PMDM03: PROGRAMACIÓN DE APLICACIONES PARA DISPOSITIVOS MÓVILES (II)

1-DESACOPLAR LÓGICA DE LOS CONTROLADORES.

Tal y como ya hemos mencionado anteriormente, los controladores de las vistas no deben contener código que no esté relacionado con la vista y la captura de los eventos. Por lo tanto, en el controlador de la vista no debe haber código que gestione el acceso a datos que estén en una base de datos local, ni una petición http ni la carga de datos inicial. Aunque, pueda parecer que el constructor de la vista inicial puede ser una opción para inicializar los valores iniciales, no es así, ya que seguramente la inicialización se realice cargando datos desde una base de datos o desde una petición http.

Por lo tanto, en el controlador de la vista únicamente tendremos los métodos que se encarguen de capturas los eventos y el constructor. En el constructor se puede gestionar el paso de parámetros, pero una vez que estos han sido obtenidos se delegará cualquier trabajo que deba realizarse con estos que no sea una asignación a una variable o atributo. Por lo tanto, para gestionar el acceso a Storage (almacenamiento local) y para gestionar peticiones HTTP a un servidor REST utilizaremos como mínimo un proveedor.

2-FORMATO JSON.

JSON será el formato que utilicemos tanto para almacenar datos, como para las peticiones HTTP. Por lo tanto, vamos a ver algunos ejemplos para entender cómo funciona:

- Array de cadenas: ["uno","dos","tres"]
- Objeto: {"descripcion":"uno","esImportante":false,"finalazida":false}
- Array de objetos: [{"descripcion":"uno","esImportante":false,"finalazida":false}, {"descripcion":"dos","esImportante":false,"finalazida":false}]

Como se puede observar en los ejemplos, los atributos de los objetos se convierten en pares clave-valor. La clave corresponde al nombre del atributo y el valor al valor que toma en la instancia que se desea convertir a formato JSON. Si el elemento que se desea convertir a formato JSON es un array, los diferentes elementos se separan con comas y se introducen entre corchetes.

La pregunta que debe surgirnos ahora es cómo convertir los datos a formato JSON y cómo instanciar o inicializar nuestro modelo. Para la primera pregunta no será necesario muchas explicaciones, ya que no es necesario realizar ninguna acción, es decir, el proceso es "transparente". Para la segunda en cambio necesitaremos añadir a las clases de nuestro modelo un método estático que se encargue de instanciar objetos a partir de los datos en formato JSON.

```
//el tipo de datos del parametro de entrada es any
    static fromJson(data:any){
    //se comprueba que todos los campos necesarios para instanciar un TareaModel estan
        if(!data.descripcion|| ! data.esImportante || ! data.finalizada){
            //si falta alguno se lanza una exception
            throw(new Error("Invalid argument: argument structure do not match with model fields"));
        }
        return new TareaModel(data.description, data.listId, data.isImportant);
    }
```

Otro aspecto que se debe tener en cuenta es que si hemos almacenado un array de objetos o cadenas, necesitaremos una estructura repetitiva para instanciar los n elementos del array.

3-STORAGE.

Ionic Capacitor ofrece el paquete Storage para poder almacenar datos de forma persistente. Es decir, nos va a permitir almacenar datos mientras el usuario realiza acciones en la aplicación y cuando el usuario vuelva a abrir la aplicación los datos introducidos por el usuario van a poder ser recuperados y cargados en la aplicación. Storage provee un almacenamiento clave/valor.

Storage utiliza la ubicaciones diferentes de almacenamiento basada en la plataforma en la que se está utilizando. Cuando se ejecuta la aplicación en iOS este plugin usará UserDefaults y en Android SharedPreferences. Los datos almacenados se borran si la aplicación se desinstala. Los sistemas operativos móviles pueden borrar periódicamente los datos establecidos en window.localStorage, por lo que esta

API debería utilizarse en lugar de window.localStorage. Esta API volverá a utilizar localStorage cuando se ejecute como una aplicación web progresiva.

Si al crear el proyecto hemos indicado que queremos integrar Capactior no será necesario realizar ninguna instalación. En caso contrario tendremos que añadir Capacitor a nuestro proyecto

```
$npm install –save @capacitor/core @capacitor/cli
```

Para la facilitar el uso de Storage se recomienda utilizar el siguiente servicio e inyectarlo por dependencias cuando se quiera hacer:

```
import { Injectable } from "@angular/core";
import { Plugins } from "@capacitor/core";

const { Storage } = Plugins;
@Injectable({
  providedIn: "root",
})
export class StorageServiceService {
  constructor() {}

  async setString(key: string, value: string) {
    await Storage.set({ key, value });
  }

  async getString(key: string): Promise<{ value: any }> {
    return await Storage.get({ key });
  }

  async setObject(key: string, value: any) {
    await Storage.set({ key, value: JSON.stringify(value) });
  }

  async getObject(key: string): Promise<{ value: any }> {
    const ret = await Storage.get({ key });
    return JSON.parse(ret.value);
  }
```

```
  async removeItem(key: string) {
    await Storage.remove({ key });
  }

  async clear() {
    await Storage.clear();
  }
}
```

En el siguiente código se puede ver el uso del servicio en los métodos addPersona y eliminarPersona:

```
import { Injectable } from '@angular/core';
import {Persona} from '../modelo/persona';
import { StorageServiceService } from './storage-service.service';
@Injectable({
  providedIn: 'root'
})
export class ServicioPersonasService {


  public personas:Persona[]=[];
  constructor(private servicioStorage:StorageServiceService) {

  }
  public addPersona(item:Persona){
    this.personas = [...this.personas,item];
    this.servicioStorage.setObject("personas",this.personas);

  }
  public getPersona(id):Persona{
```

```
    return this.personas.find(persona=>persona.id==id);

  }

  public eliminarPersona(item: Persona) {

    let indice = this.personas.indexOf(item);

    this.personas = [...this.personas.slice(0,indice),...this.personas.slice(indice+1)];

    this.servicioStorage.setObject("personas",this.personas);

  }


}
```

4-SERVIDOR REST.

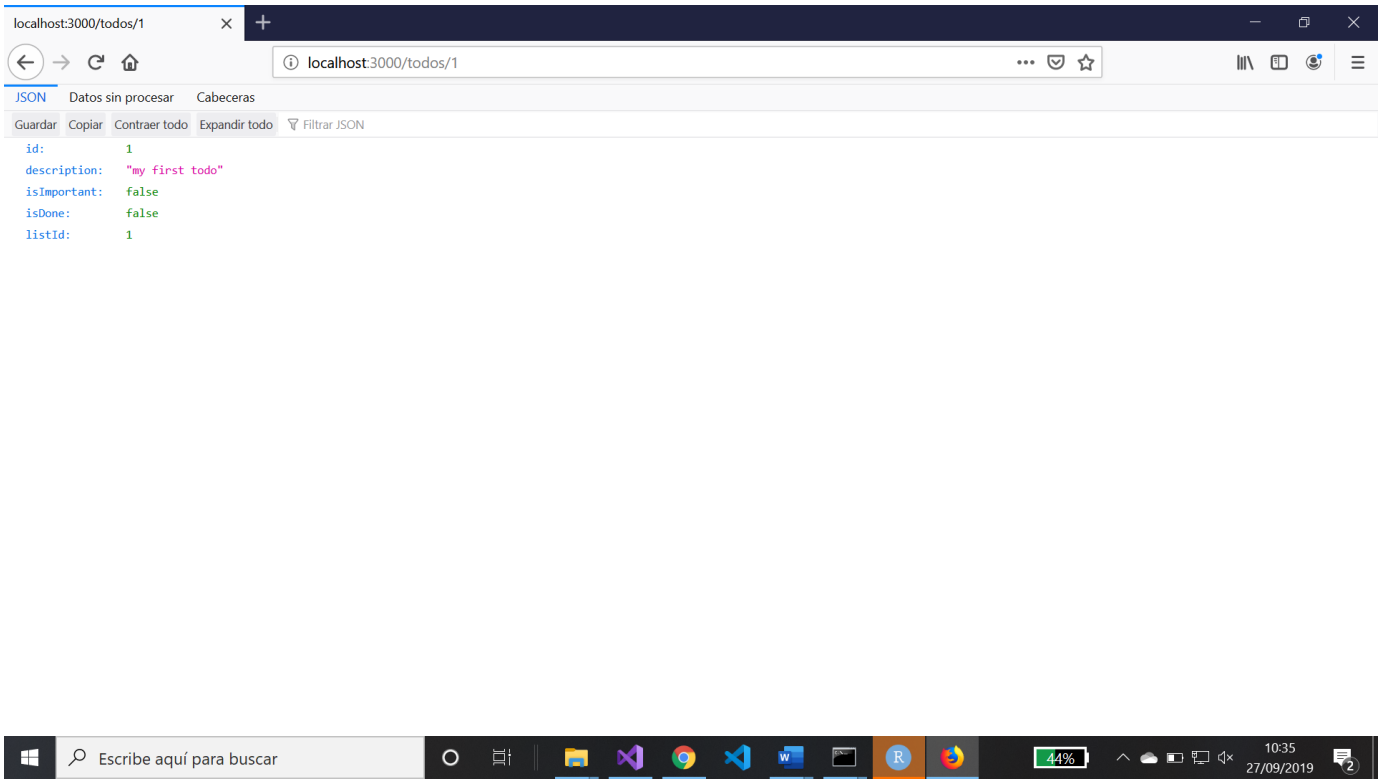
Antes de comenzar con las peticiones http, vamos a ver cuáles son las operaciones que ofrece un API REST.

- GET: con esta operación consultaremos los recursos
- POST: con esta operación se crear registros
- PUT: con esta operación se actualizan registros
- DELETE: con esta operación se borrar registros

Como cualquier otro servidor, el servidor debe tener un url, que será la que utilizaremos para acceder a las operaciones. Una vez que el servidor está funcionando indica los recursos a los que tenemos acceso y su ruta.

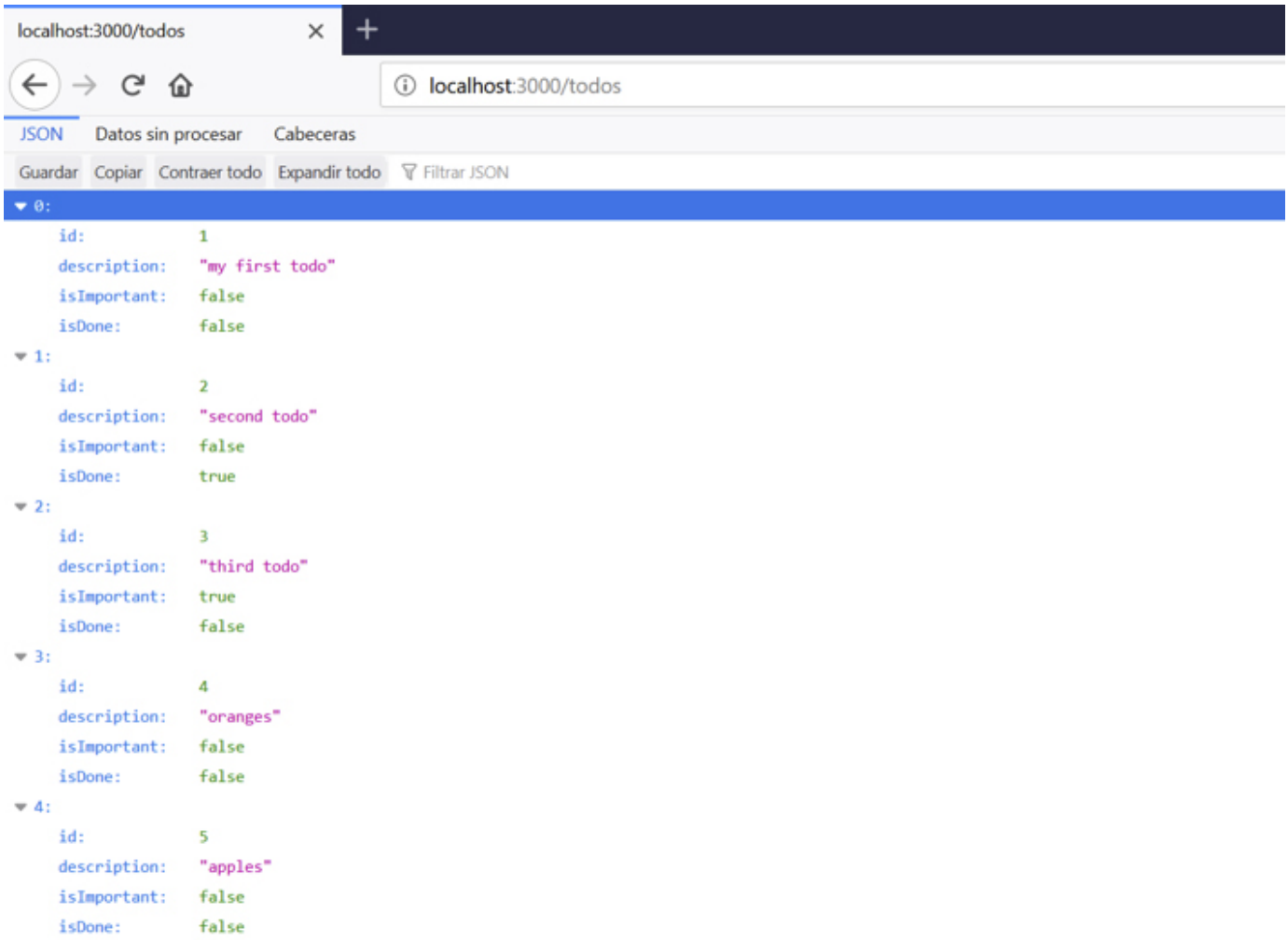
```
Loading db.json
Done

Resources
http://localhost:3000/todos

Home
http://localhost:3000
```

A continuación, se muestran unas capturas para entender mejor como funciona se accede a los recursos:



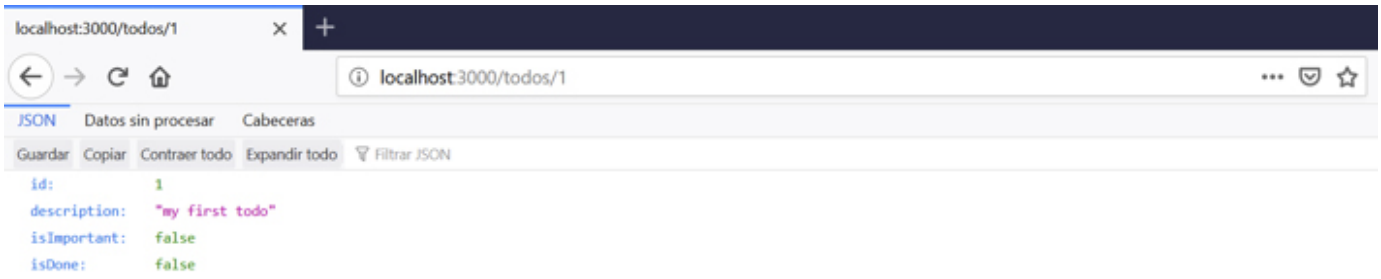
BIRT LH (Copyright (cita)) *(link: https://www.google.es)*

Obra derivada de captura de pantalla del programa Mozilla Firefox, propiedad de Mozilla Foundation



BIRT LH (Copyright (cita)) *(link: https://www.google.es)*

Obra derivada de captura de pantalla del programa Mozilla Firefox, propiedad de Mozilla Foundation



BIRT LH (Copyright (cita)) *(link: https://www.google.es)*

Obra derivada de captura de pantalla del programa Mozilla Firefox, propiedad de Mozilla Foundation

5-PETICIONES HTTP.

Para realizar las diferentes peticiones http, vamos a utilizar los interfaces que nos ofrece HTTPClient. Todos los métodos retornan un Observable. En el siguiente punto veremos que son y gestionar las peticiones.

A continuación se incluye una clase que incluye la funcionalidad para realizar las llamadas haciendo uso de HttpClient:

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

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

import { Observable, throwError } from "rxjs";

import { retry, catchError } from "rxjs/operators";

// Se importan las clases del modelo necesarias para utilizar en la llamadas <T>

// Incluir la clase no asegura ningun tipo de comprobacion. Si se quiere realizar

// comprobacion de los campos de debe realizar

import { Persona } from "../modelo/persona";


@Injectable({

  providedIn: "root",

})

export class HttpServiceService {

  // url base. Se puede incluir en un fichero de configuración

  base_path = "http://localhost:3000/person";


  constructor(private http: HttpClient) {}


  httpOptions = {

    headers: new HttpHeaders({
```

```
    "Content-Type": "application/json",
  }),
};

// Handle API errors
handleError(error: HttpErrorResponse) {
  if (error.error instanceof ErrorEvent) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error("An error occurred:", error.error.message);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong,
    console.error(
      `Backend returned code ${error.status}, ` + `body was: ${error.error}`
    );
  }
  // return an observable with a user-facing error message
  return throwError("Something bad happened; please try again later.");
}

createItem(item): Observable<Persona> {
  console.log(item);
  return this.http
    .post<Persona>(this.base_path, JSON.stringify(item), this.httpOptions)
    .pipe(retry(2), catchError(this.handleError));
}

getItem(id): Observable<Persona> {
  return this.http
    .get<Persona>(this.base_path + "/" + id)
    .pipe(retry(2), catchError(this.handleError));
}
```

```
  getList(): Observable<Persona[]> {
    return this.http
      .get<Persona[]>(this.base_path)
      .pipe(retry(2), catchError(this.handleError));
  }

  updateItem(id, item): Observable<Persona> {
    return this.http
      .put<Persona>(
        this.base_path + "/" + id,
        JSON.stringify(item),
        this.httpOptions
      )
      .pipe(retry(2), catchError(this.handleError));
  }

  deleteItem(id) {
    return this.http
      .delete<Persona>(this.base_path + "/" + id, this.httpOptions)
      .pipe(retry(2), catchError(this.handleError));
  }
}
```

Es necesario importar HttpClientModule en el fichero app.module.ts. Si no lo hacemos la app lanzará un error:

```
import { BrowserModule } from '@angular/platform-browser';
import { RouteReuseStrategy } from '@angular/router';

import { IonicModule, IonicRouteStrategy } from '@ionic/angular';
import { SplashScreen } from '@ionic-native/splash-screen/ngx';
import { StatusBar } from '@ionic-native/status-bar/ngx';
```

```
import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(), AppRoutingModule,HttpClientModule],
  providers: [
    StatusBar,
    SplashScreen,
    { provide: RouteReuseStrategy, useClass: IonicRouteStrategy }
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Por último, es recomendable tener un fichero de configuración en el que se incluya la url del servidor para realizar los cambios en un único lugar. El fichero sería el siguiente:

```
export class AppSettings{
    public static get API_ENDPOINT():string{return 'http://localhost:3000'}
}
```

Para utilizarlo en nuestras peticiones tendríamos que importar la clase y a continuación definir el string que utilizamos con la url con las siguientes comillas ` ` y haciendo uso del operador ${}, tal y como se muestra a continuación:

```
http.delete(`${AppSettings.API_ENDPOINT}/todos/${id}`)
```

6-OBSERVABLE.

Tal y como hemos visto anteriormente los diferentes métodos de HTTPClient retorna un tipo de datos Observable con los cuales vamos a tener que trabajar. Lo primero que vamos a definir son dos conceptos:

Observable: representa la idea de una colección invocable de valores o eventos futuros. Hasta que no nos suscribimos al observable, no se ejecuta la petición.

Subscription: representa la ejecución de un Observable, en el momento en el que nos suscribimos a un Observable se realizar la petición sobre el servidor REST en nuestro caso.

Como a un Observable se pueden suscribir n “observers” se debe tener cuidado, ya que si no se comparte el Observable se realizarán n peticiones. Esto puede ocasionar que en vez de un POST se realicen n POSTs. Por ello, cuando vayamos a compartir el Observable

utilizaremos share. Si no compartimos el Observable no hace falta incluir la llamada al método share.

En nuestro caso ya solo nos falta convertir los datos recibidos a una instancia o instancias de nuestro modelo. Si no lo hacemos, no será posbile comprobar si la respuesta del servidor incluye todos los campos necesario para instanciar los objetos del modelo. Para ello haciendo uso de la función estática de cada una de las clases se convierten los datos a instancias o a una instancia en función de la respuesta del servidor.

```
.map((persona) => Persona.fromJson(persona))
```

Con todo el código vista hasta ahora, todavía no se ejecutaría la petición ya que no nos hemos suscrito todavía. A la hora de realizar la subcripción, el método subscribe toma como parámetros de entrada dos funciones de callback. La primera función es la que se ejecuta cuando no ha habido ningún problema. En nuestro caso, la función tiene un parámetro de entrada de tipo TodoModel. La segunda función, en cambio, toma como parámetro de entrada el error que se ha producido y lo muestra en la consola.

```
this.servicioHttp.getList().subscribe(
      (datos) => {
        datos.map((persona) => Persona.fromJson(persona));
      },
      (error) => console.log(error)
    );
```

Existe la posibilidad de hacer return de un Observable, para que la suscripción se realice en diferentes puntos del código. En ese caso, habría que eliminar la parte que corresponde a la suscripción.