

Material de estudio OBLIGATORIO EJE 2 Angular - Programación Reactiva

Sitio: [Instituto Superior Politécnico Córdoba](#)

Curso: Programador Web - TSDWAD - 2022

Libro: Material de estudio OBLIGATORIO EJE 2 Angular -
Programación Reactiva

Imprimido por: Ezequiel Maximiliano GIAMPAOLI

Día: miércoles, 10 mayo 2023, 4:44 PM

Tabla de contenidos

1. Introducción

2. MVC en Angular

3. Consumir servicios API Rest

3.1. Ejemplo. Consumir API Rest de Prueba.

4. Observables y Promesas

Esta sección tiene por objeto repasar los conceptos básicos de MVC pero desde el punto de vista del frontend a fin de integrar el backend con el frontend. Para ello, utilizaremos inicialmente un API de prueba: <https://www.npmjs.com/package/json-server> o bien un archivo data.json.

En esta sección nos introduciremos a la programación reactiva a fin de dejar lo más preparado posible el frontend para luego, ser integrado con el backend.

Objetivos

- Integrar el patrón MVC a una aplicación de Angular.
- Crear modelos, vistas y controladores basados en el patrón MVC desde la perspectiva de Angular.
- Consumir un servicio API Rest de prueba.

El modelo MVC (modelo-vista-controlador) es un patrón de arquitectura de software, que separa la parte visual, de la funcional y las estructuras de datos. Por ello, se compone de 3 componentes:

- el modelo (la parte de acceso a datos), permite acceder a los datos y a los mecanismos para manipularlos. Ej. a través de un servicio que se conecte con un API Rest o servicio web (backend).
- la vista (la parte visual), presenta el modelo en un formato adecuado para la interacción del usuario. Ej. formularios.
- el controlador (la parte funcional), controla las interacciones entre la vista y el modelo. Ej. reglas de negocio.

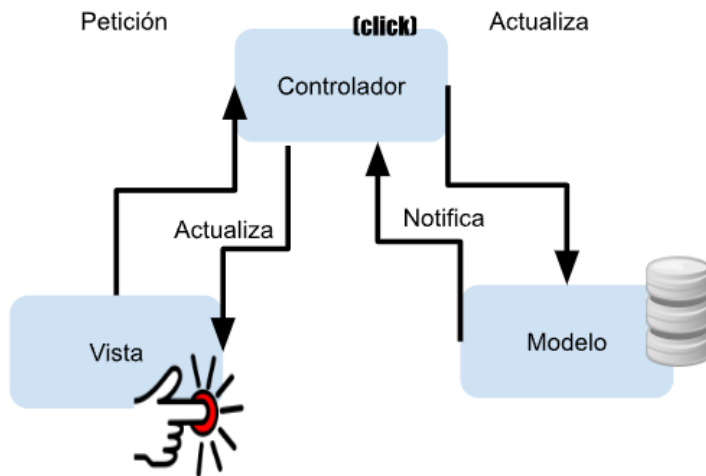


Figura: Patrón MVC

Para comprenderlo, supongamos que en la vista hacemos clic en un botón (html), a continuación el controlador (ts) que está escuchando eventos, detecta el clic del botón y le dice al modelo (service) que actualice los datos. El modelo notifica al controlador sobre dicha acción y este finalmente actualiza la vista.

De acuerdo a estos conceptos básicos y teniendo en cuenta que Angular es un framework de frontend, podemos presentar la implementación del modelo MVC en Angular como sigue:

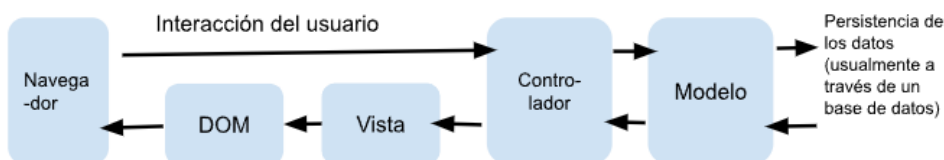


Figura: Patrón MVC desde la perspectiva de Angular (primer approach)

Dónde el objetivo del controlador y las vistas es operar los datos del modelo para manipular así el DOM de manera que el usuario pueda interactuar con él ya sea para acceder a los datos o manipularlos.

Sin embargo, es importante mencionar que angular difiere un poco en cuanto a esta terminología dado que la arquitectura de Angular, como mencionamos previamente, se basa en módulos y componentes. Lo que quiere decir, que la implementación del modelo MVC en Angular más correcto luce como sigue:

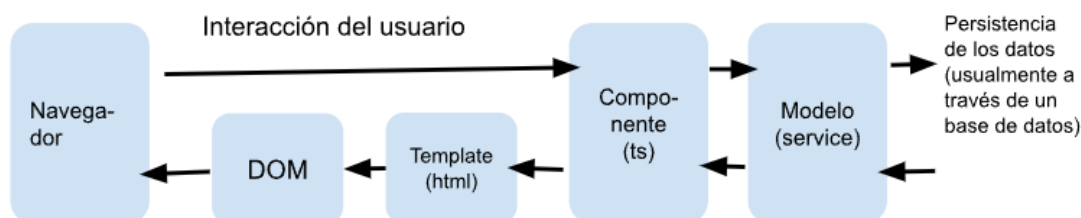


Figura: Patrón MVC desde la perspectiva de Angular

Modelos desde la perspectiva de Angular

Los modelos en angular encapsulan los datos y definen la lógica para manipularlos. Por ejemplo, un modelo puede representar un personaje en un video juego, una cuenta en una billetera digital, etc.

Los modelos deberían:

- contener la lógica para crear, administrar y modificar los datos del dominio (incluso si eso significa ejecutar la lógica remota a través de servicios web)
- exponer de manera prolija los datos del modelo y las operaciones en ella.

Y no deberían:

- exponer cómo el modelo de datos es obtenido o administrado. No debería exponer por ejemplo el API Rest o servicio web a la vistas y componentes.
- contener lógica en relación a la interacción con el usuario (porque es trabajo del componente).
- contener lógica de presentación de datos (porque es trabajo del template).

Controladores/componentes desde la perspectiva de Angular

Los controladores, conocidos como componentes en Angular, son los que permiten la interacción entre el template y el modelo. Por lo general, los componentes agregan la lógica de negocio requerida para presentar algunos aspectos al modelo y ejecutar acciones sobre estos.

En otras palabras, los controladores interpretan las acciones del usuario realizadas en la vista y comunican datos nuevos o modificados al modelo. A su vez, cuando los datos cambian el modelo notifica al controlador quien comunica los nuevos datos a la vista para mostrarlos.

Los componentes deberían:

- contener la lógica necesaria para configurar el estado inicial del template.
- contener la lógica y comportamientos requeridos para presentar datos del modelo.
- contener la lógica y comportamientos requeridos para actualizar el modelo en función de la interacción del usuario.

Y no deberían:

- contener la lógica que manipula el DOM (porque es trabajo del template)
- contener la lógica que gestiona la persistencia de los datos (porque es trabajo del modelo)

Vistas desde la perspectiva de Angular

Las vistas, conocidas como template o plantillas, se definen mediante elementos HTML las cuales, están mejoradas en Angular gracias a los sistemas de enlaces (data binding).

Los template deberían:

- contener la lógica de presentación.

Y no deberían:

- contener la lógica compleja.
- contener la lógica que manipula el modelo.

Nota: Las vistas pueden tener lógica (por ej. a través de las directivas) pero debe ser simple y debe usarse con moderación.

Consumir servicios API Rest

Hasta ahora hemos visto qué es un servicio y cómo inyectarlo en un componente para que este finalmente muestre los datos en pantalla. Sin embargo, los datos por el momento se encuentran en duro en el servicio. Es por ello que, debemos comenzar a pensar en la comunicación con el backend a fin de, acceder y manipular los datos que residen persistentemente en la base de datos. ¿Cómo?, mediante la comunicación con un servicio API Rest.

Una API es un conjunto de rutas que provee un servidor (backend) que permiten el acceso y la manipulación de los datos.

En nuestro caso, estaremos trabajando con APIs Rest dado que se caracterizan por no tener estado. Es decir, es como si no tuvieran memoria, cada petición es independiente de la anterior. Es por ello que, es necesario proveer toda la información necesaria en la petición HTTP.

Consumir un servicio API Rest es de suma importancia en cualquier proyecto dado que, nos permite separar las responsabilidades de cada componente de la aplicación y facilita el manejo de peticiones http de manera asíncrona. A continuación, se describen algunas de las ventajas de este enfoque:

- **Separación de responsabilidades:** Al utilizar servicios para consumir API REST, se separa la lógica de la comunicación con la API del resto de la aplicación. Esto facilita el mantenimiento y la escalabilidad del proyecto, ya que cada componente y servicio tiene una función específica y bien definida.
- **HttpClient:** Angular proporciona el módulo HttpClient para realizar llamadas HTTP. Este módulo ofrece una API basada en observables de RxJS, lo que permite un manejo más flexible y reactivo de las respuestas y errores de las peticiones HTTP.
- **Asíncrono:** Los servicios de Angular trabajan de forma asíncrona, lo que significa que las peticiones HTTP y el procesamiento de datos se realizan en segundo plano, sin bloquear el hilo principal de la aplicación. Esto permite una mejor experiencia de usuario, ya que la interfaz no se bloquea mientras se espera la respuesta de la API.
- **Reutilización de código:** Al encapsular la lógica de comunicación con la API en un servicio, se puede reutilizar este servicio en diferentes componentes de la aplicación. Esto evita la duplicación de código y facilita la actualización de la lógica de comunicación en caso de cambios en la API.
- **Gestión de errores:** Al utilizar los servicios de Angular y HttpClient, se puede manejar de manera centralizada y uniforme los errores que puedan ocurrir durante las peticiones HTTP. Esto facilita la implementación de estrategias de manejo de errores y la presentación de mensajes de error al usuario.

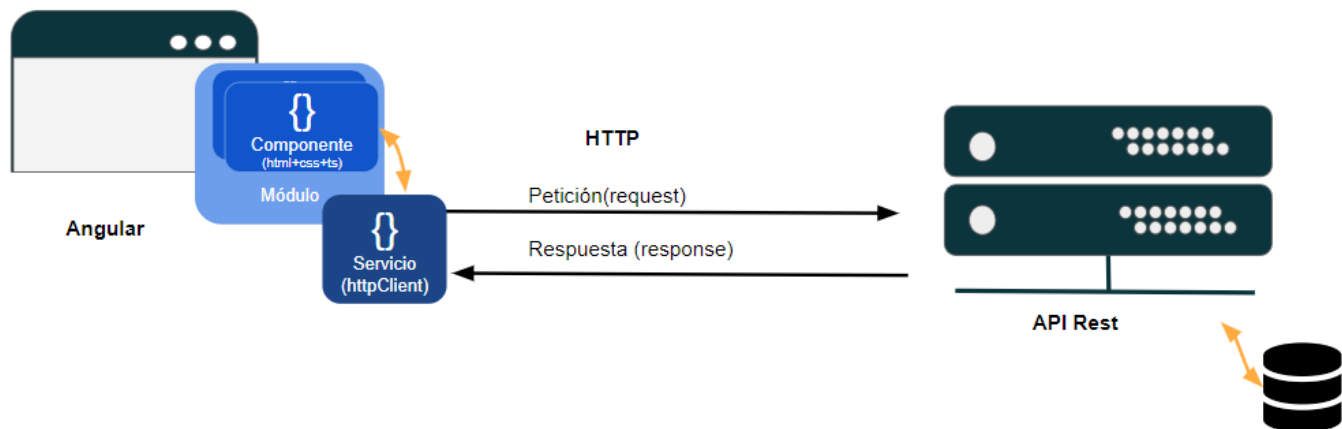


Figura: Implementación angular - api rest

Como puedes observar en la figura de arriba, es el servicio quién se conecta con el el API Rest mediante el protocolo HTTP.

Recordemos que el protocolo HTTP es quién define cómo los clientes y servidores se comunican entre sí. Es decir que, cuando un cliente desea comunicarse con un servidor, realiza los siguiente pasos:

1. Abre una conexión TCP: esta conexión se utilizará para realizar una o varias peticiones y recibir respuestas.
2. Realiza una petición HTTP.
3. Lee la respuesta enviada por el servidor.



Investiga. ¿Cuáles son las secciones en las que se dividen tanto las peticiones HTTP, como las respuestas HTTP? ¿Qué indican los métodos en las peticiones HTTP? ¿Qué indican los códigos de estado en las respuestas HTTP? ¿Por qué será importante conocer esto?

Para realizar la comunicación a un servicio mediante el protocolo HTTP, Angular nos provee un artefacto que nos facilita la tarea, el servicio `httpClient`.

Para más información, visita la fuente oficial de Angular en relación a la comunicación con el backend mediante servicios utilizando `http`: <https://angular.io/guide/http>;

¿Cómo consumir un servicio API Rest?

Para ello:

1. Importar el módulo `HttpClientModule`, en la sección `imports` de el `app.module.ts` como sigue:

```
...  
  
import { HttpClientModule } from '@angular/common/http';  
  
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    HttpClientModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

Nota: `HttpClient` usa Observables de RxJS. Para más información consulta el sitio oficial: <https://rxjs-dev.firebaseapp.com/>

2. Importar e inyectar en el constructor del servicio `<name-servicio>.service.app` que consumirá la API Rest:

```
import { Injectable } from '@angular/core';  
  
import { HttpClient } from '@angular/common/http';  
  
@Injectable({  
  providedIn: 'root'  
})  
export class UsuarioService {  
  
  constructor(private http: HttpClient) {  
  }  
}
```

Inyectar HttpClient en el constructor del servicio.

Con esto, ya podemos hacer llamadas `http`!!

Llamadas HTTP

A continuación se enumeran los métodos de petición `http` más utilizados:

- **GET**, permite recuperar recursos del servidor (datos). Si la respuesta es positiva (200 OK), el método GET devuelve la representación del recurso en un formato concreto: HTML, XML, JSON u otro. De lo contrario, si la respuesta es negativa, devuelve 404 (not found) o 400 (bad request).

- **POST**, permite crear o ejecutar acciones sobre recursos del servidor. Generalmente se utiliza este método cuando se envían datos de un formulario al servidor. Si la respuesta es positiva, el método POST devuelve 201 (created).
- **PUT**, permite modificar recursos del servidor (aunque permite también crear). Si la respuesta es positiva, el método PUT devuelve 201(created) o 204 (no response).
- **DELETE**, permite eliminar recursos del servidor. Si la respuesta es positiva, el método DELETE devuelve 200 junto con un body response, o 204 sin body.

Peticiones HTTP desde Angular

Llamadas Get

Son las llamadas más frecuentes, y permiten obtener datos desde el servidor.

Sintaxis:



```
this.http.get("https://url-api-rest");
```

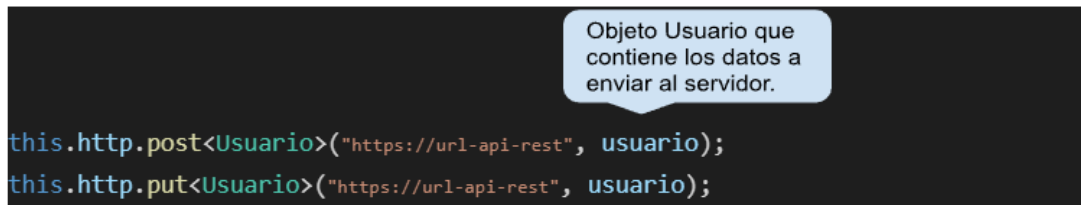
Llamadas Post y Put

Las llamadas POST y PUT sirven para enviar datos al servidor y que éste nos responda.

Las peticiones POST se usan frecuentemente para crear recursos como por ejemplo: crear usuarios mientras que las peticiones PUT, se usan para actualizar un objeto previamente creado.

En ambos casos, se pasa un objeto o conjunto de objetos a crear o modificar.

Sintaxis:



```
this.http.post<Usuario>("https://url-api-rest", usuario);  
this.http.put<Usuario>("https://url-api-rest", usuario);
```



Investiga, ¿cómo se realiza una petición http para hacer uso del método Delete?

Consumir un servicio API Rest de prueba

Para ello, vamos a partir del servicio cuenta desarrollado previamente:

```
TS cuenta.service.ts X
src > app > services > TS cuenta.service.ts > ...
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root'
5  })
6  export class CuentaService {
7
8    constructor() { }
9
10   ObtenerUltimosMovimientos()
11   {
12     return [{operacion:"Extracción",monto:1500}, {operacion:"Depósito", monto:1520}];
13   }
14 }
15
```

Como podemos observar, el servicio cuenta.service retorna un arreglo de operaciones en duro cosa que deberemos modificar a fin de consumir un API Rest dado que es quién proveerá los datos finalmente. Pero ¿cómo hacerlo si aún no tenemos el backend?

Para ello, utilizaremos un api de pruebas [JSON Server](#).

JSON Server es una herramienta que nos permite crear un servidor API Rest de prueba de manera rápida y sencilla utilizando un archivo JSON como base de datos para almacenar y manipular los datos. Esta herramienta es especialmente útil para el desarrollo frontend ya que, permite simular la interacción con una API real sin necesidad de crear el backend.

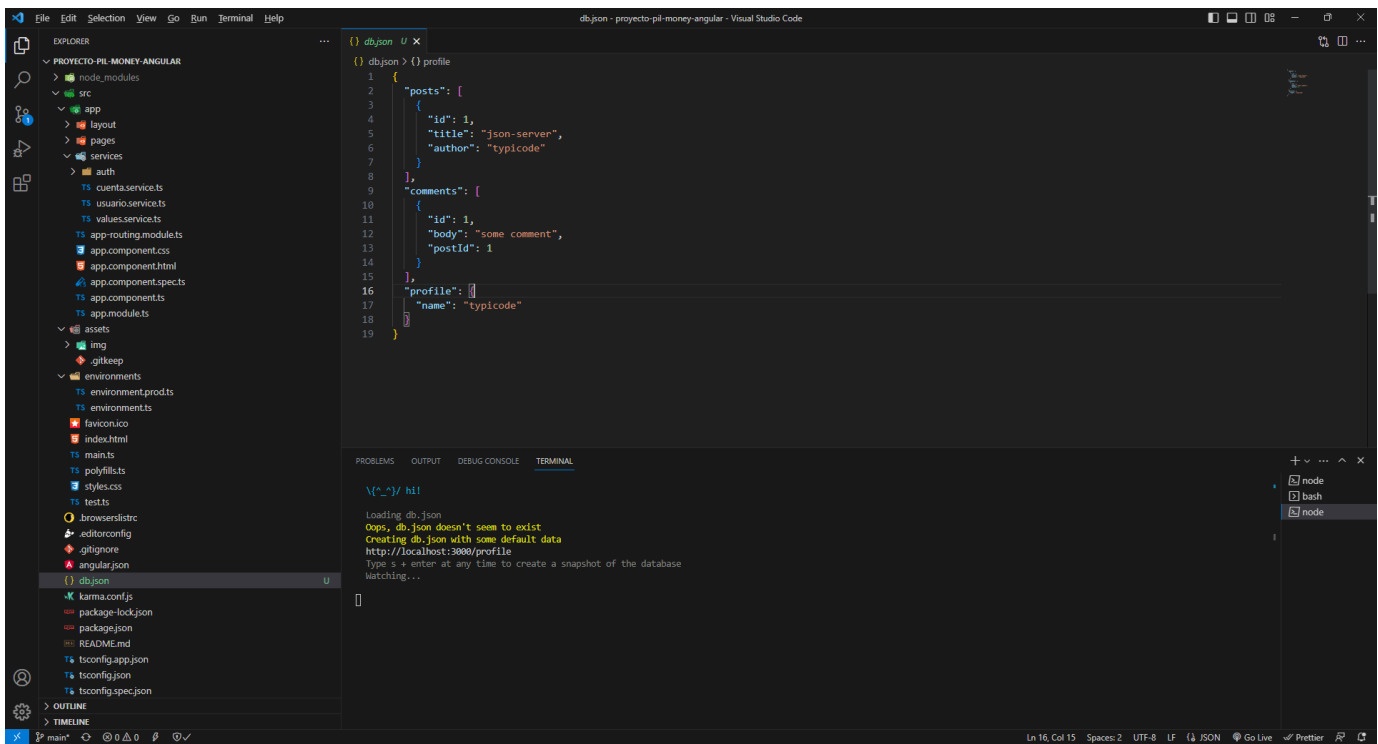
JSON, por sus siglas en inglés (JavaScript Object Notation), es un formato de texto que se deriva de la sintaxis de JavaScript y se utiliza para el acceso, almacenamiento e intercambio de datos. A pesar de estar basado en JavaScript, JSON puede ser utilizado independientemente de este lenguaje, ya que muchos entornos de programación tienen la capacidad de leer y manipular JSON.

JSON Server se utiliza para almacenar y manipular datos. La herramienta nos permite crear endpoints de API Rest de prueba lo que permite realizar peticiones y recibir respuestas como si se tratara de una API real.

Para utilizar JSON Server, ejecutar los siguientes pasos:

1. Ir a la consola DOS o "Símbolo del Sistema" del sistema operativo o Terminal de VSCode.
2. Instalar JSON Server. Para ello, ejecutar el comando: **npm install -g json-server**
3. Levantar el servicio JSON Server. Para ello, ejecutar el comando: **json-server --watch db.json**

Al levantar el servicio, si no existe el archivo db.json, JSON Server lo creará por nosotros:



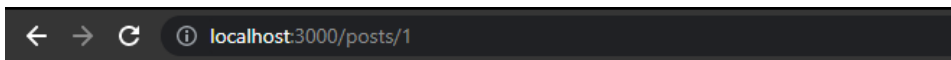
The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left displaying the project structure for 'PROYECTO-PIL-MONEY-ANGULAR'. The main editor area shows the 'db.json' file with the following content:

```
{
  "posts": [
    {
      "id": 1,
      "title": "json-server",
      "author": "typicode"
    }
  ],
  "comments": [
    {
      "id": 1,
      "body": "some comment",
      "postId": 1
    }
  ],
  "profile": {
    "name": "typicode"
  }
}
```

The bottom panel shows the Terminal with the following output:

```
\(^_^)/ hi!
Loading db.json
Oops, db.json doesn't seem to exist
Creating db.json with some default data
http://localhost:3000/profile
Type s + enter at any time to create a snapshot of the database
Watching...
```

Ahora, utilizando un browser o bien, una aplicación de testing de Apis (como por ej. Postman) para evaluar los endpoints, evaluar la ruta: <http://localhost:3000/posts/1>.

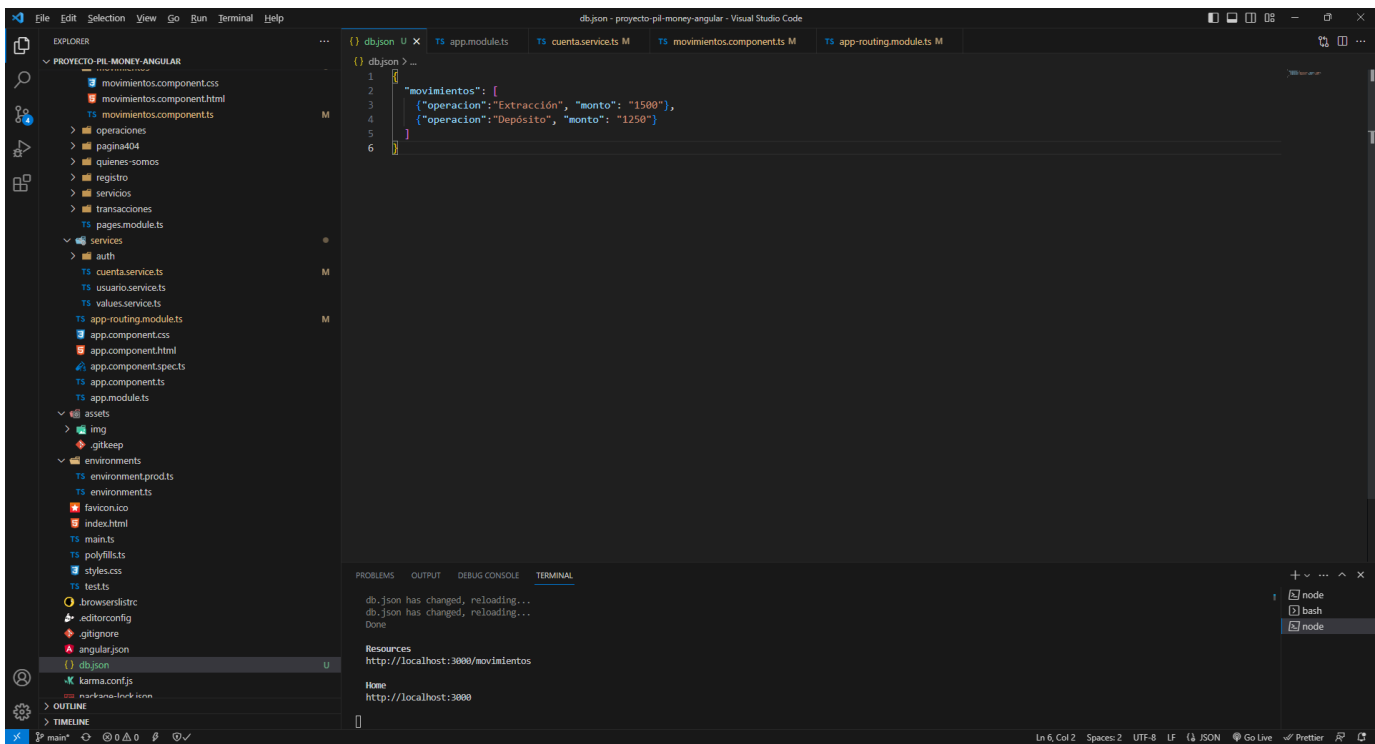


```
{
  "id": 1,
  "title": "json-server",
  "author": "typicode"
}
```

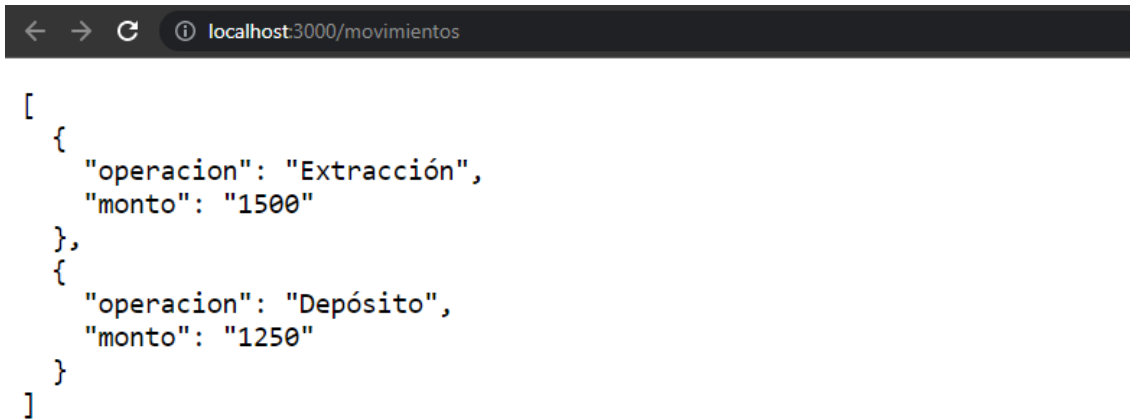
Observa que este servidor de pruebas, por defecto está en el puerto 3000 y nos provee los datos especificados en el archivo db.json

4. Modificar el archivo db.json a fin de ajustar al modelo de datos requerido.

Siguiendo nuestro ejemplo, modificamos como sigue:



Si el servicio está corriendo podremos acceder a los datos a través de la ruta: <http://localhost:3000/movimientos> como sigue:



5. Consumir el Api Rest de prueba en el servicio. Para ello:

a. Importar el módulo `HttpClientModule`, en la sección `imports` de el `app.module.ts` como sigue:

```
...
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

b. Importar e inyectar en el constructor del servicio <<name-servicio>>.service.app que consumirá la API Rest:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class UsuarioService {

  constructor(private http:HttpClient) {

  }
}
```

Inyectar HttpClient
en el constructor del
servicio.

c. En el servicio cuenta.service, crear un atributo para almacenar la url y modificar el método obtenerUltimosMovimientos a fin de realizar una petición http y devolver un observable como sigue:

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient } from '@angular/common/http';
3  import { Observable } from 'rxjs';
4
5  @Injectable({
6    providedIn: 'root'
7  })
8  export class CuentaService {
9    url:String= "http://localhost:3000/";
10   constructor(private http:HttpClient) { }
11
12   ObtenerUltimosMovimientos(): Observable <any>
13   {
14     return this.http.get(this.url+"movimientos");
15   }
16 }
17
```

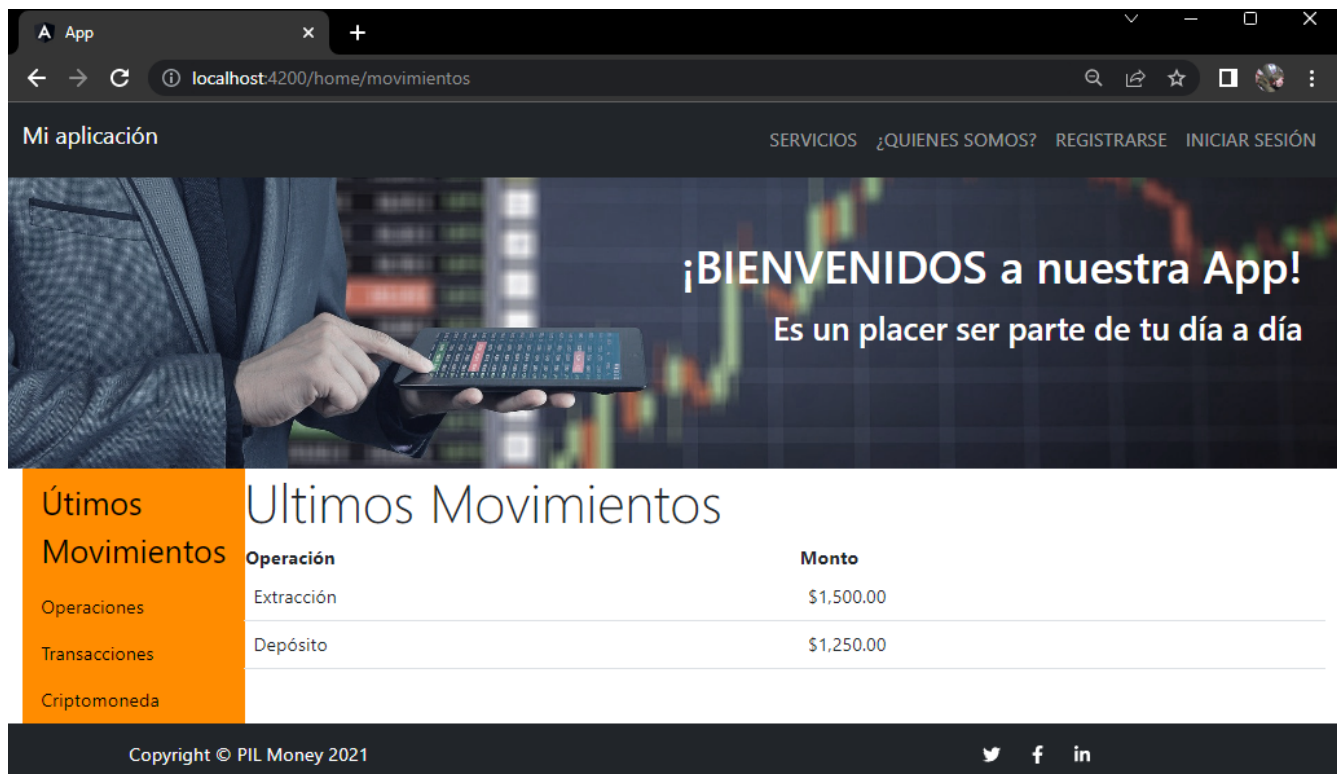


Investiga. ¿Por qué no es recomendable trabajar con el tipo any? ¿Cómo podrías resolver?

d. En el componente movimientos.component modificar la llamada al servicio a fin de subscribirse al observable como sigue:

```
1 import { Component, OnInit } from '@angular/core';
2 import { CuentaService } from 'src/app/services/cuenta.service';
3
4 @Component({
5   selector: 'app-movimientos',
6   templateUrl: './movimientos.component.html',
7   styleUrls: ['./movimientos.component.css']
8 })
9 export class MovimientosComponent implements OnInit {
10   hoy= new Date();
11   mostrarMovimientos: boolean=true;
12   movimientos:any;
13   constructor( private cuenta: CuentaService)
14   {
15     this.cuenta.ObtenerUltimosMovimientos().subscribe({
16       next: (movimientosData) => {
17         this.movimientos=movimientosData
18       },
19       error: (errorData) => {
20         console.error(errorData);
21       }
22     });
23   }
24
25   ngOnInit(): void {
26   }
27
28 }
```

Si todo va bien, deberíamos poder ver los registros de las operaciones ahora de manera dinámica en nuestro frontend como sigue:



The screenshot shows a web browser window with the URL `localhost:4200/home/movimientos`. The page has a dark theme and a navigation bar with links: `SERVICIOS`, `¿QUIENES SOMOS?`, `REGISTRARSE`, and `INICIAR SESIÓN`. The main content area features a large banner with the text `¡BIENVENIDOS a nuestra App!` and `Es un placer ser parte de tu día a día`. Below the banner, there is a section titled `Últimos Movimientos` which contains a table of recent transactions.

	Operación	Monto
Operaciones	Extracción	\$1,500.00
Transacciones	Depósito	\$1,250.00
Criptomoneda		

The footer of the page includes the text `Copyright © PIL Money 2021` and social media icons for Twitter, Facebook, and LinkedIn.



Modifica el servicio `cuenta.service` desarrollado previamente a fin de consumir el API Rest de prueba y finalmente inyectarlo en el

componentes `movimientos` para mostrar finalmente los datos en pantalla.

Observables

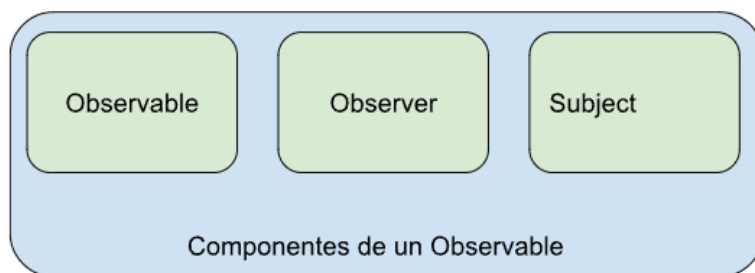
El observable (observador) es un patrón de diseño de software en el que un objeto (sujeto), mantiene una lista de dependientes, llamados observadores a quienes les notifica automáticamente los cambios de estado.

Los Observables brindan soporte para pasar mensajes entre partes de su aplicación. Se utilizan con frecuencia en Angular y son la técnica recomendada para el manejo de eventos, la programación asíncrona y el manejo de múltiples valores. (<https://docs.angular.lat/guide/observables>).

En otras palabras, un observable no es más que una colección de futuros eventos a los que nos suscribimos y que llegarán de forma asíncrona.

La utilización de los observables ayuda a aumentar el rendimiento al hacer menos consultas repetitivas para acceder a la fuente de información.

Componentes de un observable:



- **Observable**, hace referencia al encargado generar un evento, aquello que queremos observar.
- **Observer**, hace referencia al observador, quien se dedica a observar.
- **Subject**, hace referencia al emisor del evento, quien crea el flujo de eventos cuando el observable sufre cambios. Eventos que serán consumidos por el observador.

Nota: observable y subject, son librerías de RXJS. La programación Reactiva es un paradigma de programación asíncrono interesado en los flujos de datos y la propagación al cambio. RxJS (Por sus siglas en Inglés, "Reactive Extensions for JavaScript") es una librería para programación reactiva usando observables que hacen más fácil la creación de código asíncrono o basado en callbacks. (<https://docs.angular.lat/guide/comparing-observables>).

Promesas

Es un objeto que representa la terminación o el fracaso de una operación asíncrona. Se utilizan por lo general para la realización de tareas asíncronas, como por ejemplo la obtención de respuesta a una petición HTTP.

Una promesa puede tener 4 estados:

- **Pendiente:** Es su estado inicial, no se ha cumplido ni rechazado.
- **Cumplida:** La promesa se ha resuelto satisfactoriamente.
- **Rechazada:** La promesa se ha completado con un error.
- **Arreglada:** La promesa ya no está pendiente. O bien se ha cumplido, o bien se ha rechazado.

Sintaxis:

```
new Promise(function(resolve, reject) { ... });
```

Cuando creamos una promise, le pasamos una función en cuyo interior deberían producirse las operaciones asíncronas, que recibe 2 argumentos:

- **Resolve:** Es la función que llamaremos si queremos resolver satisfactoriamente la promesa.
- **Reject:** Es la función que llamaremos si queremos rechazar la promesa.

Nota: Al momento de trabajar con proyectos robustos lo recomendable es utilizar los observables, ya que presentan más ventajas en su lógica.

Diferencias entre Promesas y Observables

Observables	Promesas	Observaciones
Son declarativos; La ejecución no comienza hasta la suscripción.	Las promesas se ejecutan inmediatamente después de la creación.	Esto hace que los observables sean útiles para definir recetas que se pueden ejecutar cuando necesites el resultado
Proporcionan muchos valores.	Las promesas proporcionan un valor.	Esto hace que los observables sean útiles para obtener múltiples valores a lo largo del tiempo.
Diferencian entre encadenamiento y suscripción.	Solo tienen cláusulas <code>.then()</code>	Esto hace que los observables sean útiles para crear recetas de transformación complejas para ser utilizadas por otra parte del sistema, sin que el trabajo se ejecute.
<code>subscribe()</code> es responsable de manejar los errores	Generan los errores a promesas hijas	Esto hace que los observables sean útiles para el manejo centralizado y predecible de errores.

Tabla : comparación de observables y promesas.

Fuente: <https://docs.angular.lat/guide/comparing-observables>