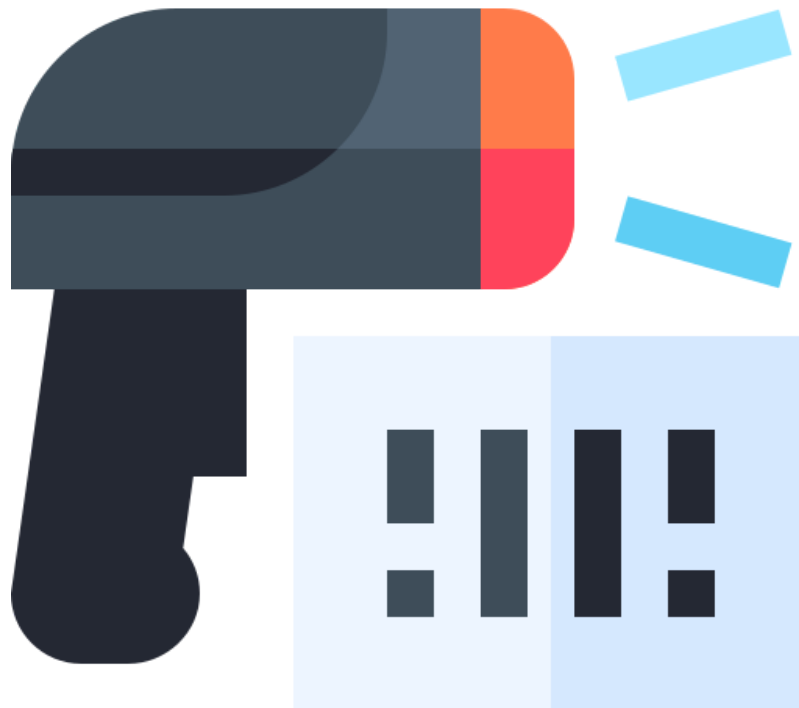


# MEMORIA TFG

*App para gestión de productos mediante código qr*



**José Carlos Rodríguez Sánchez**

10/01/2022

2.º DAW, INFORMÁTICA

# Tabla de Contenido

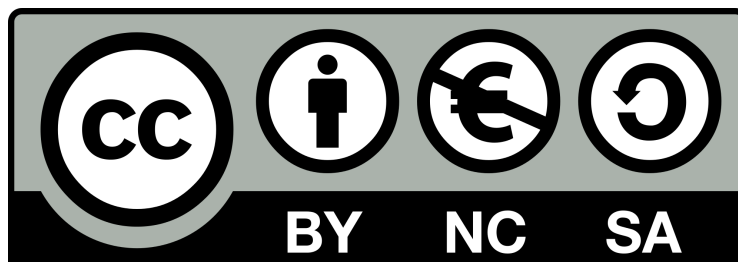
<b>LICENCIA</b>	<b>2</b>
<b>INTRODUCCIÓN</b>	<b>3</b>
<b>IMPLEMENTACIÓN</b>	<b>3</b>
ANÁLISIS	35
Lista de requerimientos iniciales	35
Actores del sistema	35
Diagrama de casos de uso	36
Descripción de cada caso de uso	37
MANUAL DE USUARIO	37
HERRAMIENTAS Y TECNOLOGÍAS EMPLEADAS	43
<b>PROPUESTAS DE MEJORA</b>	<b>45</b>
<b>VALORACIÓN PERSONAL</b>	<b>46</b>
TRABAJO EN GRUPO Y METODOLOGÍA SCRUM	46
VALORACIÓN CRÍTICA DE LA METODOLOGÍA EMPLEADO Y MEJORAS QUE INCLUIRÍAS	47
PUNTOS A DESTACAR DE TU PROYECTO	47
<b>REFERENCIAS</b>	<b>48</b>

## LICENCIA

Este proyecto se encuentra bajo la licencia de “Creative Commons” **Reconocimiento-NoComercial-CompartirIgual CC-BY-NC-SA**.

Esta licencia permite a otros modificar a partir de esta obra con fines no comerciales, siempre y cuando le reconozca la autoría y sus nuevas creaciones estén bajo una licencia con los mismos términos.

<https://creativecommons.org/licenses/by-nc/4.0/>



## INTRODUCCIÓN

La idea de este proyecto surgió de una necesidad.

Hace algunos meses, un familiar necesitaba ayuda para realizar un trabajo, el cual consistía en hacer una lista con los códigos de cada uno de los productos del almacén.

En este caso los productos eran botellas de diferentes bebidas, las cuales llevan un precinto de la agencia tributaria que las identifica de forma individual.

El primer día que fui a ayudar en esta tediosa labor, me dí cuenta al instante de que había un gran problema y creí que podría ayudar a solucionarlo, creando una aplicación para la gestión de los códigos en cuestión.

El problema comienza cuando la cantidad de botellas, y por lo tanto, códigos distintos que hay que listar, supera las 8.000 uds. y esto requería mucha mano de obra y mucho tiempo para realizarlo, sin contar con el factor humano, el cual generaba gran cantidad de errores con los códigos y hacía casi imposible que coincidieran los supuestos códigos recopilados con los que en realidad había en la lista.

Por lo tanto, decidí crear una aplicación que solucionase este problema y a su vez lo convertí en mi proyecto de final de grado superior.

La aplicación se ayuda de un lector de códigos QR, para realizar las lecturas de cada uno de los productos y una vez escaneados los guarda en la base de datos para posteriormente exportarlos a un archivo en formato xlsx (excel).

## IMPLEMENTACIÓN

El proyecto se compone de dos partes bien diferenciadas, la parte del servidor y la parte del cliente.

En esta ocasión decidí comenzar por la parte del servidor, la cual he desarrollado utilizando el framework de Laravel, ya que de este modo podría generar los endpoints necesarios y tenerlos preparados para mostrar los datos de la aplicación, lo que me ayudó bastante durante el desarrollo de la parte del cliente.

Para comenzar con el proyecto, primero hay que preparar el equipo con todos los

programas necesarios.

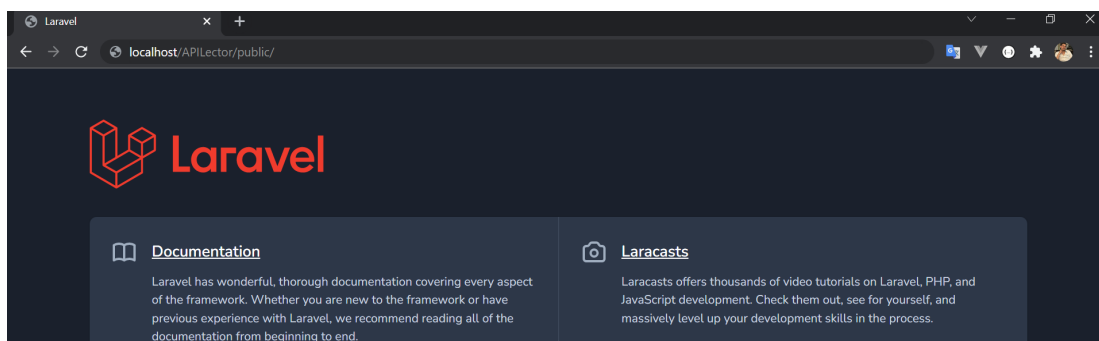
- En primer lugar instalaremos XAMPP, esto nos va a instalar algunos de los programas que necesitamos como: Apache, MariaDB y PHP.
- A continuación, necesitaremos instalar Composer, que nos ayudará con la gestión de paquetes necesarios para programar en PHP y también instalaremos NPM que también nos servirá como gestor de dependencias.
- Por último, necesitaremos una terminal, si bien en mi caso dispongo de la terminal de windows, he preferido instalar Git Bash, para lanzar los comandos.

Una vez tenemos instalados los programas y bien configurados, nos situaremos en el directorio htdocs de XAMPP, que es donde vamos a alojar la parte del servidor de nuestro proyecto y a través de la línea de comandos de Git lanzaremos el siguiente comando:

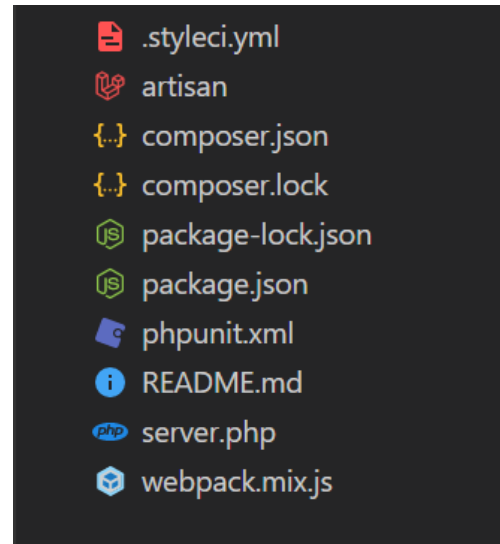
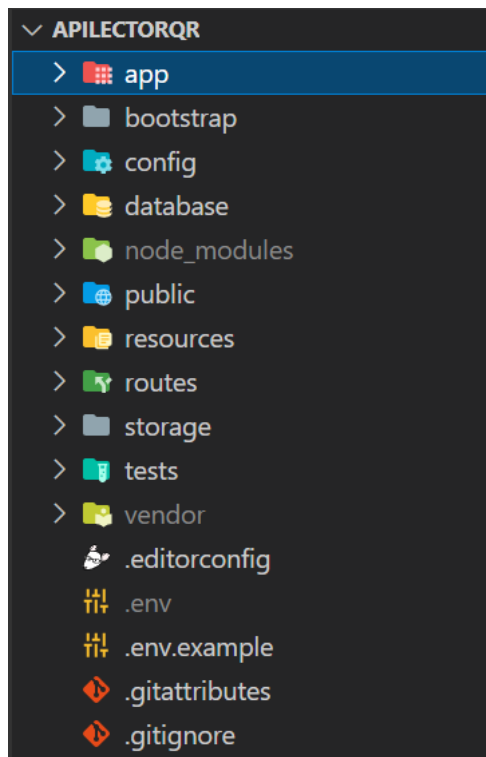
```
$ laravel new <nombre_proyecto>
```



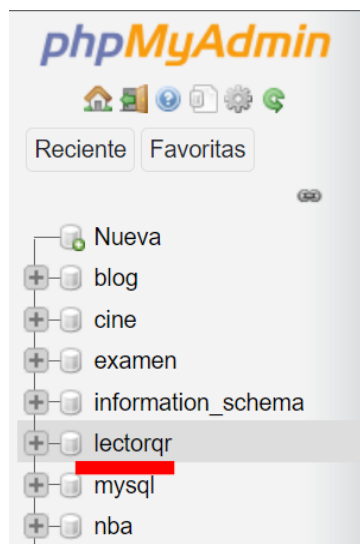
Una vez finalizada la instalación, ya podemos acceder desde el navegador para ver el contenido. Como para esta ocasión, sólo vamos a utilizarlo como servidor, no vamos a hacer nada con esta URL, en los próximos pasos, nos limitaremos a crear URLs también llamados endpoint, para gestionar los datos que guardaremos, borraremos, editaremos o leeremos en nuestra base de datos.



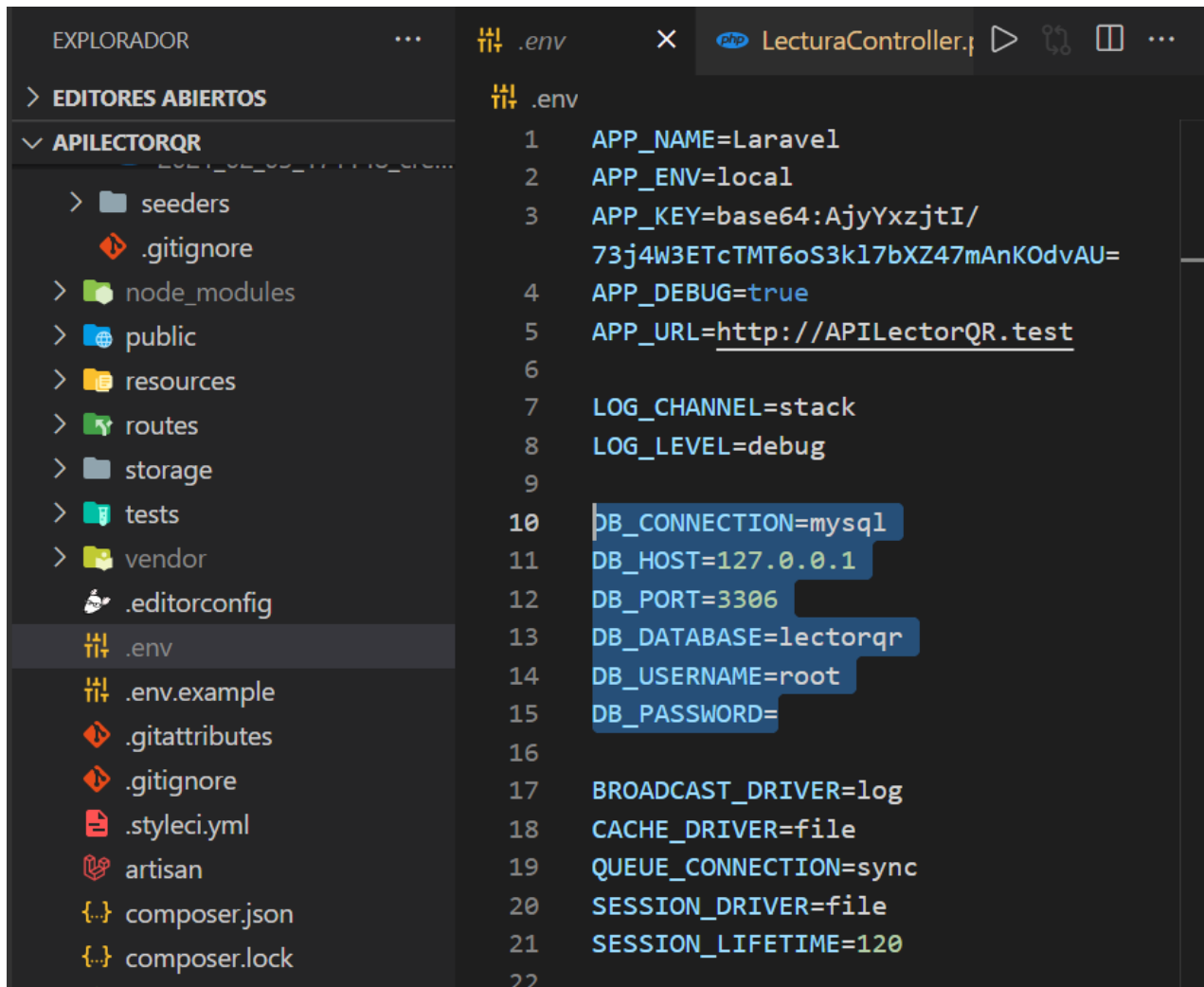
Esta instalación nos ha creado una estructura de carpetas como la siguiente:



El siguiente paso será crear la base de datos, donde alojaremos todos los datos relativos a nuestra aplicación.



Una vez creada, vamos a configurar el proyecto, de modo que pueda acceder a la nueva base de datos que acabamos de crear. Esto lo conseguiremos abriendo el archivo `.env` y añadiendo el tipo de conexión, la dirección IP, el puerto, el nombre de la base de datos, el nombre de usuario y por último la contraseña.



```
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:AjyYxzjtI/
  73j4W3ETcTMT6oS3k17bXZ47mAnKOdvAU=
4 APP_DEBUG=true
5 APP_URL=http://APILectorQR.test
6
7 LOG_CHANNEL=stack
8 LOG_LEVEL=debug
9
10 DB_CONNECTION=mysql
11 DB_HOST=127.0.0.1
12 DB_PORT=3306
13 DB_DATABASE=lectorqr
14 DB_USERNAME=root
15 DB_PASSWORD=
16
17 BROADCAST_DRIVER=log
18 CACHE_DRIVER=file
19 QUEUE_CONNECTION=sync
20 SESSION_DRIVER=file
21 SESSION_LIFETIME=120
22
```

A continuación, vamos a crear el controlador, la migración y el modelo de la lectura y del producto, que en este caso será una botella.

Con el fin de ser más productivos, vamos a utilizar ARTISAN, esta herramienta nos va a permitir crear todo lo que necesitamos, de un modo mucho más rápido y nos va a ayudar a tener una estructura de carpetas más organizada.

Para ello ejecutaremos los siguientes comandos, esta vez los comandos los lanzaremos directamente desde visual studio code, pero utilizando la herramienta Git Bash:

```
TERMINAL  CONSOLA DE DEPURACIÓN  PROBLEMAS  SALIDA

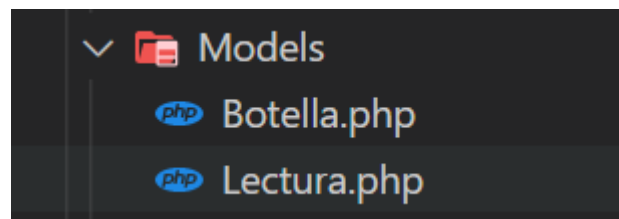
Jose@jose MINGW64 /c/xampp/htdocs/APILectorQR (main)
$ php artisan make:model Lectura -mcr
```

```
TERMINAL  CONSOLA DE DEPURACIÓN  PROBLEMAS  SALIDA

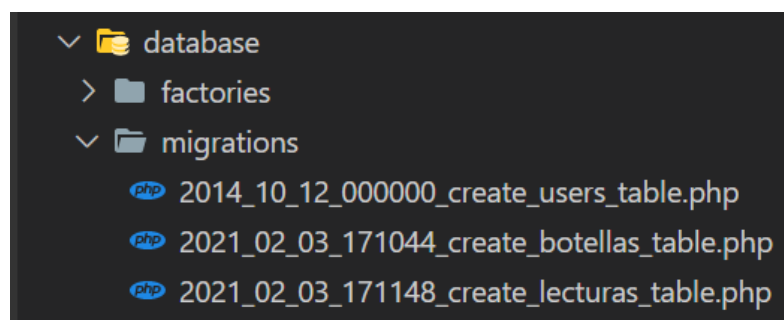
Jose@jose MINGW64 /c/xampp/htdocs/APILectorQR (main)
$ php artisan make:model Botella -mcr
```

Ahora podemos comprobar el gran valor que nos aporta la herramienta de ARTISAN, que ha creado todos los archivos necesarios para este desarrollo, con solo ejecutar dos simples comandos.

Modelos:

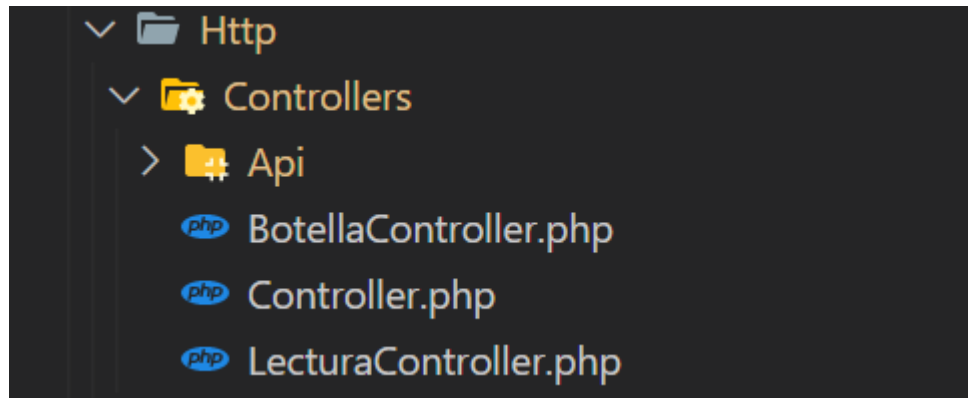


Migraciones:





Controladores:



Una vez tenemos todo creado, vamos a crear las tablas en la base de datos y esta tarea la vamos a realizar desde Laravel. Abrimos los archivos de las migraciones recién creadas y los modificamos con los datos necesarios para nuestro proyecto.

```
class CreateBotellasTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('botellas', function (Blueprint $table) {
            $table->id();
            $table->integer('id_lectura');
            $table->bigInteger('precinto');
            $table->timestamps();
        });
    }
}
```

```
class CreateLecturasTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('lecturas', function (Blueprint $table) {
            $table->id();
            $table->string('contenido');
            $table->integer('unidades_caja');
            $table->integer('cantidad_cajas');
            $table->timestamps();
        });
    }
}
```

Una vez modificamos estos archivos, ya estará listo para crear las tablas en la base de datos. Para ello, volveremos a utilizar ARTISAN, en este caso con el siguiente comando. En esta primera migración no es necesario añadirle “fresh”, pero tampoco perjudica, sin embargo, las próximas veces que utilicemos este comando, sí que le añadiremos “fresh”, lo cual revertirá todas las migraciones anteriores y luego ejecutará el comando en una base de datos limpia.

```
TERMINAL  CONSOLA DE DEPURACIÓN  PROBLEMAS  SALIDA

Jose@jose MINGW64 /c/xampp/htdocs/APILectorQR (main)
$ php artisan migrate:fresh
```

Llegados a este punto, vamos a indicar, qué atributos de los modelos que hemos creado se incluirán en un nuevo registro de la base de datos. Para esta tarea, en el archivo de cada modelo respectivamente, utilizaremos el atributo `$fillable`, que es un array que contendrá los campos de la tabla que se pueden completar mediante asignación masiva.

Con asignación masiva, nos referimos al envío de un array al modelo para crear un registro nuevo en la base de datos.

```
class Lectura extends Model
{
    use HasFactory;

    protected $fillable = [
        'contenido',
        'unidades_caja',
        'cantidad_cajas',
    ];
}
```

```
class Botella extends Model
{
    use HasFactory;

    protected $fillable = [
        'id_lectura',
        'precinto',
    ];
}
```

Como he basado este proyecto en una base de datos relacional, tengo que indicar cuales son las relaciones entre las tablas, no es una labor muy tediosa, al tratarse de una base de datos que sólo contiene dos tablas.

Nos situamos en el archivo correspondiente al modelo Lectura y como una lectura se refiere a una lista que va a contener muchas botellas le asignaremos “hasMany”.

```
public function botellas()  
{  
    return $this->hasMany('App\Models\Botella');  
}
```

Por otro lado, nos situaremos en el archivo correspondiente al modelo Botella, el cual se refiere a cada uno de los productos, en este caso botellas, que se incluyen dentro de una lectura, y le asignaremos “belongsTo”.

```
public function lectura()  
{  
    return $this->belongsTo('App\Models\Lectura');  
}
```

Ahora pasaremos a los controladores, los cuales se ocupan de gestionar la lógica de las peticiones.

En el controlador de Botella encontraremos una serie de funciones, las cuales hay que rellenar con la lógica que necesitamos, para nuestra aplicación.

- Index(), devuelve en un json, la lista completa de botellas, sin ningún tipo de filtro además de el código de estado de la petición, que en caso de ir todo bien será 200.

```
public function index()  
{  
    $botellas = Botella::get();  
    return response()->json($botellas, 200);  
}
```

- Store(Request \$request), utilizamos esta función para crear una nueva botella que le será pasada como parámetro y se almacenará en la base de datos, en esta ocasión, la llamada devuelve un json, con la propia botella recién creada y el código de estado de la petición.

```
public function store(Request $request)  
{  
    $botella = new Botella();  
    $botella->id_lectura = $request->id_lectura;  
    $botella->precinto = $request->precinto;  
    $botella->save();  
  
    return response()->json($botella, 201);  
}
```

- Destroy(Botella \$botella), elimina de la base de datos la botella que recibe por parámetro y devuelve 204 como código de estado.

```
public function destroy(Botella $botella)
{
    $botella->delete();
    return response()->json(null, 204);
}
```

Del mismo modo, ahora aplicaremos la lógica necesaria para el controlador de Lectura, utilizando las siguientes funciones.

- Index(), devuelve en un json, la lista completa de lecturas, sin ningún tipo de filtro además de el código de estado de la petición, que en caso de ir todo bien será 200.

```
public function index()
{
    $lecturas = Lectura::get();
    return response()->json($lecturas, 200);
}
```

- Store(Request \$request), utilizamos esta función para crear una nueva Lectura que le será pasada como parámetro y se almacenará en la base de datos, en esta ocasión, la llamada devuelve un json, con la propia lectura recién creada y el código de estado de la petición.

```
public function store(Request $request)
{
    $lectura = new Lectura();
    $lectura->contenido = $request->contenido;
    $lectura->unidades_caja = $request->unidades_caja;
    $lectura->cantidad_cajas = $request->cantidad_cajas;
    $lectura->save();

    return response()->json($lectura, 201);
}
```

- Update(Request \$request, Lectura \$lectura), esta función será la encargada de modificar los atributos de una lectura existente, devolverá la lectura con los nuevos datos.

```
public function update(Request $request, Lectura $lectura)
{
    $lectura->contenido = $request->contenido;
    $lectura->unidades_caja = $request->unidades_caja;
    $lectura->cantidad_cajas = $request->cantidad_cajas;
    $lectura->save();
    return response()->json($lectura);
}
```

- Destroy(Lectura \$lectura), elimina de la base de datos la lectura que recibe por parámetro y devuelve 204 como código de estado. En esta función, he creado la lógica para que se eliminen todas las botellas que dependan de esta lectura, pero para evitar perder horas de trabajo con un error, esta parte del código está comentada, de modo que si por error, el usuario, elimina una lista, todas las botellas de esta lectura permanecen en la base de datos.

```
public function destroy(Lectura $lectura)
{
    // $botellas = Botella::get()->where('id_lectura', $lectura->id);
    // $botellas->delete();
    $lectura->delete();
    return response()->json(null, 204);
}
```

- TopId(), esta función devuelve el último registro de la base de datos, de modo que el cliente lo llamará para asignar la id del nuevo registro y de este modo evitar que se repita.

```
public function topid()
{
    $topId = Lectura::get()
        ->last();
    return response()->json($topId, 200);
}
```

- IndexBotellas(\$id), esta función devuelve un json con todas las botellas que corresponden a una lectura, la cual se conoce por la id, que recibe como parámetro.

```
public function indexBotellas($id)
{
    $botellas = Botella::get()->where('id_lectura', $id);

    return response()->json($botellas, 200);
}
```



Para que todas estas funciones puedan ser alcanzadas desde el cliente, tenemos que crear las rutas, estas rutas las asignaremos desde el archivo api.php dentro de la carpeta routes

```
Route::get('/lecturas/top', [LecturaController::class, 'topid']);  
Route::get('/lecturas/{id}/botellas', [LecturaController::class, 'indexBotellas']);  
Route::apiResource('lecturas', LecturaController::class);  
  
Route::apiResource('botellas', BotellaController::class);
```

Llegados a este punto, la parte del servidor, ya estaría listo para dar respuestas a nuestras peticiones, por lo que es el momento de comenzar con el cliente que va a solicitar todas estas peticiones.

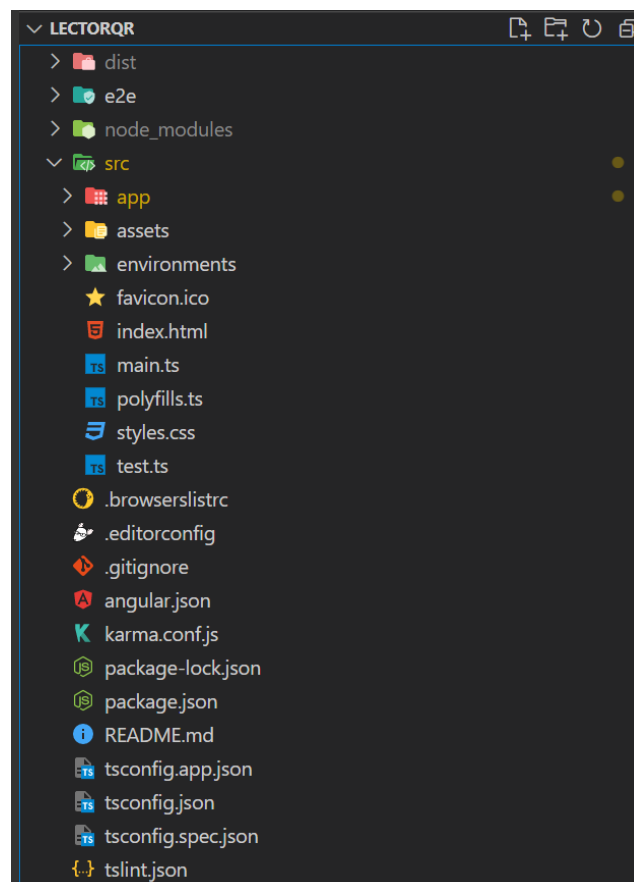
Para desarrollar el lado del cliente, he utilizado el framework de Angular. En primer lugar he instalado el CLI de angular utilizando el siguiente comando:

```
Jose@jose MINGW64 ~  
$ npm install -g @angular/cli|
```

Una vez instalado el CLI de angular, dispongo de una gran cantidad de comandos con los que desarrollar la aplicación. Empezaré por el comando necesario para crear un nuevo proyecto. Este comando solicitará información sobre las funciones que debe incluir el proyecto inicial y también instala las dependencias npm y paquetes necesarios.

```
Jose@jose MINGW64 ~  
$ ng new lectorQR|
```

Una vez termine la instalación, que tardará unos minutos, podemos ver la estructura de carpetas que se generan por defecto.



Una vez tengo la estructura, voy a instalar Bootstrap, este framework lo he utilizado para aplicar el diseño a las vistas de toda la aplicación. Lo instalo desde la consola con el siguiente comando:

```
TERMINAL  CONSOLA DE DEPURACIÓN  PROBLEMAS  3  SALIDA

Jose@jose MINGW64 ~/Desktop/DAW/ProyectoLectorQR/lectorQR (main)
$ npm install bootstrap jquery @popperjs/core
```

A continuación, abrimos el archivo angular.json y colocamos las siguientes instrucciones en “styles” y en “scripts”:

```
{
  "styles": [
    "node_modules/bootstrap/dist/css/bootstrap.min.css",
    "src/styles.css"
  ],
  "scripts": [
    "node_modules/jquery/dist/jquery.min.js",
    "node_modules/@popperjs/core/dist/umd/popper.min.js",
    "node_modules/bootstrap/dist/js/bootstrap.min.js"
  ]
}
```

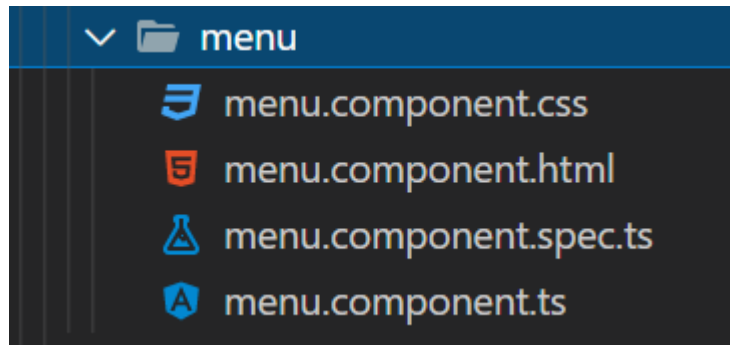
Con esto, ya tengo todo listo para comenzar a desarrollar.

En la carpeta src/app es donde se aloja el código del proyecto, esta carpeta contiene los componentes que forman el proyecto, los modelos y servicios entre otros archivos.

Gracias al CLI de angular, crear los componentes y servicios es una tarea casi automática con los siguientes comandos:

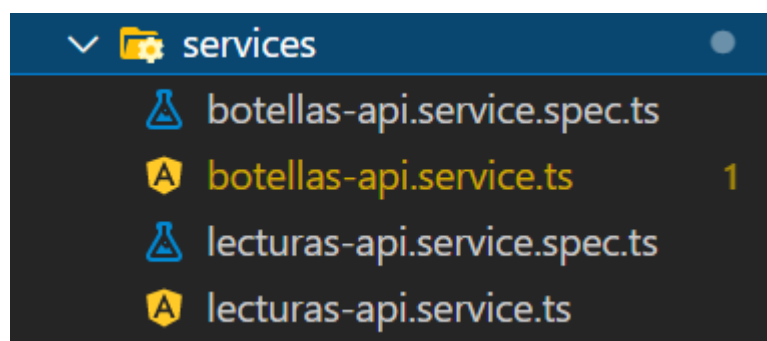
```
Jose@jose MINGW64 ~/Desktop/DAW/ProyectoLectorQR/lectorQR (main)
$ ng generate component <nombredelcomponente>
```

El anterior comando, genera una carpeta con el nombre del componente y en su interior, crea cuatro archivos, correspondientes con los estilos (esto dependerá de la configuración al momento de crear el proyecto, en mi caso css, aunque no lo utilizo, ya que los estilos se los doy mediante bootstrap), el cuerpo del componente en html, la lógica del componente en typescript y un último archivo para los test.



```
Jose@jose MINGW64 ~/Desktop/DAW/ProyectoLectorQR/lectorQR (main)
$ ng generate service services/<nombredelservicio>
```

Con el comando de la imagen anterior, se generan dentro de la carpeta services dos archivos por cada servicio, ambos en typescript, uno contiene la lógica y como en el caso de los componentes, el otro archivo está previsto para hacer test. El servicio será nuestro enlace con el servidor, desde aquí se hacen las llamadas.

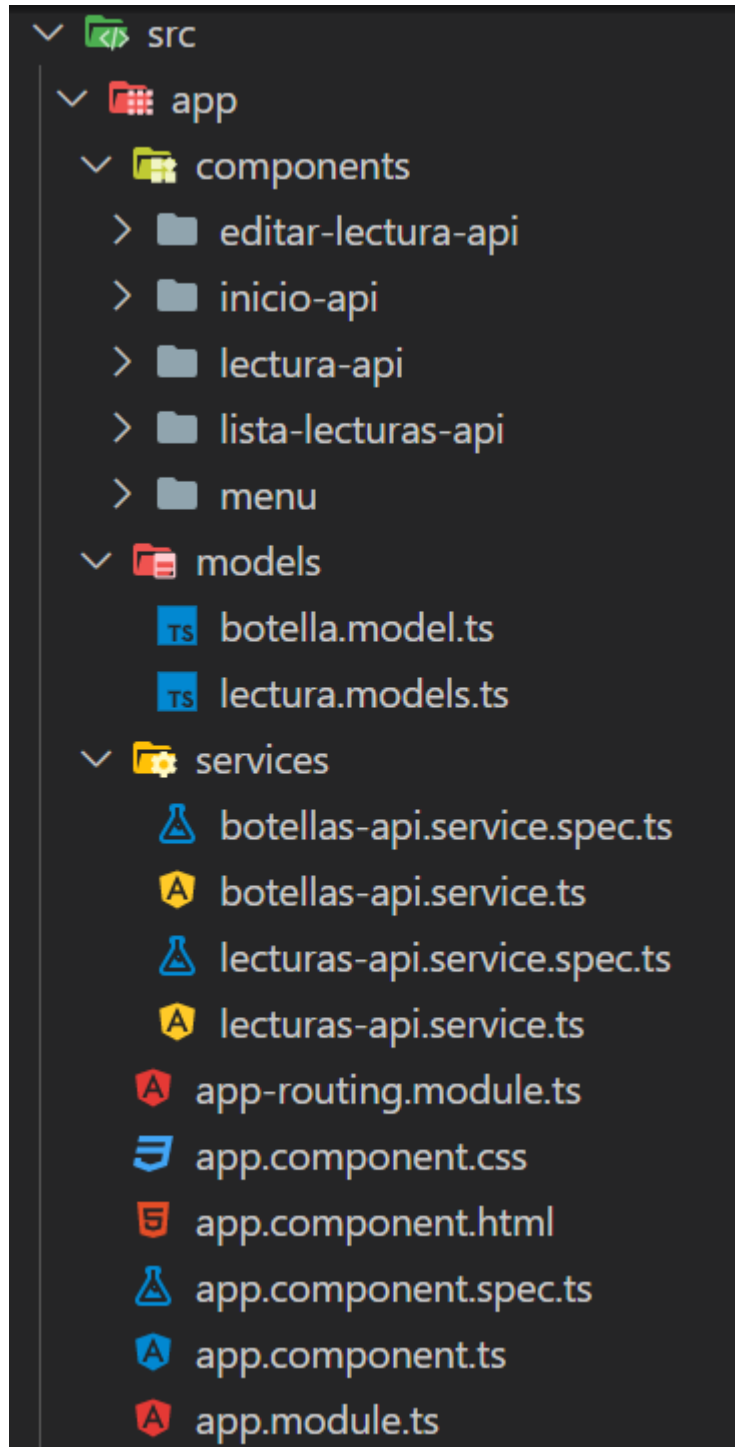


Estos nuevos módulos se declaran como clases TypeScript y se añaden automáticamente en la propiedad imports, además de importar cada componente en el archivo app.module.ts.

```
app.module.ts 1 X
c > app > app.module.ts > AppModule
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { HttpClient, HttpClientModule } from '@angular/common/http';
4
5 import { AppRoutingModule } from './app-routing.module';
6 import { AppComponent } from './app.component';
7 import { MenuComponent } from './components/menu/menu.component';
8 import { FormsModule } from '@angular/forms';
9 import { ListaLecturasApiComponent } from './components/lista-lecturas-api/lista-lecturas-api.component';
10 import { InicioApiComponent } from './components/inicio-api/inicio-api.component';
11 import { LecturasApiService } from './services/lecturas-api.service';
12 import { LecturaApiComponent } from './components/lectura-api/lectura-api.component';
13 import { BotellasApiService } from './services/botellas-api.service';
14 import { EditarLecturaApiComponent } from './components/editar-lectura-api/editar-lectura-api.component';
```

```
@NgModule({
  declarations: [
    AppComponent,
    MenuComponent,
    ListaLecturasApiComponent,
    InicioApiComponent,
    LecturaApiComponent,
    EditarLecturaApiComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [ LecturasApiService, BotellasApiService ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Tras crear todos los componentes, modelos y servicios, en la siguiente imagen se ve, cual es el resultado del árbol de directorios.



Cuando abrimos la aplicación, comienza con el archivo `index.html`, en él se encuentra la estructura de html y en la etiqueta body, una sola etiqueta llamada `<app-root>`. Podríamos decir que en esta etiqueta se encuentra toda la aplicación.

```
index.html ×
src > index.html > ...
1  <!doctype html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>LectorQR</title>
6    <base href="/">
7    <meta name="viewport" content="width=device-width, initial-scale=1">
8    <link rel="icon" type="image/x-icon" href="favicon.ico">
9  </head>
10 <body>
11   <app-root></app-root>
12 </body>
13 </html>
```

Igual que los componentes creados anteriormente, esta etiqueta, se compone por cuatro archivos.

```
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
```

En primer lugar nos fijamos en el archivo que contiene la lógica del componente, `app.components.ts`.

```
src > app > app.component.ts > ...  
1  import { Component } from '@angular/core';  
2  
3  @Component({  
4    selector: 'app-root',  
5    templateUrl: './app.component.html',  
6    styleUrls: ['./app.component.css']  
7  })  
8  export class AppComponent {  
9    title = 'lectorQR';  
10 }  
11
```

En primer lugar está la sentencia de importación de los elementos que utiliza el componente, a continuación, el decorador `@Component` que contiene tres metadatos, el metadato `selector`, es el que define el nombre de la etiqueta html, el metadato `templateUrl`, indica la ruta al archivo que contiene la vista del componente en html y por último, el metadato `styleUrls`, contiene un array con las rutas de las hojas de estilos del componente.

Finalmente `export` es la clase, aquí se incluye la lógica del componente.

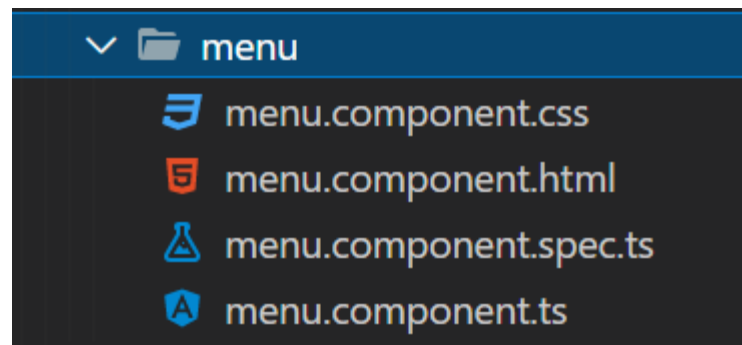
En la siguiente imagen se puede ver el archivo html, que contiene la vista del componente.

```
app.component.html X  
src > app > app.component.html > app-menu  
Go to component  
1  <app-menu></app-menu>  
2  
3  <router-outlet></router-outlet>  
4
```



En esta primera vista, se pueden ver dos etiquetas:

- `<app-menu>`: Esta etiqueta es un componente personalizado, se encuentra en la carpeta `componentes/menu`. Este componente se va a mostrar durante toda la aplicación, es el componente responsable de mostrar el menú. Como cualquier otro componente, el componente del menú, está compuesto por los archivos mencionados anteriormente.



- `<router-outlet>`: Esta etiqueta es la que nos permite poder navegar entre los diferentes componentes de forma dinámica. Para conseguir esto, tenemos que añadir las rutas en el archivo `app-routing.module.ts`, en la constante `routes`, como se ve en la siguiente imagen.

```
src > app > TS app-routing.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { EditarLecturaApiComponent } from './components/editar-lectura-api/editar-lectura-api.component';
4  import { InicioApiComponent } from './components/inicio-api/inicio-api.component';
5  import { LecturaApiComponent } from './components/lectura-api/lectura-api.component';
6  import { ListaLecturasApiComponent } from './components/lista-lecturas-api/lista-lecturas-api.component';
7
8  const routes: Routes = [
9    { path: '', component: ListaLecturasApiComponent },
10   { path: 'inicioApi', component: InicioApiComponent },
11   { path: 'listaApi', component: ListaLecturasApiComponent },
12   { path: 'lecturaApi/:lectura', component: LecturaApiComponent },
13   { path: 'lectura/:lectura/editar', component: EditarLecturaApiComponent },
14   { path: '**', redirectTo: '', pathMatch: 'full' },
15 ];
16
17 @NgModule({
18   imports: [RouterModule.forRoot(routes)],
19   exports: [RouterModule]
20 })
21 export class AppRoutingModule { }
22
```

Antes de analizar los componentes, vamos a ver los dos modelos que han hecho falta para esta aplicación:

- **Modelo Botella**, este modelo está compuesto por cuatro atributos, en los cuales, se puede ver de qué tipo son, en este caso todos serán de tipo número, también se puede apreciar en dos de los atributos un signo de interrogante "?", esto hace que el atributo sea opcional.

```
src > app > models > TS botella.model.ts > Botella
1  export interface Botella {
2      id_lectura: number;
3      id?: number;
4      precinto: number;
5      nEnCaja?: number;
6  }
```

- **Modelo lectura**, este modelo, se ha tenido que crear en segundo lugar, ya que uno de sus atributos es un array de botella, por lo que he tenido que importar el modelo botella, como se ve en la primera línea, junto con la ruta, donde se encuentra dicho modelo, esto se tendría que hacer por cada modelo que fuese necesario.

```
src > app > models > TS lectura.models.ts > ...
1  import { Botella } from "../botella.model";
2
3  export interface Lectura {
4      id?: number;
5      contenido: string;
6      unidades_caja: number;
7      cantidad_cajas: number;
8      botellas?: Botella[];
9  }
```

Una vez hemos visto los modelos, veremos los servicios, que en este caso, coinciden, un servicio por cada modelo:

- **Botellas-api-service**, en este servicio se utiliza el modelo botella visto anteriormente, por lo que será necesario volverlo a importar y también importamos **HttpClient** para las peticiones al servidor. Podemos ver la clase llamada **BotellasApiService**, que contiene tres literales que formarán las rutas con las que se realizan las llamadas, a continuación, **HttpClient** se inyecta en el constructor del servicio. A partir de aquí, se crean las funciones que hacen la petición a la base de datos y devuelven los datos solicitados o ejecutan acciones en la base de datos, como puede ser crear, editar o eliminar una botella.

```
1  import { HttpClient } from '@angular/common/http';
2  import { Injectable } from '@angular/core';
3  import { Observable } from 'rxjs';
4  import { Botella } from '../models/botella.model';
5
6  @Injectable({
7    providedIn: 'root'
8  })
9  export class BotellasApiService {
10
11    path1: string = "http://lector/api/lecturas/";
12    path2: string = "/botellas";
13
14    pathBotellas: string = "http://lector/api/botellas";
15
16    constructor(
17      private http: HttpClient
18    ) { }
19
20
21    getBotellas(): Observable<Botella[]> {
22      return this.http.get<Botella[]>(this.pathBotellas);
23    }
24    getBotellasLectura(id:number): Observable<Botella[]> {
25      return this.http.get<Botella[]>(this.path1 + id + this.path2);
26    }
27
28    addBotella(botella: Botella): Observable<Botella> {
29      return this.http.post<Botella>(this.pathBotellas, botella);
30    }
31
32    deleteBotella(id: number): Observable<Botella> {
33      return this.http.delete<Botella>(this.pathBotellas + '/' + id );
```

- **Lecturas-api-service**, hacemos las importaciones necesarias como en el servicio anterior, pero en este caso importamos el modelo “lectura”. Este servicio será el responsable de realizar las acciones necesarias para gestionar las lecturas.

```
src > app > services > TS lecturas-api.service.ts > LecturasApiService > getLectura
1  import { HttpClient } from '@angular/common/http';
2  import { Injectable } from '@angular/core';
3  import { Observable } from 'rxjs';
4  import { catchError, retry, } from 'rxjs/operators';
5  import { Lectura } from '../models/lectura.models';
6
7  @Injectable({
8    providedIn: 'root'
9  })
10 export class LecturasApiService {
11
12    path: string = "http://lector/api/lecturas";
13
14    constructor(
15      private http: HttpClient
16    ) { }
17
18    getAllLecturas(): Observable<Lectura[]> {
19      return this.http.get<Lectura[]>(this.path);
20    }
21
22    getLectura(id: number): Observable<Lectura> {
23      return this.http.get<Lectura>(this.path + '/' + id);
24    }
25
26    addLectura(lectura: Lectura): Observable<Lectura> {
27      return this.http.post<Lectura>(this.path, lectura);
28    }
29
30    editLectura(lectura: Lectura): Observable<Lectura> {
31      return this.http.put<Lectura>(this.path + '/' + lectura.id, lectura);
32    }
33
34    deleteLectura(id:number): Observable<Lectura> {
35      return this.http.delete<Lectura>(this.path + '/' + id);
36    }
37
38    getTopId(): Observable<Lectura> {
39      return this.http.get<Lectura>(this.path + '/top');
40    }
41  }
```

Ahora que tenemos los modelos de los objetos con los que vamos a tratar y los servicios están configurados para hacer las peticiones necesarias al servidor, vamos a analizar los componentes, que serán los que interactúan directamente con el usuario.

- **Componente del Menú:**

- **Archivo html** que contiene la estructura de la vista del menú, se puede ver como los estilos están asignados mediante clases, las cuales están creadas a partir del framework de bootstrap. En cuanto a la forma de navegar entre vistas, utilizamos la referencia del **[routerLink]** donde le asignaremos el path que corresponda según el componente al que queremos navegar. Para mejorar la experiencia de la navegación, utilizamos la clase **“routerLinkActive”** de bootstrap para que active de manera dinámica el ítem del menú según la vista en la que nos encontramos.

```
src > app > components > menu > menu.component.html > ...
1  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
2    <a class="navbar-brand" href="">Lector QR</a>
3    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav"
4    aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
5      <span class="navbar-toggler-icon"></span>
6    </button>
7    <div class="collapse navbar-collapse" id="navbarNav">
8      <ul class="navbar-nav">
9        <li class="nav-item">
10         <a class="nav-link" [routerLink]='["listaApi"]' routerLinkActive='active'>Lista de lecturas</a>
11        </li>
12      </ul>
13    </div>
14    <a class="btn btn-success " [routerLink]='["inicioApi"]' routerLinkActive='active'>Nueva Lectura</a>
15  </nav>
```

- **Archivo TypeScript**, para el menú. No he necesitado aplicar ningún tipo de lógica, aun así, se puede ver el selector, donde se asigna el nombre que se tiene que usar, como si fuese una etiqueta de html, para presentar el componente en los archivos html.

```
src > app > components > menu > TS menu.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-menu',
5    templateUrl: './menu.component.html',
6    styleUrls: ['./menu.component.css']
7  })
8  export class MenuComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit(): void {
13   }
14
15 }
```

- **Componente de inicio:**

- **Archivo html**, es aquí donde se genera el formulario, que será el encargado de crear la nueva lista de lectura. Este archivo contiene tres input con la directiva **ngModel** asociada al nombre de la variable que corresponde en cada caso. Este es el modo de pasar datos al archivo Typescript del componente para poder tratarlos. Por último un botón que ejecuta una función que está declarada en el archivo que contiene la lógica.

```
src > app > components > inicio-api > inicio-api.component.html > h3.text-center.mt-3
1  <h3 class="text-center mt-3">Nueva lectura</h3>
2
3  <div class="p-5">
4    <div class="form-group">
5      <label>Nombre de producto</label>
6      <input type="text" class="form-control" name="producto" id="producto" autocomplete="off" [(ngModel)]="producto" >
7      <small class="form-text text-muted">Ejemplo: VODKA SMIRNOFF RED 37,5º 1L</small>
8    </div>
9    <div class="form-group">
10     <label>Cantidad de unidades por caja</label>
11     <input type="text" class="form-control" name="unidadesCaja" id="unidadesCaja" autocomplete="off" [(ngModel)]="unidadesCaja" >
12     <small class="form-text text-muted">Introducir sólo números</small>
13   </div>
14   <div class="form-group">
15     <label>Número de cajas</label>
16     <input type="text" class="form-control" name="totalCajas" id="totalCajas" autocomplete="off" [(ngModel)]="totalCajas" >
17     <small class="form-text text-muted">Introducir sólo números</small>
18   </div>
19
20   <button class="btn btn-primary" (click)="inicioLectura()">Iniciar lectura</button>
21 </div>
22
```

- **Archivo TypeScript**, en las primeras líneas de este archivo se declaran los imports que serán necesarios para el desarrollo, como ya pasaba en el componente del menú y como pasa en el resto de componentes.

```
src > app > components > inicio-api > TS inicio-api.component.ts > InicioApiComponent > producto
1  import { Component, OnInit } from '@angular/core';
2  import { Router } from '@angular/router';
3  import { Lectura } from 'src/app/models/lectura.models';
4  import { LecturasApiService } from 'src/app/services/lecturas-api.service';
5
```

A continuación, justo debajo del decorador **@Component**, inicia la clase, en primer lugar declaramos las variables que serán necesarias para el desarrollo.

```
11  export class InicioApiComponent implements OnInit {
12
13      producto:string;
14      unidadesCaja:number;
15      totalCajas:number;
16      lectura:Lectura;
17      topId:number;
18
```

Seguidamente, en el constructor declaramos las variables para instanciar las clases que nos van a hacer falta. En este caso necesitaremos “LecturasApiService” que nos va a proporcionar todas las funciones que hemos creado en el servicio y se encargará de hacer las peticiones al servidor, en función de las necesidades del componente.

La clase Router, será la encargada de la navegación entre las diferentes vistas.

```
    constructor(
        private lecturasApiServ: LecturasApiService,
        private myRoute: Router
    ) { }
```

Finalmente, continuamos con la lógica de la vista, tenemos una función llamada `inicioLectura()`, esta será la encargada de comprobar que los campos del formulario no están vacíos y en su caso crear el objeto de tipo lectura. Una vez creado el objeto, se harán uso de las funciones de la clase `LecturasApiService`, para enviar esta información al servidor. En caso de que alguno de los campos esté vacío, se mostrará una alerta informando del error. Dentro de esta función, se llama a `asignaId()` para saber cual es la id siguiente en la base de datos, la asigna a esta lectura y navega hasta el siguiente componente de la aplicación.

```
inicioLectura() {  
  if (this.producto && this.unidadesCaja && this.totalCajas) {  
    this.lectura = {  
      id: this.topId,  
      contenido: this.producto,  
      unidades_caja: this.unidadesCaja,  
      cantidad_cajas: this.totalCajas,  
    };  
  
    this.lecturasApiServ.addLectura(this.lectura)  
      .subscribe(lectura => {  
        this.lectura = lectura;  
        this.asignaId();  
      });  
  
  } else {  
    alert('Debe rellenar todos los campos');  
  }  
}  
  
asignaId() {  
  this.lecturasApiServ.getTopId()  
    .subscribe(lectura => {  
      this.topId = lectura.id;  
      console.log(this.topId);  
      this.myRoute.navigate(['lecturaApi/' + this.topId]);  
    })  
}
```



- **Componente de lectura:**

- **Archivo html**, contiene la vista de la lista de lectura seleccionada. Para mostrar las propiedades del objeto lectura y los objetos botella, utilizamos la interpolación.

```
<div class=" my-2 mx-4 ">
  <div class="d-flex justify-content-between">
    <h4>{{ lectura?.contenido}} - {{lectura?.cantidad_cajas}} cajas - {{lectura?.unidades_caja}} Unidades/caja</h4>
    <span class="float-right mr-lg-5 mt-1">{{numBotellas}} de {{lectura?.cantidad_cajas*lectura?.unidades_caja}}</span>
  </div>
</div>
```

También hemos utilizado la directiva **ngFor**, para realizar las iteraciones y presentar las botellas en la tabla, otra de las directivas utilizadas en la tabla es **ngStyle**, la cual nos ha proporcionado la posibilidad de asignar unos estilos distintos según la lógica asignada en dicha directiva.

```
<tbody id="tableBody">
  <tr *ngFor="let botella of objectKeys(botellas); index as i"
    [ngStyle]='{"border-top": i % lectura.unidades_caja == 0 ? "3px solid black" : ""}'>
    <td scope="row" class="h4">{{i+1 + '/' + lectura?.cantidad_cajas*lectura?.unidades_caja}}</td>
    <td class="h1">{{i % lectura.unidades_caja + 1}}</td>
    <td class="h4">{{botella?.precinto}}</td>
    <td>
      <button class="btn btn-danger mr-2" (click)='eliminarBotella(botella)'+>eliminar</button>
    </td>
  </tr>
```

- **Archivo TypeScript**, en esta ocasión, a parte de importar los servicios y modelos necesarios, hemos importado una librería que nos brinda la posibilidad de exportar nuestra tabla generada en el archivo html a un archivo con formatoxlsx (excel).

```
o > components > lectura-api > TS lectura-api.component.ts > LecturaApiComponent
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router } from '@angular/router';
import { Botella } from 'src/app/models/botella.model';
import { Lectura } from 'src/app/models/lectura.models';
import { BotellasApiService } from 'src/app/services/botellas-api.service';
import { LecturasApiService } from 'src/app/services/lecturas-api.service';
import { ChangeDetectorRef } from '@angular/core';
import * as XLSX from 'xlsx';
```

Dentro de la clase se declaran todas las variables necesarias y dentro del constructor instanciamos una nueva clase llamada **ActivatedRouter**, esta clase nos proporciona las herramientas necesarias para recuperar la id de la dirección url y poder recuperar la lectura seleccionada.

```
export class LecturaApiComponent implements OnInit {

  botellas: Array<Botella> = [];
  numBotellas: number;
  botella: Botella;
  routeLectura: number;
  lectura: Lectura;
  url: string;
  fileName = '';
  topId: number;
  cajas: any = 1;

  constructor(
    private lecturaApiServ: LecturasApiService,
    private botellaApiServ: BotellasApiService,
    private rutaActiva: ActivatedRoute,
    private myRoute: Router,
  ) { }
```

Ahora llegamos al método **ngOnInit()**, que hasta el momento no habíamos utilizado, este método permite ejecutar todo el código que contenga en el momento en el que el componente haya cargado.

```
ngOnInit(): void {  
  this.cargaLectura();  
  this.cargaBotella();  
}
```

La primera función será la encargada de solicitar al servicio todos los datos relativos a la lectura actual, para ello, se ayuda de la propiedad **params** de la clase **activatedRoute** mencionada anteriormente.

```
cargaLectura() {  
  this.routeLectura = this.rutaActiva.snapshot.params.lectura;  
  this.lecturaApiServ.getLectura(this.routeLectura)  
    .subscribe(lecturas => {  
    this.lectura = lecturas;  
    this.fileName = this.lectura.contenido + '.xlsx';  
  });  
}
```

La segunda función, carga los datos de cada una de las botellas que forman parte de esta lectura.

```
cargaBotella() {  
  this.botellaApiServ.getBotellasLectura(this.routeLectura)  
    .subscribe(botella => {  
    this.botellas = botella;  
    this.numBotellas = Object.values(botella).length;  
  });  
}
```

Esta clase está formada por algunas funciones más que forman la lógica de este componente.

- En los componentes restantes resto continuamos los mismos patrones, en cuanto a las estructuras de los archivos html y la lógica empleada para cada componente, según las necesidades

## ANÁLISIS

### Lista de requerimientos iniciales

La función principal de la aplicación consta en crear una lista con los códigos de los productos escaneados y una vez creada esta lista, exportarla en formato csv o excel, para poder ser usada con posterioridad.

Pero antes de empezar a escanear los productos, se deberá crear una lista con las características específicas del tipo de producto que será incluido en ella.

Esta lista se crea mediante un pequeño formulario, que consta del nombre del producto, la cantidad de unidades que tiene cada caja y por último la cantidad de cajas.

Una vez creada esta lista, pasaremos a la vista donde se realizan las lecturas, para poder realizar estas lecturas, es necesario una pistola de escaneo, que sea compatible con este tipo de códigos.

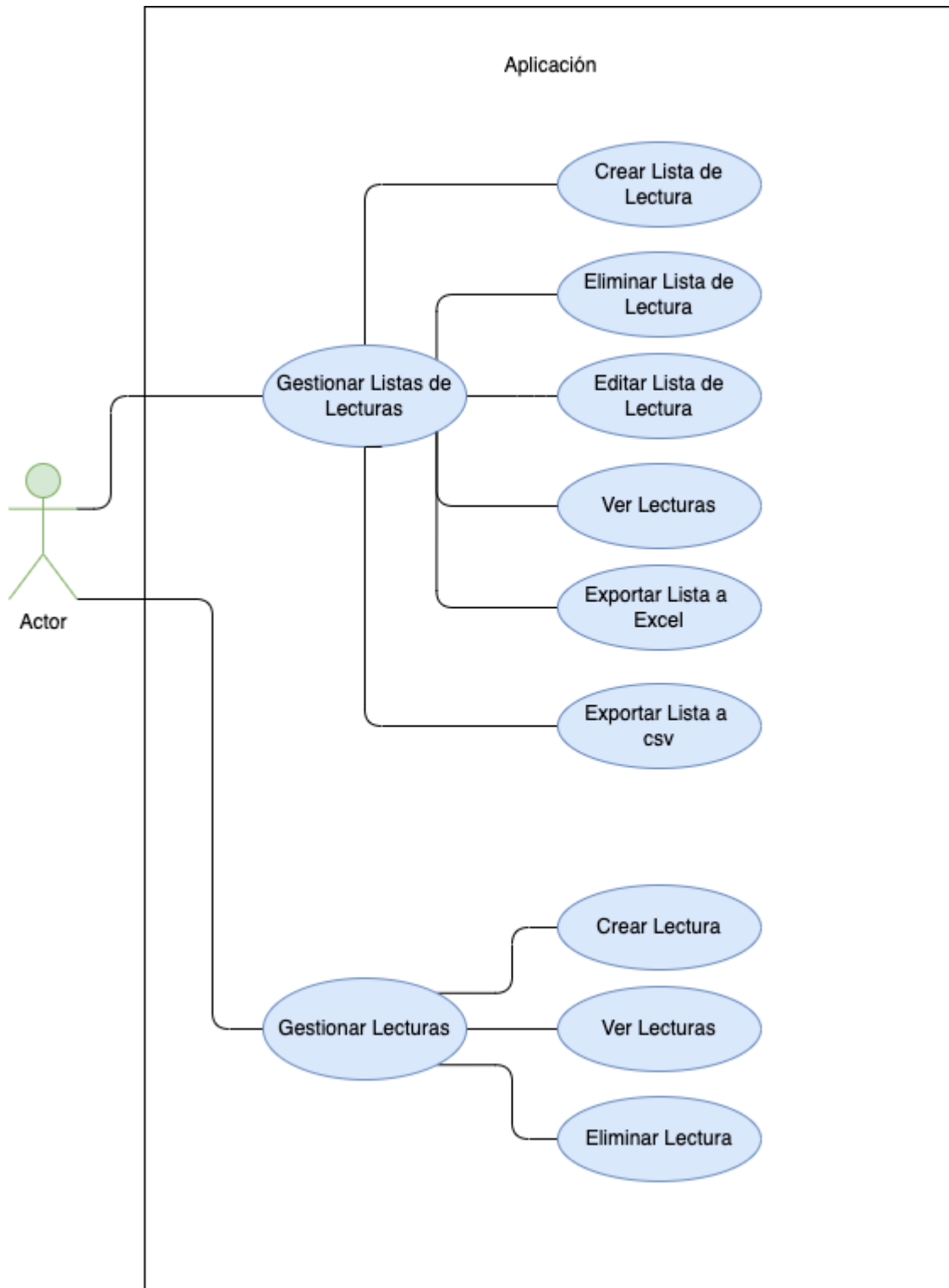
Tras escanear todos los códigos de los productos, se dispone de dos botones, con los que exportar esta información en los formatos comentados anteriormente.

### Actores del sistema

Por el momento, la aplicación comprende un único actor, este podrá realizar todos los casos de uso.

Una de las mejoras propuestas, añadiría la opción de usuarios registrados y no registrados, en el que todos los usuarios podrían realizar todos los casos de uso, pero solo de las lecturas que le correspondan.

## Diagrama de casos de uso



## Descripción de cada caso de uso

**Gestionar las listas de lecturas**, comprende **crear** nuevas listas de lecturas, esto se realizará mediante un formulario, una vez creada la lista, podremos, **consultarla**, **editarla**, mediante un formulario similar al de la creación y por último, podremos **eliminar** esta lista.

**Gestionar lecturas**, una vez hemos creado una lista con todos los atributos necesarios mediante el correspondiente formulario, accedemos a la vista donde se **mostrarán** todas las lecturas, la cual, se encontrará vacía y podremos crear una nueva lectura mediante el escáner QR, esta lectura también se podrá **eliminar**, pero no podremos editarla, ya que una de las cosas que se pretende automatizando este proceso, es minimizar al máximo el error humano.

Una vez hemos terminado de escanear todos los productos, podremos **exportar** cada lista en dos formatos distintos, según convenga. Esto creará un archivo en dicho formato y lo guardará automáticamente en la carpeta de descargas del navegador con el mismo nombre que le hemos dado a la lista.

## MANUAL DE USUARIO

Lo primero que veremos al acceder a nuestra aplicación, será la vista que muestra la lista de lecturas, al entrar por primera vez, como la base de datos no tiene almacenada ninguna lectura, muestra la lista vacía.

Lector QR

Lista de lecturas

Nueva Lectura

Lista de lecturas

#	Producto	Und/Caja	Total Cajas	Lecturas	Acción

En la parte superior de la pantalla, se muestra el menú, donde podemos ver un botón con el texto “Nueva Lectura”, si seleccionamos este botón, navegaremos hasta la siguiente vista.

Lector QR

Lista de lecturas

Nueva Lectura

## Nueva lectura

Nombre de producto

VODKA SMIRNOFF RED 37,5° 1L

Ejemplo: VODKA SMIRNOFF RED 37,5° 1L

Cantidad de unidades por caja

4

Introducir sólo números

Número de cajas

3

Introducir sólo números

Iniciar lectura

En esta vista podemos ver un pequeño formulario, con los datos necesarios para crear la nueva lectura.

En el primer campo, escribiremos el nombre del producto que va a ser escaneado, en el segundo campo, indicaremos la cantidad de unidades que hay en cada caja y por último, indicaremos el número total de cajas.

Una vez hemos completado todos los campos del formulario, seleccionamos el botón de “Iniciar lectura”. Esto nos llevará a la siguiente vista.

Lector QR [Lista de lecturas](#)

Nueva Lectura

VODKA SMIRNOFF RED 37,5° 1L - 3 cajas - 4 Unidades/caja

0 de 12

#	Nº botella	Precinto	Acción
---	------------	----------	--------

Introduce código

Añadir

Exportar excel

Exportar CSV



En esta vista se mostrará una lista con todos los productos escaneados. Al tratarse de una nueva lista de lecturas, esta se encuentra vacía.

En la parte superior izquierda, justo debajo del menú, se muestra el nombre del producto que se va a escanear junto con el resto de datos introducidos en el formulario del paso anterior, mientras que en la parte superior derecha se muestra un contador dinámico, con el número de productos escaneados y el total de productos, calculado, con el número de cajas y la cantidad de unidades que tiene cada caja.

Si continuamos analizando esta vista, un poco más abajo, nos encontramos con la cabecera de la tabla, que contiene la información que mostrará la tabla de cada producto escaneado.

Una vez llegamos al pie de la página, en la parte izquierda, nos encontramos con un campo de texto, el cual nos servirá para introducir el texto que se escanee. Como norma general, el usuario no debe introducir ningún dato en este campo, será el escáner el que se



encargará de este trabajo, pero siempre se podrá añadir información de manera manual, por si el código no se pudiera leer bien. Justo a la derecha de este campo de texto, se encuentra el botón de “Añadir”, igual que para el campo de texto, este botón no será necesario, salvo que se tengan que introducir los datos de forma manual, y servirá para que una vez tengamos los datos en el campo de texto, estos se añadan a la tabla.

A la derecha del botón “añadir” nos encontramos dos botones, estos nos servirán para exportar la tabla en los formatos que indica cada cada uno de los botones.

Y por último, dos botones con flechas, que nos enviarán al principio o al final de la página.

Vamos a comenzar escaneando el primer producto, para ello, situaremos el foco en el campo de texto y con el escáner, comenzaremos a leer los códigos de los productos.

Lector QR

Nueva Lectura

VODKA SMIRNOFF RED 37,5° 1L - 3 cajas - 4 Unidades/caja6 de 12

#	Nº botella	Precinto	Acción
1/12	1	20000000456	eliminar
2/12	2	20000000457	eliminar
3/12	3	20000000458	eliminar
4/12	4	20000000459	eliminar
5/12	1	20000000460	eliminar
6/12	2	20000000483	eliminar

Una vez comenzamos a escanear los productos, estos, van apareciendo en la tabla.

- **El primer campo** de la tabla muestra el número que ocupa el producto escaneado, frente al total de productos que hay que escanear.
- **El segundo campo** de la tabla, indica el número de productos escaneados por caja, cuando este número llega la cantidad máxima que contiene una caja, se pinta una línea que divide la tabla por cajas y el número de esta campo, reinicia su contador.

- **El tercer campo** de la tabla muestra el código de la etiqueta escaneada, este código tiene que ser único para cada lista de lectura, si por error, se vuelve a escanear el mismo producto, el programa mostrará una alerta con el texto que nos informa de que se ha repetido uno de los códigos.
- **El cuarto campo** de la tabla contiene un botón con el que se eliminará la fila correspondiente, al seleccionar este botón, se mostrará un alerta, con el número de código que se va a eliminar, esta acción se podrá cancelar.

Llegados a este punto, podríamos seleccionar el botón de exportar excel y nos descargará un archivo excel con el mismo nombre que el de la lista. (la función de exportar a CSV está en desarrollo)

En todo momento, desde el menú superior, se podrá volver a la vista que muestra todas las listas de lecturas, en caso de tener varias, se mostraría como en la imagen siguiente.

Lector QR

Lista de lecturas

Nueva Lectura

Lista de lecturas

#	Producto	Und/Caja	Total Cajas	Lecturas	Acción		
14	VODKA SMIRNOFF BLUE 50° 1 LITRO	12	30	360/360	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
15	GREY GOOSE VODKA 1 LITRO 40°	6	30	180/180	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
16	BALLANTINES 40° DE 1 LITRO	12	50	600/600	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
17	WHISKY J WALKER BLACK 40° 1L	12	15	180/180	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
18	VODKA ABSOLUT BLUE 40° 1L 108/200 CAJAS - LIDIA	12	108	1296/1296	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
23	JOHNNIE WALKER RED LABEL 1 L 40%	12	50	600/600	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
24	VODKA ABSOLUT 40% 1 L	12	45	540/540	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
26	SMIRNOFF 1L 37,5°	6	190	1140/1140	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>
28	VODKA SMIRNOFF RED 37,5° 1L	4	3	4/12	<a href="#">Abrir</a>	<a href="#">Editar</a>	<a href="#">Eliminar</a>

En esta vista, podemos ver una tabla que nos muestra algo de información sobre las listas de lectura, cabe destacar la columna con el título “Lecturas”, esta columna nos va a informar del número de lecturas que se han realizado, con respecto a la cantidad total de lecturas previstas, esto nos ayuda a saber qué lista está incompleta sin tener que entrar en cada una de las listas.

En este caso, la última columna esta incluye tres botones:

- **Abrir**, con este botón navegamos a la vista que contiene la tabla con las lecturas del producto, una vez en esta vista se podrán realizar todos los casos descritos anteriormente para esta vista.
- **Editar**, con este botón navegamos a la vista que muestra el mismo formulario que se utilizó para crear la lista de lectura, pero en esta ocasión, los campos están rellenos con los datos actuales. Se podrán modificar estos datos y guardar con los nuevos datos.

Lector QR

Lista de lecturas

Nueva Lectura

Editar lectura

Nombre de producto

VODKA SMIRNOFF RED 37,5° 1L

Ejemplo: VODKA SMIRNOFF RED 37,5° 1L

Cantidad de unidades por caja

4

Introducir sólo números

Número de cajas

3

Introducir sólo números

Guardar cambios

Cancelar

- **Eliminar**, con este botón se elimina una lista de lectura, lo que a su vez elimina todas las lecturas que tenga en su interior, este botón muestra una alerta, en la que tendremos que confirmar para eliminar la lista, ya que esta acción es irreversible.

## HERRAMIENTAS Y TECNOLOGÍAS EMPLEADAS

El principal objetivo de este proyecto es adquirir más conocimientos y consolidar los ya adquiridos sobre las tecnologías aplicadas durante la realización del mismo.



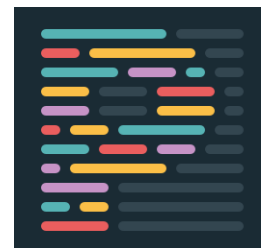
En primer lugar, para desarrollar este proyecto, me he servido del IDE Visual Studio Code. Este IDE ya lo venía usando durante el curso y me pareció muy completo, no sólo por lo que lo que el propio IDE ofrece, si no por la posibilidad de instalar una gran cantidad de plugins, que te hacen la vida mucho más fácil.

Algunos de los plugins utilizados y que se me han hecho imprescindibles para desarrollar son:



**TAG TAG**, cambia automáticamente el nombre de una etiqueta HTML/XML emparejada, ahorra mucho tiempo y dolores de cabeza.

Una de las cosas que más me ayuda cuando estoy desarrollando, es ver el código ordenado y bien indentado. Para ello he usado **PRETTIER**





Git ha sido una de las aplicaciones transversales del proyecto. Me ha servido para gestionar el control de versiones, además de utilizarlo como terminal por defecto durante todo el proyecto, con el cual he creado los proyectos e instalado dependencias entre otras cosas.

Las principales tecnologías utilizadas son **Angular**, **Laravel** y **Bootstrap**.



He elegido el framework de **Angular** para la realización del Frontend. Este framework desarrollado por Google y de código abierto, me ha permitido construir toda la estructura de la aplicación web SPA (Single Page Application).

TypeScript es el lenguaje con el que se desarrolla en Angular, aunque también se podría utilizar JavaScript.



Laravel, para la parte del Backend, se trata de uno de los frameworks de PHP más utilizados, a la vez que uno de los más sencillos de utilizar y en este punto ha tenido mucho que ver **Artisan**.

Artisan es un sistema de comandos de consola muy potente, que nos permite resumir muchas de las tareas más tediosas y repetitivas mientras desarrollamos nuestro proyecto.

# Bootstrap

Finalmente, he realizado todo el diseño de la aplicación usando Bootstrap, uno de los frameworks más conocidos, este framework está basado en componentes, con el que se pueden implementar temas de diseño completos y siendo igual de válido para pequeños proyectos como para proyectos mucho más complejos.

Una vez comprendes el funcionamiento de bootstrap, aporta mucha sencillez y agilidad al desarrollo de CSS.

## PROPUESTAS DE MEJORA

Una vez lanzada la primera versión de la aplicación y después de hacer varias pruebas con algunos códigos, comenzaron a utilizarla en el terreno, y esto generó una lluvia de propuestas de mejora.

Algunas eran tan simples como ampliar el tamaño de la fuente, ya que cuando se alejaban un poco para poder escanear los códigos, no lograban ver bien la pantalla del portátil y esto les hacía perder algo de tiempo.

En este sentido, incluí todas las mejoras que me solicitaron, pero aún quedan algunas que por temas concretos del puesto donde se realizan los trabajos no se pueden implementar.

Por ejemplo, me gustaría usar un servidor para todos los dispositivos y poder sacar todo a una misma base de datos, junto con esta mejora, agregaría usuarios, con el fin de poder pasar de un equipo a otro y poder continuar con las lecturas, ya que las lecturas se realizan con ordenadores portátiles y la duración de la batería podría llegar a ser un problema.

También tengo pendiente mejorar la velocidad con la que se comprueba la existencia de un código, ya que cuando hay más de 2.000 códigos en una misma lista, hay que hacer las

---

lecturas un poco más despacio, para darle tiempo a la aplicación a comprobar que el código no esté duplicado.

Otra de las mejoras, sería mejorar la experiencia de usuario (UI UX), de modo que fuese todo más intuitivo y fácil de utilizar.

Esta lista de mejoras nunca se terminaría, me gustaría parametrizar la aplicación para que se pueda utilizar con distintos tipos de códigos (actualmente sólo es compatible para los códigos de la agencia tributaria), pero como última idea, que no afecta al uso de la aplicación, pero si a la comprensión del código, me gustaría refactorizar nombres de variables, modelos, funciones y organizar un poco mejor el código en general.

## VALORACIÓN PERSONAL

### TRABAJO EN GRUPO Y METODOLOGÍA SCRUM

Debido a las circunstancias sanitarias actuales, el proyecto ha tenido que realizarse de forma individual, lo cual, le quita fuerza y algo de sentido a la metodología SCRUM.

Aun así, he querido adaptarlo a esta circunstancia, de modo que he creado un tablero de kanban utilizando la web de Trello, después de definir cuáles serían las tareas iniciales para crear el producto mínimo viable (MVP), este proceso es llamado Product Backlog.

Esto me ha ayudado a ser más realista con las tareas, en algunas ocasiones he tenido que desglosar alguna de las tareas en otras más pequeñas para evitar tener que invertir demasiado tiempo en ella y que se convirtiese en una tarea casi imposible de terminar.

También, de este modo, he podido ver de una forma más global, el proceso en sí. En casi todo momento era consciente de lo que me faltaba para terminar el proyecto, incluso, durante el desarrollo, cuando me encontraba alguna cosa que tenía que cambiar o no funcionaba del todo bien, lo apuntaba como correctivo, de esta forma, hasta que no terminaba la tarea en curso, no me ponía con otras cosas. Esta tarea de correctivos me la asignaba para otro momento.

---

## VALORACIÓN CRÍTICA DE LA METODOLOGÍA EMPLEADO Y MEJORAS QUE INCLUIRÍAS

Durante la realización de este proyecto, he aprendido mucho sobre las tecnologías con las que lo he desarrollado, pero también a tener en cuenta los aspectos más prácticos para empezar a hacer pruebas y dejar un poco más para el final los estilos de la aplicación, primero que funcione bien y luego el diseño.

Otro de los aspectos importantes que me ha aportado este proyecto ha sido el concepto de escalabilidad. Hay que tener en cuenta e incluso dejar preparado el código, para futuras mejoras, de forma que, por una pequeña modificación, no sea necesario tener que rediseñar una vista completa o incluso modificar las llamadas al backend y los modelos de los objetos. Cosa que me ha pasado en varias ocasiones durante el desarrollo del proyecto.

## PUNTOS A DESTACAR DE TU PROYECTO

Se trata de un proyecto que soluciona un problema real y el cuál actualmente se está utilizando y sigo manteniendo constantemente y realizando todas las mejoras que me solicitan.

Si bien es cierto que las tecnologías utilizadas para la realización de este proyecto, son las mismas que hemos visto en el módulo, no he podido adquirir muchos más conocimientos de desarrollo web durante mi formación en centro de trabajo (FCT) ya que la labor que he desempeñado desde que las comencé hasta la actualidad, ha sido desarrollando una aplicación nativa para móviles iOS (Apple).



## REFERENCIAS

Página oficial de Angular:

<https://angular.io/>

Página oficial de Laravel:

<https://laravel.com/>

Página oficial de Bootstrap:

<https://getbootstrap.com/>

Tutorial Laravel en YouTube del canal Coders Free:

<https://www.youtube.com/playlist?list=PLZ2ovOgdI-kWWS9aq8mfUDkJRfYib-SvF>

Tutorial en YouTube del canal Duilio Palacios:

<https://www.youtube.com/watch?v=ft9is7r6QoM&list=PL1r3w0C4CIYRbiTB4o70CyJEW6hUWJ39X&index=11>

Foro de programación:

<https://es.stackoverflow.com/>

Blog con tutoriales de desarrollo:

<https://desarrolloweb.com/>

Curso de Angular:

<https://escuela.it/cursos/curso-angular-8>

Curso de Bootstrap:

<https://escuela.it/cursos/taller-bootstrap>

Curso de Laravel:

<https://escuela.it/cursos/laravel>

Material Design para algunas imágenes:

<https://materialdesignicons.com/>