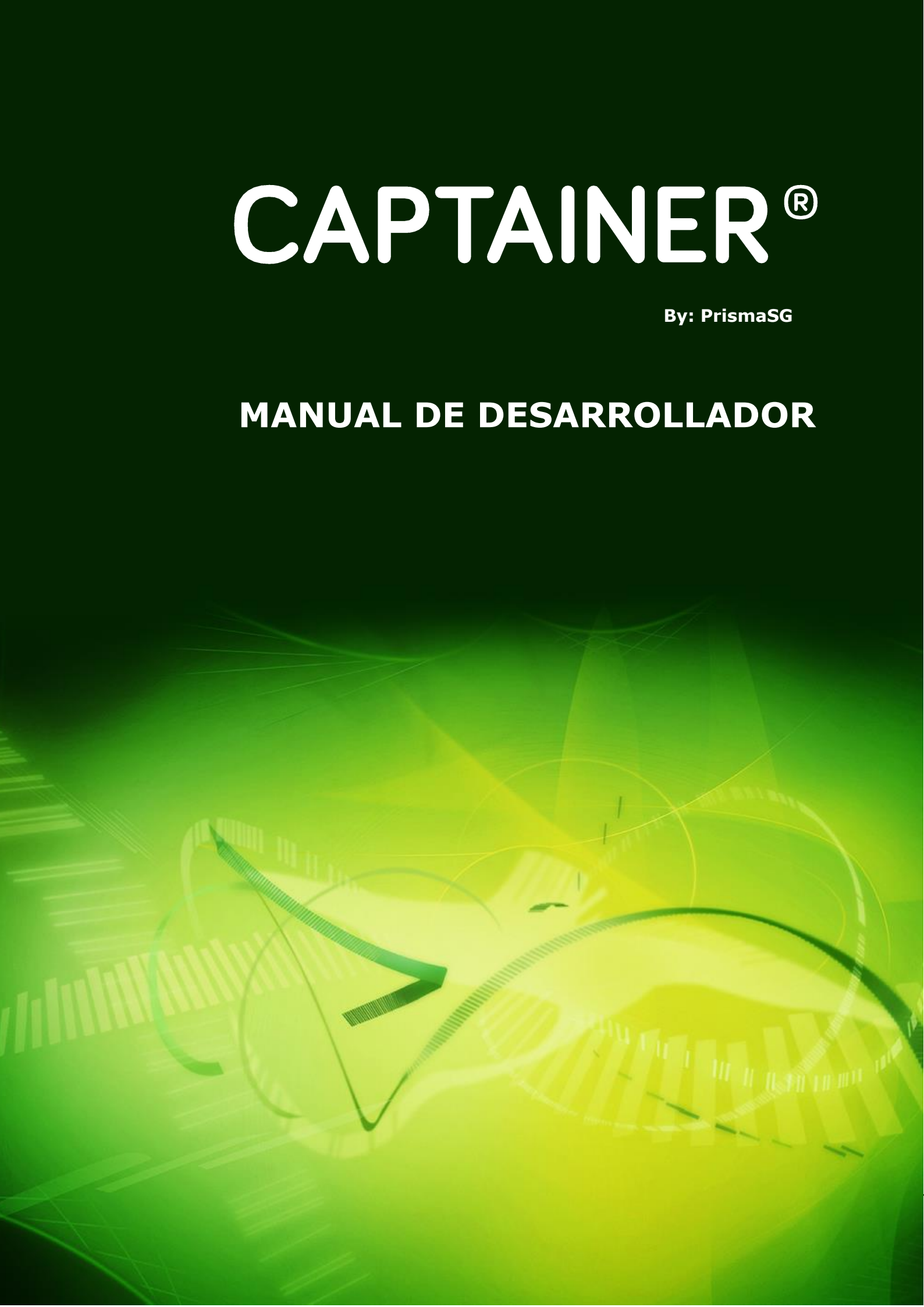
CAPTAINER®

By: PrismaSG

MANUAL DE DESARROLLADOR



Resumen:

Es importante conservar este documento en pos de futuros desarrollos de la aplicación. Dicha aplicación continuará su desarrollo e integración con otras herramientas de la empresa. Por lo que es importante conocer cuales son sus peculiaridades, así como la forma más adecuada de continuar su desarrollo.

A continuación, resaltaremos cuales son aquellos scripts y/o fragmentos de código alterables y cuales no. Del mismo modo, detallaremos cual ha sido el desarrollo hasta ahora y, en base a ello, como deberíamos operar si queremos reutilizar partes del desarrollo.

Los elementos utilizados para el desarrollo en Angular (Front-End) son los módulos, componentes, servicios, pipes y directivas; mientras que los elementos empleados en el Framework Lumen (Back-End) son los módulos, los controladores y la clase web.php, donde añadimos las rutas de nuevas consultas.

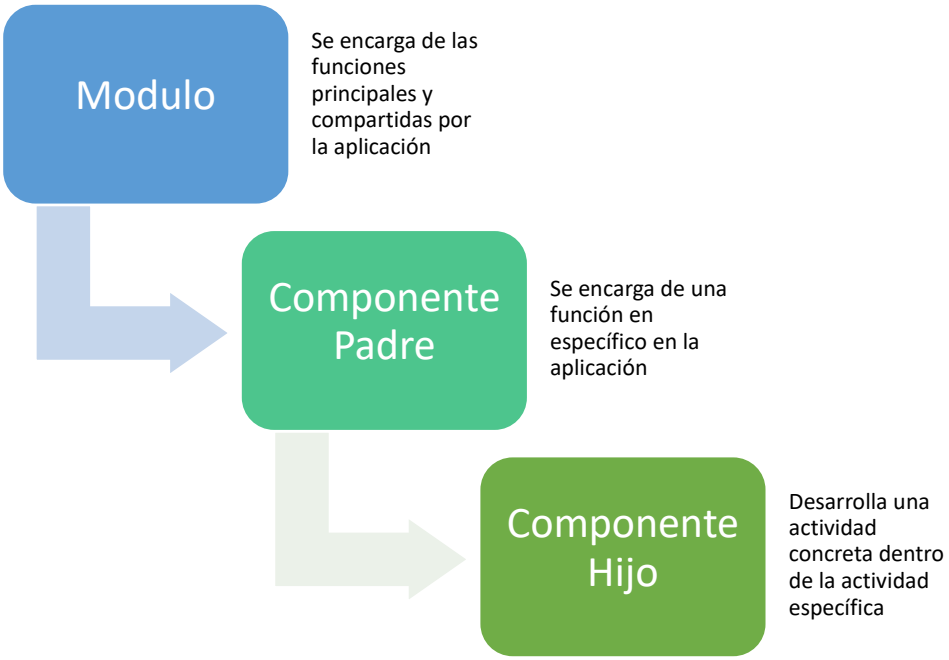
Será aconsejable preservar la estructura actual de la aplicación, ya que esta no ha sido elegida al azar, sino que se trata de una estructura escogida por su alta escalabilidad y es, por tanto, una garantía de crecimiento de la aplicación. Modificar la estructura de la aplicación como tal podría generar errores de dependencias y los efectos de los cambios desencadenarían diversos errores en el funcionamiento. Algunos indetectables, y por tanto muy peligrosos, y otros irreversibles, haciendo imposible retomar los pasos anteriores.

Sea consciente de la responsabilidad de continuar con el desarrollo de Captainer, y contribuya a este proyecto con buenas prácticas de programación. Y, ante todo, disfrute con el desarrollo:

Muchas gracias.
J.J. González – 'Desarrollador del proyecto'

1. Estructura de los componentes Angular:

Nuestra aplicación se estructura de la siguiente manera:



Por tanto, a la hora de desarrollar una nueva funcionalidad, debemos preguntarnos las siguientes cuestiones:

- A qué nivel de aplicación afecta la nueva función
- En qué parte o como queremos que se muestre en pantalla
- Que autoridad tiene, dentro de los permisos de la aplicación

En función de lo que hayamos contestado, debemos identificar en que categoría de las anteriormente mencionadas se encuentra nuestra nueva función. Vemos como desarrollamos una **vista detalles** de un **modelo** de nuestra aplicación:

Lo primero que debemos hacer es definir la jerarquía. En este caso, la vista detalles es una función específica, pero independiente de otras funciones específicas ya existentes. Por tanto, se trata de un componente, a secas. Para generar un componente en nuestra aplicación Angular, empleamos el siguiente comando, ejecutándolo en la raíz del proyecto:

```
ng generate component nombre_del_componente
```

Se creará una carpeta con el nombre del componente con la siguiente estructura:

En el cual encontraremos los ficheros necesarios para la confección de nuestra aplicación:

- Plantilla HTML para la interfaz
- CSS para el estilo de la plantilla
- Typescript para la lógica del componente

El cuarto elemento*, spec.ts,* no será utilizado, ya que no forma parte de los contenidos editables del componente.

Es importante que hagamos este procedimiento de esta forma, y no de forma manual, ya que de esta manera nos ahorraremos tener que importar el componente y la clase Typescript en nuestro fichero *app.module.ts.*

Nuestro fichero app.module tiene una estructura bien definida, y no conviene alterarla. Esta es:

```
import { AppComponent } from './app.component';
import { CommonModule } from '@angular/common';
import { RequestService } from './servicios/request.service';

@NgModule({
  declarations: [
    AppComponent,
],
imports: [
    CommonModule,
],
providers: [RequestService],
bootstrap: [AppComponent]
})

export class AppModule { }
```

Una vez creado nuestro componente, debemos realizar un **desarrollo escalonado** y procurando siempre verificar que el desarrollo **no cause errores** en el sistema. Yo recomiendo desarrollar el componente con el **servidor** virtual abierto. Esto se hace ejecutando el comando:

```
ng serve
```

De esta forma, además de desarrollar, podremos ir haciendo sobre la marcha **pruebas de integración ascendente** con un navegador web, escribiendo en él la **dirección**: *localhost:4200***.**

Para evitar problemas de inconsistencia de información. Primero recomiendo desarrollar los elementos de la plantilla HTML, y posteriormente aplicar el estilo con CSS. Una vez tengamos nuestra interfaz estructurada y comprobemos la usabilidad y visibilidad de todos sus elementos. Comenzamos con el diseño. Nuevamente, recomiendo no entrar directamente con todas las características de Angular. Primero, vamos a desarrollar los métodos de la clase Typescript, y comprobar que interactúan con las directivas del DOM de la plantilla HTML, empleando *logs* en la consola del navegador. Esto lo conseguimos mediante los **`console.log()`**. Ahora que ya tenemos los métodos desarrollados, deben tener la siguiente forma:

Plantilla HTML

```
<table class="table table-striped table-responsive table-hover">

    <tr (click)= “pulsame()”>
      <th>
        Nombre de la implantación:
      </th>
      <td>
      </td>
    </tr>
</table>
```

Typescript

```
pulsame() {
        console.log("Has pulsado una fila de la tabla")
```

Ya tenemos la tabla preparada para los detalles de una implantación, pero no contamos con los datos necesarios para alimentarla. Para ello, haremos una petición a nuestro servicio HTTP en Angular, *requesting.service.ts*, encargado de la comunicación con el Back-End.

2. Comunicación entre Front-End y Back-End

Esta comunicación la realizaremos mediante una API RESTFul alojada en nuestra aplicación. Lo primero es realizar el desarrollo del servicio Angular de comunicación HTTP. Pero, para ello, primero debemos realizar las consultas a la base de datos y conocer las rutas de la API.

Lo primero es crear un **Controlador** en nuestro Back-End, encargado de precargar un **Modelo** de una tabla de la base de datos, y trabajar con él. La facilidad de trabajar con *Lumen* es que contamos con todas las bondades de *Laravel* y *Symphony.* Esto quiere decir que hacer una consulta a la base de datos nunca fue tan fácil. Veamos por qué:

DeployController.php

```
use App\Deploy;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class DeployController extends BaseCrudController {
  public function __construct() {
    parent::__construct(Deploy::class);
  }

  /* GET */

  public function listDeploy()
  {
    $deployList = DB::table('deploy')
      ->select('deploy.*')
      ->get();
    return $deployList;
  }


  public function listDeployById($id)
  {
    $deployList = DB::table('deploy')
      ->select('deploy.*')
      ->where('deploy.id', '=', $id)
      ->get();
    return $deployList;
  }
}
```

Contamos con **dos peticiones GET:** Con la primera de ellas nos traemos toda la información acerca de las implantaciones. Con la segunda de ellas, consultamos una implantación concreta mediante su *ID.* Como vemos, trabajar con Lumen es **muy sencillo.**

Ahora, definimos sus rutas en el fichero ***web.php:***

Web.php

```
$app->group(['prefix' => 'deploy'], function ($app) {
    $app->get('{id}', 'DeployController@listDeployById');
    $app->get('', 'DeployController@listDeploy');
    $app->post('', 'DeployController@createDeploy');
    $app->put('{id}', 'DeployController@fileDeploy');
    $app->delete('{id}', 'DeployController@destroy');
  });
```

Ahora, si queremos hacer la petición, solo tendremos que realizar la siguiente consulta mediante las URL:

- ✓ Dirección_del_servidor/api_lumen/public/api/deploy → Todas
- ✓ Dirección_del_servidor/api_lumen/public/api/deploy/:id → Por ID

Una vez probado el servicio REST, el siguiente paso es desarrollar el servicio Angular:

Este servirá para **todas las peticiones CRUD de la aplicación**, por lo que es interesante hacer un servicio, puesto que nos ahorramos muchas líneas de código y bastante tiempo de desarrollo:

Requesting.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { AuthenticationService } from './autenticacion.service'
import { Issue } from '../modelos/Issue';
import { Observable } from 'rxjs';

@Injectable()
export class RequestService {


  public url = "dirección_del_servidor/api_lumen/public/";

  constructor(
    private http: HttpClient,
    private auth: AuthenticationService
  ) { }
```

```
  /** PETICIONES CRUD */

  //Metodos GET

  getAll(url: string): Observable<any> {
    let tok = this.auth.currentUserValue.token;
    let header = new HttpHeaders();
    header = header.set('Authorization', 'Bearer ' + tok).set('Content-
Type', 'application/json; charset=utf-8');
    return this.http.get(url, { headers: header });
  }

  get(url: string): Observable<any> {
    let tok = this.auth.currentUserValue.token;
    let header = new HttpHeaders();
    header = header.set('Authorization', 'Bearer ' + tok).set('Content-
Type', 'application/json; charset=utf-8');
    return this.http.get(url, { headers: header });
  }

  // Metodo POST

  post(url: string, item: any): Observable<any> {
    let tok = this.auth.currentUserValue.token;
    let header = new HttpHeaders();

    header = header.set('Authorization', 'Bearer ' + tok).set('Content-
Type', 'application/json; charset=utf-8');
    return this.http.post(url, item, { headers: header });
    //item:Issue --> anterior
  }


  // Metodo PUT

  put(url: string, item: Issue): Observable<any> {
    let tok = this.auth.currentUserValue.token;
    let header = new HttpHeaders();
    header = header.set('Authorization', 'Bearer ' + tok).set('Content-
Type', 'application/json; charset=utf-8');
    return this.http.put(url, item, { headers: header });
  }

  // Metodo DELETE

  del(url: string): Observable<any> {
    let tok = this.auth.currentUserValue.token;
    let header = new HttpHeaders();
```

```
    console.log(tok);
    header = header.set('Authorization', 'Bearer ' + tok).set('Content-
Type', 'application/json; charset=utf-8');
    return this.http.put(url, null, { headers: header });
  }

  // Metodo DELETE

  destroy(url: string): Observable<any> {
    let tok = this.auth.currentUserValue.token;
    let header = new HttpHeaders();
    header = header.set('Authorization', 'Bearer ' + tok).set('Content-
Type', 'application/json; charset=utf-8');
    return this.http.delete(url, { headers: header });
  }
}
```

Donde:

- url: es una **dirección que comparten todas las peticiones**, y hace referencia al prefijo de la dirección absoluta. En nuestro caso es *dirección_del_servidor/api_lumen/api*, ya que es una referencia que no cambia. La usamos como atajo.
- Tok: Es el **token** usado en la validación y recogido en el *Local Storage.*

Con estos cuatro métodos, que corresponden a los verbos HTTP, podremos hacer todas las peticiones AJAX de la aplicación.

Finalmente, realizamos los **métodos observables** en la clase *Typescript,* que realicen las ***promesas*** y rellenen los *arrays* responsables de inyectar en la tabla los valores mostrados mediante una función de *Callback* a través del *JSON* del Back-End:

Typescript:

```
deploys: Deploy[] = [];

deploy() {
    this.requesting.get(this.requesting.url + 'api/deploy).subscribe(
      response => {
        this.deploys = response;
        console.log(response)
        this.loading = false;
      },
      error => {
        console.log ("Se ha producido un error ")
        console.log(error);
      }
    )
  }
```

3. Binding en el DOM de la plantilla HTML

Como ya tenemos la lógica desarrollada tanto en el Back-End como en el Front-End, solo tendremos que diseñar la plantilla HTML para que muestre la información. En nuestra aplicación empleamos los ***Bindings***. Estas directivas **permiten mostrar variables pertenecientes al Typescript** del componente. Veamos como:

Plantilla HTML

```
<table class="table *ngFor="let deploy of deploys">

    <tr >
      <th colspan="2" style="text-align:center">
        DETALLES DE IMPLANTACION: {{deploy.id}}
      </th>
    </tr>
    <tr>
      <th>
        Nombre de la implantación:
      </th>
      <td>
        {{deploy.name}}
      </td>
    </tr>
    <tr>
      <th>
        Estado de soporte:
      </th>
      <td>
        {{deploy.support_status}}
      </td>
    </tr>
    <tr>
      <th>
        Licencia
      </th>
      <td>
        {{deploy.license}}
      </td>
    </tr>
  </table>
```

Al emplear la **directiva *ngFor,*** podremos crear un bucle For-Each del array *deploys[]* que tenemos en nuestro Typescript. **Se mostrarán las propiedades del objeto del array** que corresponda a cada ciclo del bucle que lo recorre.

Todo este procedimiento nos permite hacer una consulta de información, y mostrar los datos de la siguiente manera:

LISTA DE IMPLANTACIONES

Nombre	Status de soporte	ID de cliente	ID de proyecto	Licencia	Status
San Isidro	1	1	1	L-540uAAAAAAAAA	1
Piantini	0	1	1	L-562dAAAAAAAAA	1
Ozama	1	1	1	L-5839AAAAAAAAA	1
Carrefour Huelva	1	4	2	L-582dAAAAAAAAA	1
Salvense 1	0	5	2	FHGE GHTJ WKDI IHO1	1
Santurtzi	1	6	1	045-403-238490283-434	1
Integracion Santisima trinidad	1	8	1	L-5839AAAAAAAAB	1
Salvensen 2	1	5	2	213h12i3uh4801232984hoih	1

4. Insertar y modificar registros

Si, por el contrario, necesitamos crear formularios en nuestra aplicación, bien para **insertar** información, o bien para **editar** la información existente, necesitamos seguir el siguiente orden:

En primer lugar, lo que haremos será **implementar el componente**, haciendo uso de las herramientas que ya hemos visto en los puntos anteriores. Posteriormente, debemos **diseñar el formulario en nuestra plantilla HTML** del componente, y otorgarle estilos desde nuestro CSS.

Los formularios recomendados para el correcto uso de la aplicación son los siguientes:

Plantilla HTML - Formulario

```
<form #myform>
    <div class="row">
      <div class="col-25">
        <label for="name">Nombre de la implantacion</label>
      </div>
      <div class="col-75">
        <input #nameBox type="text" id="name" name="name" placehol-
der="Introduce nombre de la implantacion"
          (keyup)="name=nameBox.value" (keyup.enter)="anadir(myform)">
      </div>
    </div>
```

```
    <div class="row">
      <div class="col-25">
        <label for="support_status">Status de soporte</label>
      </div>
      <div class="col-75">
        <input #supportBox type="number" id="support_status" name="sup-
port_status"
          (keyup)="support_status=supportBox.value" (keyup.enter)="ana-
dir(myform)">
      </div>
    </div>

    <div class="row">
      <div class="col-25">
        <label for="client_id">Id Cliente</label>
      </div>
      <div class="col-75">
          <select name="client_id" form="client_id" [(ngModel)]="selec-
tedClient">
              <option *ngFor="let client of clients" va-
lue="{{client.id}}">{{client.company_name}}</option>
            </select>
      </div>
    </div>

    <div class="row">
      <button class="btn btn-primary form-btn" (click)="ana-
dir(myform)">
        <i class='fas fa-save'></i> Añadir
      </button>
      <button class="btn btn-danger form-btn" (click)="clear(myform)">
        <i class='fas fa-eraser'></i> Limpiar
      </button>
    </div>
  </form>
```

Como vemos, se hacen uso de funciones *'click'* o *'keyup'*, directivas *ngFor* y nombres identificativos asignados a distintas etiquetas del *DOM* (#myform)

Ahora que ya tenemos definida la comunicación desde la plantilla, **debemos diseñar el método 'anadir()'**, que será el responsable de **capturar el objeto** generado a partir del formulario y realizar la *suscripción* al **servicio Angular** para que este realice la *promesa* y devuelva el resultado en la función de *Callback*.

Typescript

```
  name: string;
  support_status: number;
  client_id: number;
  project_id: number;
  license: string;
  status: number;
  selectedClient: string;
  selectedProject: string;
  selectedStatus: number;


anadir() {

  let uri = this.requesting.url+'api/deploy';

let deploy = new Deploy(this.name, this.support_status, this.selec-
tedClient, this.selectedProject, this.license, this.selectedStatus);

     this.requesting.post(uri, deploy).subscribe(
       response => {
         console.log("datos introducidos correctamente")
         console.log(response);
       },
       error => {
         this.fail = true;
         console.log("Se ha producido un error ")
         console.log(error);
       }
     )
  }
```

Recordemos que *this.requesting* es una instancia de **nuestro servicio Angular** *request.service.ts*, y que *post* es un método de dicho servicio que realiza precisamente la petición POST. Desde el DOM, igualaremos los valores de las propiedades del objeto *Deploy* por aquellas entradas recibidas por los *inputs* de la plantilla.