pdo\_pgsql  
extension=pgsql

Se deben activar retirando el punto y coma que hay delante de ellos.

**Creación de los modelos:**

Los modelos son aquellos que nos permiten gestionar las operaciones con la base de datos. Para la creación de los modelos de la aplicación es necesario seguir los siguientes pasos, se debe crear un modelo por cada tabla que tengamos en la base de datos. Los modelos automáticamente incluyen la clase Eloquent que contiene nuestro ORM, este último nos permite hacer consultas e instrucciones a la base de datos de una manera sencilla sin necesidad de escribir código SQL. Tratamos la base de datos como si fuera un objeto.

1. Abrir la terminal en la carpeta donde esta nuestro proyecto y escribir la siguiente instrucción:

*php artisan make:model [NombreModelo]*

1. Se debe indicar en la clase de cada modelo la tabla de la base de datos que va a usar de la siguiente manera:

*Protected $table = ‘NombreTablaBD’;*

1. En caso de ser necesario, se deben indicar las relaciones que existen entre las tablas de la base de datos (uno a uno, uno a muchos, muchos a uno). Para hacerlo se procede de la siguiente manera:

* Se crea un método público con el nombre de la tabla con la cual se relaciona la tabla de la clase y luego se usa los métodos de relaciones entre tablas de Laravel: <https://laravel.com/docs/8.x/eloquent-relationships#one-to-one>

*Public function [NombreTablaConlaqueseRelaciona]*

*{*

*return $this->hasOne(Clase);*

*}*

1. Es interesante luego de que se creen todas las relaciones de la base de datos, generar algunas pruebas para confirmar que todo funciona correctamente.

* Para probar los modelos necesitas usar un controlador, este controlador debe importar los modelos que va a usar de la siguiente manera: el controlador lo puedes llamar *PruebasController.*

*Use App\Models\[nombreModelo];*

* Luego puedes crear un método para testear el ORM en el cuál hagas una consulta a una tabla de la base de datos de la siguiente manera:

*Public function testOrm()*

*{*

*$[tabla] = Tabla::all();*

*Foreach($tablas as tabla)*

*{*

*//muestra los campos de una tabla.  
echo $tabla->campo;*

*//muestra los campos de una tabla relacionada.*

*echo $tabla->tablaRelacionada->campoRelacionado.*

*}*

*}*

* Debes crear una ruta de prueba para que te funcione la consulta y prueba que estas realizando:

*Route::get('/test-orm', [PruebasController::class, 'testOrm']);* las rutas se van a estudiar mas adelante.

**Creación de los controladores:**

Es importante recordar el papel de los controladores: reciben datos de las vistas y realizan algunas operaciones y conexión con la base de datos, posteriormente reciben información de la base de datos y luego la llevan a la vista nuevamente. Para crear los controladores se deben seguir los siguientes pasos:

1. Abrir la terminal y luego ir a la carpeta del proyecto, posteriormente ejecutar la siguiente instrucción:

*php artisan make:controller [TablaController];*

1. Es importante probar nuestros controladores, por eso creamos una función en cada controlador que reciba como parámetro el objeto Request (este objeto nos servirá para recibir información y operarla)

*public function pruebas(Request $request)*

*{*

*return “Probando el controlador de pruebas”;*

*}*

1. Luego para que nuestro controlador funcione, es importante crearle una ruta de la siguiente manera:

*Route::get('/ruta', [NombreController::class, 'metodo']);*

**Métodos para pasar la información:**

El API es RestFull cuando utiliza los siguientes métodos:

GET: Conseguir datos o recursos, sirve para recibir información.

POST: Guardar datos o recursos o hacer lógica desde un formulario, sirve para enviar información.

PUT: Actualizar recursos o datos.

DELETE: Eliminar datos o recursos.

Para usar los métodos anteriormente mencionados es importante utilizar un cliente REST, se recomienda *POSTMAN* este nos permite servir como un Frontend de pruebas para poder testear la aplicación.

**Desactivar protección de LARAVEL para formularios:**

Es importante comentar la siguiente línea:

*\App\Http\Middleware\VerifyCsrfToken::class,*

El archivo se encuentra en la ruta \App\Http\Kernel.php. Es una protección de Laravel a la hora de trabajar con formularios.

**Recepción de datos externos de POSTMAN en la aplicación:**

Para recibir los datos de un Frontend en nuestra aplicación, primero debemos crear una variable para guardar los datos que trae nuestro formulario y luego con el método input() lo vamos a recibir. A continuación, se detalla cómo debe realizarse:

*$variable = $request->input(‘campo’);*

En la variable quedaría guardada la información que traes del Frontend. (realiza pruebas con postman).

**Devolver respuestas en JSON desde mi aplicación:**

Las respuestas que mi aplicación va a entregar siempre deben estar en JSON, para realizar una respuesta JSON se procede de la siguiente manera: se define un array en el cuál se relaciona la información que se quiere enviar:

*$datos = array*

*(*

*‘status’ => ‘Error’,*

*‘code’ => ‘código de error’ 200 correctas, 400 incorrectas.*

*‘message’ => ‘Aquí colocas un mensaje de error cualquiera’*

*);*

Luego de generar el Array entonces se debe retornar en formato JSON de la siguiente manera, el método json() me convierte un array en JSON.

*return response()->json($datos, [código]);*

**Método de registro de usuarios:**

Para generar un registro de usuarios, en nuestro controlador de usuarios vamos a generar un método de registro que debe recibir como parámetro el objeto Request y que tendrá las siguientes capas:

Desde postman debes hacer una petición al API y enviar un JSON:

{“variable1”:”valor1”,”variable2”:”valor2”}

***//Recoger el JSON con los datos de usuario por medio del método POST.***

*$json = $request->input(‘json’,null);*

***//Se debe decodificar el JSON como parámetro o array.***

*$params = json\_decode($json); //parámetros en objeto.  
 $params\_array = json\_decode($json); //parámetros en array.*

*//****Puedes validar si hay datos en los parámetros***

*If(!empty($params)&&!empty($params\_array))*

*{Encierras toda la validación aquí} else {Mensaje error}*

*//****Limpiar datos recibidos por JSON con el método trim.***

$params\_array = array\_map(‘trim’, $params\_array);

***//Validar que los datos sean digitados de manera correcta.***

Para realizar las validaciones en laravel, vamos a utilizar la librería *Validator* que nos proporciona laravel y le vamos a enviar un array con los datos que queremos validar. Para documentarse mejor con relación a los tipos de validaciones puedes revisar: <https://laravel.com/docs/8.x/validation#manually-creating-validators>

$validate = \Validator::make($params\_array, [

‘CampoaValidar’ => ‘VALIDACION’  
 ‘campoaValidar’ => ‘unique:users’ //***//Comprobar si el usuario existe en el sistema.***

]);

if($validate->fails())

{

$datos = array(

*‘status’ => ‘Error’,*

*‘code’ => ‘código de error’ ,*

*‘message’ => ‘mensaje de error’,*

*‘errors’ => $validate->errors()*

);

}else

{

$datos = array(

*‘status’ => ‘Sucess’,*

*‘code’ => ‘código éxito’ ,*

*‘message’ => ‘mensaje’*

);

}

*return response()->json($datos, [código]);*

***//Cifrar contraseñas***

*$pwd = hash(‘sha256’,$params->password)*

***//Creación del usuario.***

**\*\*** Recordar que es importante incluir el modelo para poder hacer acciones con la base de datos:

*use App\Models\[ModeloUsar];*

//instanciamos un objeto del modelo.

$user = new User();

$user->campo = $params\_array[‘datoGuardar’];

***//Guardar usuario.***

$user->save();

**instalación de la librería JWT:**

En primer lugar, debemos instalar la librería de JWT para hacer validaciones por medio de Token en la autenticación de usuarios. Para instalarla procedemos de la siguiente manera.

* Vamos al archivo composer.json y añadimos la siguiente dependencia en require MIT:

"firebase/php-jwt":"3.0.0"

(Se pueden probar otras versiones)

* Luego abrimos la consola de comandos en la carpeta del proyecto y ejecutamos el siguiente CMD:

*composer update*

**Creación de un HELPER**

Los helpers son servicios que nos ayudan a realizar algunos procedimientos dentro de nuestra aplicación, son solo métodos que están dentro de una clase y que se vincularan dentro de nuestro proyecto laravel, en este caso vamos a crear un helper que nos ayude con el control de Tokens de los usuarios que se loguean en el sistema. Para crear un helper seguimos los siguientes pasos:

1. Creamos la carpeta Helpers dentro del directorio App, allí vamos a guardar todos los helpers de nuestra API.
2. Dentro de Helper creamos una clase con el nombre *jwtAuth.php*
3. Dentro de la clase vamos a cargar los namespaces que voy a usar.
4. namespace App\Helpers;
5. use Firebase\JWT\JWT;
6. use Illuminate\Support\Facades\DB;
7. use App\Models\User;
8. Creamos una clase JwtAuth y dentro de ella almacenamos los métodos que nos ayudarán al control de Tokens. Es importante crear un constructor de la clase jwtAuth que contenga el valor de la clave con la cuál se puede codificar y decodificar nuestro token.

Class jwtAuth

{

    Public function \_\_construct()

    {

        $this->key = 'Esta\_eS\_miS\_perCl4ve';

    }

    public function signup($email,$password,$getToken=null)

        {

*//Buscar si existe el usuario con las credenciales.*

        $user = User::where([

            'email'     =>  $email,

            'password'  =>  $password

        ])->first();

*//Comprobar si son correctas.*

            $signup = false;

            if(is\_object($user))

                {

                    $signup = true;

                }

*//Generar el token con los datos del usuario identificado.*

        if($signup)

            {

                $token = array(

                'sub'         =>  $user->id,

                'email'       =>  $user->email,

                'name'        =>  $user->name,

                'surname'     =>  $user->surname,

                'description' =>  $user->description,

                'image'       =>  $user->image,

                'iat'         =>time(),

                'exp'         =>time()+(7\*24\*60\*60)

                );

                $jwt = JWT::encode($token,$this->key,'HS256');

*//var\_dump($jwt);die();*

                $decoded = JWT::decode($jwt,$this->key,['HS256']);

                if(is\_null($getToken))

                {

                    $data = $jwt;

                }

                else

                {

                    $data = $decoded;

                }

            }

        else

            {

                $data = array(

                'status'    =>  'Error',

                'message'   =>  'Login Incorrecto'

                );

            }

            return $data;

    }

    public function checkToken($jwt, $getIdentity = false)

    {

        $auth = false;

        try{

            $jwt = str\_replace('"','',$jwt);

            $decoded = JWT::decode($jwt,$this->key, ['HS256']);

        }catch(\UnexpectedValueException $e){

            $auth = false;

        }catch(\DomainException $e){

            $auth = false;

        }

        if(!empty($decoded) && is\_object($decoded) && isset($decoded->sub))

        {

            $auth = true;

        }

        else

        {

            $auth = false;

        }

        if($getIdentity)

        {

            return $decoded;

        }

        return $auth;

    }

}

1. Luego debemos cargar el Helper dentro de la librería de Laravel, lo hacemos de la siguiente manera:
   * En la terminal ponemos el siguiente CMD:

*php artisan make:provider JwtAuthServiceProvider*

* + Luego ingresamos al servicio en Providers/servicio y vamos al método register, ponemos lo siguiente:

require\_once app\_path().'/Helpers/JwtAuth.php';

* + Posteriormente cargamos el provider en la configuración del Framewrok. Config/app.php

*App\Providers\JwtAuthServiceProvider::class*

* + Por último, añadimos un alias para que pueda ser llamado fácilmente.

‘JwtAuth’ => App\Helpers\JwtAuth::class;

1. Para tener un control de los Tokens vamos a tener en cuenta las siguientes capas dentro de la clase JwtAuth. Esto lo realizaremos en un método que se llamará signup(), este método va a recibir el email, la password y un getToken para recibir los datos de usuario y para poder validar.

*public function signup($email,$password,$getToken=null)*

*{*

***//Buscar si existe el usuario con las credenciales.***

$user = User::where([

‘email’=>$email,

‘password’=>$password

])->first();

***//Comprobar si son correctas.***

$signup = false;

if(is\_object($user))

{

$signup = true;

}

***//Generar el token con los datos del usuario identificado.***

if($signup)

{

$token = array(

‘sub’=>$user->id,

‘email’=>$user->email,

‘name’=>$user->name,

‘lastname’=>$user->lastname,

‘iat’=>time(),

‘exp’=>time()+(7\*24\*60\*60)

);

$jwt = JWT::encode($token,$this->key,’HS256’);

$decoded = JWT::decode($jwt,$this->key,’HS256’)

}else

{

$data = array(

*‘status’ => ‘Error’,  
‘message’ => ‘Login Incorrecto*

);

}

***//devolver los datos decodificados o el token en función de un parámetro. (Dentro del condicional positivo)***

If(is\_null($getToken))

{

$data = $jwt;

}else

{

$data = $decoded;

}

Return $data;

1. Para que podamos utilizar este método entonces será necesario, en el controlador de usuarios crear un método login que envíe los datos del usuario a mi validador en el Helper.

*public function login(Request $request)*

*{*

*$jwtauth = new \JwtAuth();*

***//Recibir datos por POST***

*$json = $request->input('json',null);*

*$params = json\_decode($json);*

*$params\_array = json\_decode($json,true);*

***//Validamos la información***

*$validate = \Validator::make($params\_array, [*

*'email' => 'required|email',*

*'password' => 'required',*

*]);*

*if($validate->fails()){*

*$signup = array(*

*'status'=> 'error',*

*'code'=> 404,*

*'message'=> 'El Usuario no se ha podido identificar',*

*'errors'=>$validate->errors(),*

*);*

*}else*

*{*

***//Cifrar la password***

*$pwd = hash('sha256',$params->password);*

***//Devolver token o datos***

*$signup = $jwtauth->signup($params->email,$pwd);*

*if(!empty($params->getToken))*

*{*

*$signup = $jwtauth->signup($params->email,$pwd,true);*

*}*

*}*

*return response()->json($signup,200);*

*}*

1. Es importante crear también un método que me ayude a chequear en cada una de las paginas de mi aplicación si el Token es correcto o incorrecto, para esto vamos a crear en nuestro Helper un método que nos realice un chekeo del TOKEN.

*public function checkToken($jwt, $getIdentity = false)*

*{*

*$auth = false;*

*try{*

*$jwt = str\_replace('"','',$jwt);*

*$decoded = JWT::decode($jwt,$this->key, ['HS256']);*

*}catch(\UnexpectedValueException $e){*

*$auth = false;*

*}catch(\DomainException $e){*

*$auth = false;*

*}*

*if(!empty($decoded) && is\_object($decoded) && isset($decoded->sub))*

*{*

*$auth = true;*

*}*

*else*

*{*

*$auth = false;*

*}*

*if($getIdentity)*

*{*

*return $decoded;*

*}*

*return $auth;*

*}*

A partir de este momento los códigos estarán directamente escritos en la aplicación.

**Funcion Actualizar usuario (**ver UserController)

El método para actualizar el usuario se encuentra en el UserController y tiene 6 pasos fundamentales:

1. Comprobar si el usuario esta indetificado.
2. Recoger los datos por medio del método POST
3. Si el token es verdadero y el paramatro no esta vacío entonces sacamos el usuario identificado.
4. Validamos la información que vamos a actualizar
5. Quitamos los campos del array que no queremos actualizar.
6. Llamamos el modelo ORM para hacer la consulta en la Bd
7. Devolvemos los arrays con los resultados y el objeto JSON.

**Crear middleware de autenticación:**

Se usa para realizar una validación de la autenticación del usuario, para crearlo procedemos de la siguiente manera:

1. En la terminal procedemos a crear el middleware con el siguiente comando:

*php artisan make:middleware ApiAuthMiddleware*

1. En el nuevo middleware vamos a validar si el usuario se ha autenticado correctamente con las funciones que tenemos habilitadas para ello. y retornamos un continuar o un error dependiendo del resultado de la validación de la autenticación.

public function handle(Request $request, Closure $next)

    {

*//\* Comprobamos que el usuario esta identificado*

        $token = $request->header('Authorization');

        $jwtauth = new \JwtAuth();

        $checkToken = $jwtauth->checkToken($token);

        if($checkToken)

        {

            return $next($request);

        }

        else

        {

            $data = array(

                'status'    =>  'error',

                'code'      =>  400,

                'message'   =>  'El usuario no se ha identificado'

            );

            return response()->json($data, $data['code']);

        }

1. En el Kernel.php debemos registrar el middleware para poder usarlo en Laravel, por eso vamos a editar el Route Middleware para usarlo desde una ruta.

'api.auth' => \App\Http\Middleware\ApiAuthMiddleware::class,

1. Luego en la ruta, antes de llamar al controlador, debo indicarle el middleware que voy a utilizar y usarlo como namespace.

->middleware(ApiAuthMiddleware::class);

**Upload de avatar de usuario:**

Para subir el AVATAR de usuario seguimos los siguientes pasos:

1. Recibimos el archivo Request->file.

*//\*Recibimos el archivo a subir*

        $image = $request->file('file0');

1. Guardamos la imagen y le creamos un nombre usando el atribute Date.

*//\* Guardamos la imagen*

        if($image)

        {

            $image\_name = time().$image->getClientOriginalName();

            \Storage::disk('users')->put($image\_name, \File::get($image));

            $data = array(

                'status'    =>  'Sucess',

                'code'      =>  200,

                'image' =>  $image\_name

            );

        }

1. Mostramos mensaje de error o éxito.
2. else
3. {
4. $data = array(
5. 'status'    =>  'error',
6. 'code'      =>  400,
7. 'message'   =>  'Error al subir imagen'
8. );
9. }
10. return response()->json($data, $data['code']);

**Validar la imagen:**

Para validar la imagen utilizamos el siguiente código dentro del método para subir la imágen:

*//\*Validacion de imágen*

        $validate = \Validator::make($request->all(), [

            'file0'      =>     'required|image|mimes:jpg,jpeg,png,gif'

        ]);

*/\* Guardamos la imagen*

        if(¡$image || $validate->fails())

        {

            $data = array(

                'status'    =>  'error',

                'code'      =>  400,

                'message'   =>  'Error al subir imagen',

                'Error'     =>  $validate->errors()

            );

        }

**Método para sacar una imagen**

Con este metodo vas a poder sacar una imagen del disco:

public function getImage($filename)

    {

        $isset = \Storage::disk('users')->exists($filename);

        if($isset)

        {

*//\*Saco la imagen del disco y la guardo en la variable*

            $file = \Storage::disk('users')->get($filename);

            return new Response($file,200);

        }

        else

        {

            $data = array(

                'status'    =>  'Fallo',

                'code'      =>  404,

                'Mensaje'   =>  'No existe la  imagen'

            );

            return response()->json($data, $data['code']);

        }

    }

**Metodo para sacar todos los datos de un usuario de la BD**

public function  details($id)

    {

        $user = User::find($id);

        if(is\_object($user))

        {

            $data = array(

                'status'    =>  'Sucess',

                'code'      =>  202,

                'User'  =>  $user

            );

        }

        else

        {

            $data = array(

            'status'    =>  'Error',

            'code'      =>  404,

            'Mensaje'   =>  'El usuario no existe'

        );

        }

        return response()->json($data, $data['code']);

    }

Frontend: **Angular**

**Instalación de ANGULAR**

1. Instalar NODE JS

2. En la ventana de comandos ejecutar:

- Instalar última versión de NPM: npm install -g npm@latest

- Borrar la cache de NODEJS: npm cache clean --force

- Desactivar las auditorias de NPM: npm set audit false

- Desinstalar versiones anteriores de Angular:

npm uninstall -g angular-cli

npm uninstall -g @angular/cli

- Borrar la cache de NODEJS: npm cache clean --force

3. Instalación de Angular CLI en la última versión

- npm install -g @angular/cli

4. Generación de nuevo proyecto ANGULAR

- ng new [nombre]

1. **Instalamos dependencias**

**npm i bootstrap jquery -S**

npm i bootstrap

npm i jquery

npm i angular-file-uploader --force

npm i angular-froala-wysiwyg

1. Añadir los paquetes al Angular.json

"styles": [

"src/styles.css",

"node\_modules/bootstrap/dist/css/bootstrap.min.css",

"node\_modules/froala-editor/css/froala\_style.min.css",

"node\_modules/froala-editor/css/froala\_editor.pkgd.min.css"

 ],

 "scripts": [

"node\_modules/jquery/dist/jquery.min.js",

"node\_modules/bootstrap/dist/js/bootstrap.min.js",

"node\_modules/froala-editor/js/froala\_editor.pkgd.min.js"

 ]

npm update

ng serve para arrancar el sv.

**Creación de modelos:**

Creamos algunos modelos que van a representar algunas entidades que tenemos en nuestro Backend.

1. Creamos una carpeta en App que se llamara models.
2. Dentro de la carpeta debemos crear un modelo para cada entidad. (tabla de la base de datos). En minúscula.
3. Luego configuramos el interior del archivo de la siguente manera: los atributos son los campos de la base de datos.

*//todos DEFINO UNA CLASE Y PERMITO QUE SE PUEDA EXPORTAR*

export class user{

*//\*DEFINO EL CONSTRUCTOR Y CADA UNA DE LAS PROPIEDADES*

  constructor

(

      public *id*: number,

      public *name*: string,

      public *surname*: string,

      public *rol*: string,

      public *email*: string,

      public *password*: string,

      public *description*: string,

public *image*: string

  ){}

}

**Creación de los componentes:**

**Barra de navegación**

Lo primero que vamos a hacer es borrar el contenido de app.component.html y ponemos un navbar de los que trae Bootstrap V5 en este caso, posteriormente vamos a encerrar todo en un contenedor principal, DIV class container.

*<!-- todos CONTENEDOR PRINCIPAL -->*

<div class="container">

  <br>

*<!-- todos BARRA DE NAVEGACION -->*

<nav class="navbar navbar-expand-lg navbar-dark bg-dark">

  <div class="container-fluid">

    <a class="navbar-brand" href="#">

      <img src="../assets/images/angular.png" alt="" width="30" height="24" class="d-inline-block align-text-top">

      NG-BLOG

    </a>

    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-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 me-auto mb-2 mb-lg-0">

        <li class="nav-item">

          <a class="nav-link active" aria-current="page" href="#">Inicio</a>

        </li>

        <li class="nav-item">

          <a class="nav-link" href="#">Categorias</a>

        </li>

      </ul>

      <ul class="navbar-nav navbar-right">

        <li class="nav-item dropdown">

          <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-bs-toggle="dropdown" aria-expanded="false">

            Nombre de usuario

          </a>

          <ul class="dropdown-menu" aria-labelledby="navbarDropdown">

            <li><a class="dropdown-item" href="#">crear entrada</a></li>

            <li><a class="dropdown-item" href="#">crear categoria</a></li>

            <li><hr class="dropdown-divider"></li>

            <li><a class="dropdown-item" href="#">Mi perfil</a></li>

            <li><a class="dropdown-item" href="#">Ajustes</a></li>

            <li><a class="dropdown-item" href="#">Cerrar session </a></li>

          </ul>

        </li>

      </ul>

    </div>

  </div>

</nav>

</div>

**Componentes para login**

En la consola escribimos el siguiente comando:

ng g component components/login

Esto nos va crear una carpeta en la cual vamos a guardar todos los componentes y adicionalmente allí mismo se creara un componente llamado login. En el archivo login.ts añadimos, pero antes cambiamos el nombre del selector.

 public page\_title: string;

  constructor() {

    this.page\_title = 'Identificate'

Luego en la vista login.component.html podemos usar la variable page\_title

<h1>{{page\_title}}</h1>

Luego debo insertar mi componente para que funcione debajo del NAV en el componente principal, lo debes llamar de la misma manera que nombras el componente

<login></login>

El de registro se crea de la misma manera

**Configuración del Router Angular**

Creamos un archivo app.routing.ts en la carpeta App y luego creamos nuestro sistema de rutas de la siguiente manera:

*//TODOS IMPORT NECESARIOS*

import { ModuleWithProviders } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

*//TODOS IMPORTAR COMPONENTES*

import { LoginComponent } from "./components/login/login.component";

import { RegisterComponent } from "./components/register/register.component";

*//TODOS DEFINIR LAS RUTAS*

const appRoutes: Routes = [

  {path: '',component: LoginComponent},

  {path: 'inicio',component: LoginComponent},

  {path: 'login',component: LoginComponent},

  {path: 'register',component: RegisterComponent},

];

*//TODOS EXPORTAR LAS RUTAS*

export const appRoutingProviders: any[] = [];

export const routing: ModuleWithProviders<any> = RouterModule.forRoot(appRoutes);

Luego en el app.module.ts debo importar las rutas:

import { routing } from './app.routing';

import { appRoutingProviders } from './app.routing';

y cargarlas en los imports y los providers:

imports: [

    BrowserModule,

    routing

  ],

  providers: [

    appRoutingProviders

  ],

Luego en el componente principal usamos: <router-outlet></router-outlet> para que reconozca cual es la ruta a la cual estamos intentando acceder.

**Crear mas componentes:**

Creamos componentes para la página de inicio y errores de la misma manera en que lo hemos venido haciendo. Los errores se deben redireccionar a (\*\*) que significa que la página no existe.

**Crear enlaces dentro de Angular:**

Reemplazar el HREF por

[routerLink]="['/register']"

Con esto la página no se recargara y será bastante dinámica.

**Creamos formulario para registro:**

Debemos importar en app.module.ts los formularios de la siguiente manera

import { FormsModule } from '@angular/forms';

Luego añadirlo a los import.

imports: [

    BrowserModule,

    routing,

    FormsModule

  ],

Luego creamos el formulario de registro de la siguiente manera:

<div class="col-md-12 mt-3">

  <h1>{{page\_title}}</h1>

  <p>

    Registrate en nuestra plataforma para crear nuevas entradas y mucho más!!!

  </p>

*<!-- Todos FORMULARIO DE REGISTRO -->*

  <form class="col-md-5 ml-0 pl-0">

    <div class="form-group">

      <label for="name">Nombre</label>

      <input type="text" name="name" class="form-control">

    </div>

    <div class="form-group">

      <label for="surname">Apellidos</label>

      <input type="text" name="surname" class="form-control">

    </div>

    <div class="form-group">

      <label for="email">Email</label>

      <input type="text" name="email" class="form-control">

    </div>

    <div class="form-group">

      <label for="password">Contraseña</label>

      <input type="text" name="password" class="form-control">

    </div>

    <br>

    <input type="submit" value="Registrarme" class="btn btn-success" />

  </form>

</div>

**Convertir a formulario de Angular:**

En el componente de registro importamos el modelo de usuario:

import { User } from 'src/app/models/user';

Creamos un propiedad en el componente para guardar el formulario, luego la inicializamos en el constructor:

public user: User;

**Constructor**:

this.user = new User(1, '','','ROLE\_USER','','','','');

**Luego nos creamos un método para la acción del formulario:**

 onSubmit(*form*:any)

  {

    console.log(this.user);

  }

**Luego creamos el formulario de Angular de la siguiente manera:**

<div class="col-md-12 mt-3">

  <h1>{{page\_title}}</h1>

  <p>

    Registrate en nuestra plataforma para crear nuevas entradas y mucho más!!!

  </p>

*<!-- Todos FORMULARIO DE REGISTRO -->*

  <form class="col-md-5 ml-0 pl-0" #registerForm="ngForm"

  (ngSubmit)="onSubmit(registerForm)">

    <div class="form-group">

      <label for="name">Nombre</label>

      <input type="text" name="name" class="form-control" #name="ngModel"

      [(ngModel)]="user.name" required pattern="[a-zA-Z]+">

      <small \*ngIf="!name.valid && name.touched"

      class="invalid-feedback d-block">

        El nombre no es valido

      </small>

    </div>

    <div class="form-group">

      <label for="surname">Apellidos</label>

      <input type="text" name="surname" class="form-control" #surname="ngModel"

      [(ngModel)]="user.surname" required pattern="[a-zA-Z]+">

      <small \*ngIf="!surname.valid && surname.touched"

      class="invalid-feedback d-block">

      El apellido no es valido

    </small>

    </div>

    <div class="form-group">

      <label for="email">Email</label>

      <input type="text" name="email" class="form-control" #email="ngModel"

      [(ngModel)]="user.email" required pattern="[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$">

      <small \*ngIf="!email.valid && email.touched"

      class="invalid-feedback d-block">

      El Email no es valido

    </small>

    </div>

    <div class="form-group">

      <label for="password">Contraseña</label>

      <input type="password" name="password" class="form-control" #password="ngModel"

      [(ngModel)]="user.password" required>

      <small \*ngIf="!password.valid && password.touched"

      class="invalid-feedback d-block">

      La contraseña no es valida

    </small>

    </div>

    <br>

    <input type="submit" value="Registrarme" class="btn btn-success" />

  </form>

</div>

**Mejorar el formulario:**

Desactivar el boton de registro cuando el formulario no esta completamente lleno: se debe agregar la siguiente directiva en el input

[disabled]=registerForm.invalid

Vaciar el formulario cuando ya se envíe. Se debe agregar el siguiente comando en el formulario onSubmit

*form*.reset();

**Crear servicio para guardar la información:**

1. En el app.module.ts vamos a importar un nuevo modulo, el http client.

import { HttpClientModule} from '@angular/common/http';

1. Creamos una carpeta services en el directorio app y vamos a crear un servicio con el siguiente nombre: user.service.ts, allí vamos a hacer lo siguiente:
   1. Importar todo lo necesario para crear el servicio:

import { Injectable } from "@angular/core";

import { HttpClient } from "@angular/common/http";

import { HttpHeaders } from "@angular/common/http";

import { Observable } from "rxjs";

import { User } from "../models/user";

* 1. Definimos la clase del servicio injectable y pasamos el parametro httpclient en el constructor, luego nos creamos un método de prueba

@Injectable()

export class UserService

{

  constructor

  (

    public *\_http*: HttpClient

  )

  {

  }

  test()

  {

    return "Hola mundo desde un servicio!!";

  }

}

e. Luego lo testeamos en el metodo onSubmit

console.log(this.\_userService.test());

**Registrar usuario en la base de datos:**

**Creamos el global TS con la conexión al Back**

1. En primer lugar vamos a crear el fichero *global.ts* dentro de la carpeta *services* aquí vamos a tener configuraciones globales de algunas cosas.

export var global =

{

  url: 'sitco.local/api/employee/'

}

1. Importamos el ficherlo global en el service del usuario:

import { global } from "./global";

1. Inicializamos la URL del API en el servicio.

public url:string;

  constructor

  (

    public *\_http*: HttpClient

  )

  {

    this.url = global.url;

  }

1. Creamos el método de registro en el servicio

register(*employee*:any): Observable<any>

  {

*//todos Capturo el parametro Employee y lo convierto a JSON STRING*

    let json =  JSON.stringify(*employee*);

    let params = 'json='+json;

*//todos Defino las cabeceras*

    let headers = new HttpHeaders().set('Content-Type', 'application/x-www-form-urlencoded');

*//todos Luego vamos a hacer una petición AJAX*

    return this.\_http.post(this.url+'store', params, {headers:headers});

  }

1. Utilizo el método en el register.component.ts, dentro del onSubmit.

onSubmit(*form*:any)

  {

    console.log(*form*);

    this.\_employeeService.register(this.employee).subscribe

    (

*//todos 2 FUNCIONES DE CALLBACK, UNO PARA LA RESPUESTA Y OTRO PARA EL ERROR.*

*response* =>{console.log(*response*);*form*.reset();},

*error* => {console.log(<any>*error*);}

    );

  }

**Formulario de login:**

En primer lugar vamos a colocar la propiedad del objeto usuario en el constructor de login.ts, nos quedaría de la siguiente manera: Igualmente se debe crear un atributo de tipo público a nivel de la clase.

public user: User;

this.user = new User(1, '','','ROLE\_USER','','','','');

Luego importamos en el componente de login el modelo de usuarios y el servicio de usuarios

import { User } from 'src/app/models/user';

import { UserService } from 'src/app/services/user.service';

Posteriormente actualizamos el servicio en los providers del component y declaramos la variable privada en los parámetros del constructor para poder usar los métodos:

providers: [UserService]

private *\_userService*: UserService,

Luego de tener esto, vamos al formulario HTML para crear el mismo de la siguiente manera:

<div class="col-md-12 mt-3">

<h1>{{page\_title}}</h1>

<p>

  Identificate en la plataforma

</p>

<form class="col-md-5 ml-0 pl-0" #loginForm="ngForm" (ngSubmit)="onSubmit(loginForm)">

  <div class="form-group">

      <label for="email" >Correo electrónico</label>

      <input type="text" name="email" class="form-control" #email="ngModel"

      [(ngModel)]="user.email" required pattern="[a-z0-9.\_%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$">

      <small \*ngIf="!email.valid && email.touched"

      class="invalid-feedback d-block">

      El Email no es valido

    </small>

  </div>

  <div class="form-group">

      <label for="password">Contraseña</label>

      <input type="password" name="password" class="form-control" #password="ngModel"

      [(ngModel)]="user.password" required>

      <small \*ngIf="!password.valid && password.touched"

      class="invalid-feedback d-block">

      La contraseña no es valida

    </small>

    </div>

    <br>

    <input type="submit" value="Identificarse" class="btn btn-success"

    [disabled]=loginForm.invalid/>

</form>

</div>

Luego en el componente de login tenemos que hacer el OnSubmit que va identificar el usuario:

onSubmit(*form*:any)

  {

    this.\_userService.signup(this.user).subscribe

    (

*response* =>

      {

*//todos TOKEN*

        if(*response*.status!='error')

        {

          this.status = 'success';

          this.token = *response*;

*//TODOS OBJETO USUARIO IDENTIFICADO*

          this.\_userService.signup(this.user,true).subscribe

            (

*response* =>

              {

                this.identity = *response*;

*//TODOS USAR EL USUARIO IDENTIFICADO*

                console.log(this.token);

                console.log(this.identity);

              },

*error* =>

              {

                this.status = 'error';

                console.log(<any>*error*);

              }

            );

        }

        else

        {

          this.status = 'error';

        }

      },

*error* =>

      {

        this.status = 'error';

        console.log(<any>*error*);

      }

    )

  }

En el servicio debemos crear un método ***Signup*** el cuál vamos a llamar para que funcione todo el tema de login:

signup(*user*:any, *getToken*:any = null): Observable<any>

  {

    if(*getToken* != null)

    {

*user*.getToken = 'true';

    }

    let json = JSON.stringify(*user*);

    let params = 'json='+json;

    let headers = new HttpHeaders().set('Content-Type','application/x-www-form-urlencoded');

    return this.\_http.post(this.url+'login', params, {headers:headers});

  }

**Persistir usuario identificado:**

Para esto vamos a utilizar el Local Storage. Cuando se identifique el usuario, vamos a agregar en el onSubmit las propiedades LocalStorage

localStorage.setItem('token',this.token);

localStorage.setItem('identity',JSON.stringify(this.identity));

**Mostrar datos del usuario identificado**

Se va utilizar un método llamado GetIdentity que tendrá la misión de traer los datos del usuario identificado

getIdentity()

  {

    let identity = JSON.parse(localStorage.getItem('identity')||'{}');

    if(identity && identity != "undefined")

    {

        this.identity = identity;

    }

    else

    {

      this.identity = null;

    }

    return this.identity;

  }

También vamos a crear el método GetToken que tendrá la misión de traer el token del usuario identificado.

getToken()

  {

    let token = localStorage.getItem('token');

    if(token != 'undefined')

    {

      this.token = token;

    }

    else

    {

      this.token = null;

    }

    return this.token;

  }

Luego en el App.component.ts vamos a importar el servicio de usuario

import { UserService } from 'src/app/services/user.service';

Posteriormente se carga en el provider de la misma clase.

providers: [UserService]

Creamos las propiedades identity y token para posteriomente rellnarla y pasarla a la vista.

export class AppComponent {

  public title = 'blog';

  public identity:any;

  public token:any;

Luego en el constructor voy a crear la propiedad para poder utilizar el servicio y asignamos un valor a las propiedades

constructor

(

  public *\_userService*: UserService

)

{

  this.identity = this.\_userService.getIdentity();

}

Luego en el HTML en el lugar donde debe ir el nombre se debe poner lo siguiente:

{{identity.name + '' + identity.surname}}

Vamos a generar una condición para mostrar el mensaje únicamente cuando el usuario este identificado, para ellos justo en el UL que muestra el nombre vamos a generar la consulta:

 \*ngIf="identity && identity.name"

De la misma manera vamos a hacer para las opciones de registro y login, vamos a poner que cuando la identidad NO exista entonces no muestre la opción de registro y login.

\*ngIf="!identity && identity.name"

**Cerrar sessión**

Para realizar este procedimiento vamos a realizar un pequeño truco que nos va evitar crear un nuevo componente para cerrar la sessión. Lo primero que vamos a hacer es crear una ruta en el approuting.ts y va quedar de la siguiente manera:

{path: 'logout/:sure',component: LoginComponent},

Luego debemos importar en el login component los métodos del router de la siguiente manera:

import { ActivatedRoute, Router } from '@angular/router';

import { Params } from '@angular/router';

luego debo cargar los servicios en el constructor:

private *\_router*: Router,

private *\_route*: ActivatedRoute

Luego en el login component vamos a crear un método nuevo que se va llamar logout y en este se va contener todo el código para darle de baja a la sessión actual:

logOut()

  {

    this.\_route.params.subscribe(*params* =>

      {

        let logout = +*params*['sure'];

        if(logout == 1)

        {

          localStorage.removeItem('identity');

          localStorage.removeItem('token');

          this.identity = '';

          this.token = '';

*//TODOS REDIRECCION  A LA PAGINA PRINCIPAL*

          this.\_router.navigate(['inicio']);

        }

      }

      )

  }

Luego en el ngOnit del login component vamos a llamar la función de logout solo cuando le llega el parámetro sure por la URL

 ngOnInit(): void

  {

*//TODOS SE EJECUTA Y CIERRA SOLO CUANDO LLEGA EL PARAMETRO SURE POR LA URL*

    this.logOut();

  }

Luego en el HREF de salir vamos a colocar la directiva para poder redireccionar al método logout.

[routerLink]="['/logout/1']"

Luego vamos a crear una redirección en el app component de la siguiente manera, primero vamos a importar las siguientes clases: onInit para que siempre que se cargue ejecute algo y el Docheck para que luego de cierto tiempo ejecute una actualización de lo tengo:

import { Component, OnInit, DoCheck } from '@angular/core';

Luego en la clase AppComponent vamos a implementar lo siguiente:

export class AppComponent implements OnInit, DoCheck

Dentro del constructor vamos a darle valor al token

this.token = this.\_userService.getToken();

Luego vamos a copiar la asignación de variables del token y el identity en el método Dochek

this.identity = this.\_userService.getIdentity();

this.token = this.\_userService.getIdentity();

Se propone crear un método loaduser para no repetir tantas veces el mismo proceso:

loadUser()

{

  this.identity = this.\_userService.getIdentity();

  this.token = this.\_userService.getIdentity();

}

Luego en el login component .ts vamos a hacer una redirección una vez que me loguee

*//TODOS REDIRECCION  A LA PAGINA PRINCIPAL*

                this.\_router.navigate(['inicio']);

**Modificación de usuario:**

Para realizar este procedimiento debemos trabajar en el componente, la vista y la ruta, procedemos en primer lugar a crear un componente que se llame useredit y que va encargarse de contener todo el código para la edición de usuario.

*ng g component components/user-edit*

Posteriormente vamos a añadir la ruta en el archivo de rutas que hemos definido para nuestro proyecto de la siguiente manera:

import { UserEditComponent } from "./components/user-edit/user-edit.component";

Luego vamos a añadir la ruta en el archivo de la siguiente manera:

{path: 'ajustes',component: UserEditComponent},

Luego debo ir al menú html del app component y hacer un enlace Router Link a una sección de angular ajustes:

[routerLink]="['/ajustes']"

Creamos la vista con el formulario de los ajustes de usuario. Podemos utilizar la misma vista que hemos utilizado para el registro y editarla posteriomente como se indicará a continuación, no obstante antes es importante configurar algunos ajustas en el user-edit ts para poder continuar. Recordar definir el título de la página

constructor()

  {

    this.page\_title = 'Ajustes de usuario';

  }

También se debe crear una propiedad de tipo usuario e instanciarla para luego importar el modelo y poder reconocer los atributos del modelo de usuario.

public page\_title:string;

  public user: User;

  constructor()

  {

    this.page\_title = 'Ajustes de usuario';

    this.user = new User(1, '','','ROLE\_USER','','','','');

  }

Para rellenar los datos de usuario vamos a utilizar el servicio de usuarios que hemos creado y allí vamos a programar los métodos. Lo primero es importar el servicio y cargar el provider en user edit

import { UserService } from 'src/app/services/user.service';

providers: [UserService]

Luego creamos las propiedades para el token y la identidad del usuario logueado y pasamos por parámetro en el constructor la propiedad del servicio de usuarios, luego vamos a capturar el token y la identidad por medio del servicio de usuarios y luego la vamos a asignar al usuario de la clase para que rellene los datos en el formulario.

export class UserEditComponent implements OnInit {

  public page\_title:string;

  public user: User;

  public identity;

  public token;

  constructor

  (

    private *\_userService*: UserService

  )

  {

    this.identity = this.\_userService.getIdentity();

    this.token = this.\_userService.getToken();

    this.page\_title = 'Ajustes de usuario';

    this.user = new User(1, '','','ROLE\_USER','','','','');

    this.user = this.identity;

this.user = new User

    (

      this.identity.sub,

      this.identity.name,

      this.identity.surname,

      this.identity.role,

      this.identity.email,

      '',

      this.identity.description,

      this.identity.image

    );

  }

Luego vamos a crear un método en nuestro servicio de usuarios para modificar el usuario identificado:

update(*token*:any, *user*:any)

  {

    let json = JSON.stringify(*user*);

    let params = 'json='+json;

*//!DEFINIMOS LAS CABECERAS*

    let headers = new HttpHeaders()

                                  .set('Content-Type','application/x-www-form-urlencoded')

                                  .set('Authorization',*token*);

    return this.\_http.put(this.url+'user/update', params, {headers:headers});

  }

Luego vamos a llamar el método update desde el onSubmit del formulario de ajustes, lo hacemos de la siguiente manera:

onSubmit(*form*:any)

  {

    this.\_userService.update(this.token,this.user).subscribe

    (

*response* =>

      {

        if(*response && response.status*)

        {

          this.status = 'success';

*//Actualizar el usuario en la session*

          console.log(*response*);

          if(*response*.actualizado.name)

          {

            this.user.name = *response*.actualizado.name;

          }

          if(*response*.actualizado.surname)

          {

            this.user.surname = *response*.actualizado.surname;

          }

          if(*response*.actualizado.email)

          {

            this.user.email = *response*.actualizado.email;

          }

          if(*response*.actualizado.description)

          {

            this.user.description = *response*.actualizado.description;

          }

          if(*response*.actualizado.image)

          {

            this.user.image = *response*.actualizado.image;

          }

          this.identity = this.user;

          localStorage.setItem('identity',JSON.stringify(this.identity));

        }

        else

        {

          this.status = 'error';

        }

        console.log(*response*);

      },

*error*=>

      {

        this.status = 'error';

        console.log(<any>*error*);

      }

    );

  }

Vamos a utilizar la librería del froala wysiwyg para instalar un editor de texto en el área de modificación de la descripción. En primer lugar vamos a importar en el app module el Froala Editor Module y los añadirmos en los import de la siguiente manera:

import { FroalaEditorModule } from 'angular-froala-wysiwyg';

import { FroalaViewModule } from 'angular-froala-wysiwyg';

 FroalaEditorModule.forRoot(),

 FroalaViewModule.forRoot()

Luego en la vista del user edit debo colocar la siguiente directiva en el textarea para poder convertirlo en un froala.

[froalaEditor] [(froalaModel)]="user.description"

**Subida de imágenes y carga en la barra de nav:**

**(PENDIENTE POR DOCUMENTAR)**

**Gestión de categorias (PENDIENTE POR DOCUMENTAR)**

npm install -g @angular/cli