

Módulo VI: Angular, integración con MVC

AUTORAS: ROJAS CÓRSICO, Ivana Soledad, ALVARADO, Mónica Yamil

Introducción	2
Objetivos	2
MVC en Angular	2
Modelos desde la perspectiva de Angular	4
Controladores/componentes desde la perspectiva de Angular	4
Vistas desde la perspectiva de Angular	5
Conceptos Previos	5
Observables	5
Promesas	6
Diferencias entre Promesas y Observables	7
Consumir un API Rest desde Angular	8
Llamadas HTTP	9
Peticiones HTTP desde Angular	9
Llamadas Get	9
Llamadas Post y Put	10
Implementación del patrón MVC en Angular	10
Crear el Servicio Usuarios (Modelo)	11
Crear el componente Registro de Usuario (Vista y Controlador)	12
Autenticación en Angular	18
Crear el servicios para autenticación	19
Crear Guards	22
Crear Interceptors	24
Referencias	27

Introducción

Esta sección tiene por objeto repasar los conceptos básicos de MVC desarrollados previamente pero desde el punto de vista del frontend a fin de integrar el backend con el frontend.

En esta sección ya con conceptos aprendidos de angular y backend se procederá a terminar de dar forma a nuestra aplicación, y la conexión del front-end con el back-end.

Objetivos

1. Integrar el patrón MVC a una aplicación de Angular.
2. Crear modelos, vistas y controladores basados en el patrón MVC desde la perspectiva de Angular.
3. Consumir un API Rest para conectar con el backend.
4. Agregar autenticación a una aplicación de Angular.

Nota: Todo el fuente desarrollado en este documento puedes descargarlo del repositorio <https://github.com/pgClipfs/proyecto-pil-money-angular>

MVC en Angular

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.

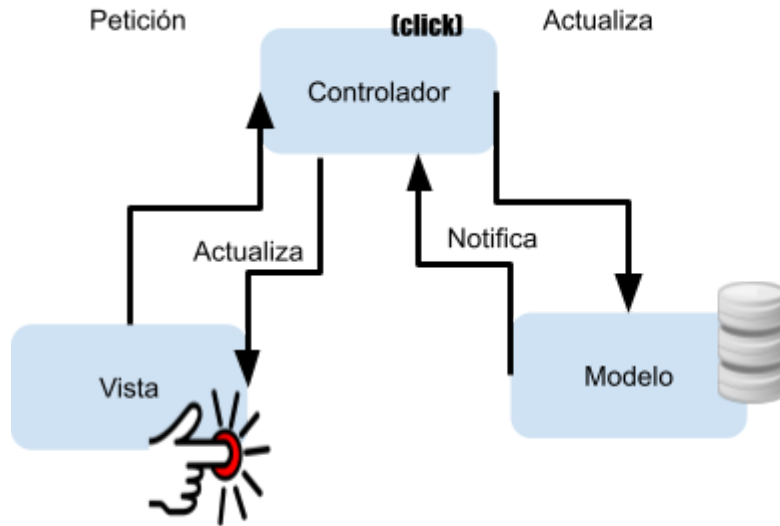


Figura 1: Patrón MVC

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:

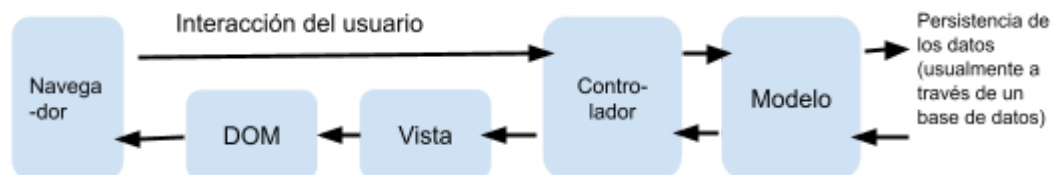


Figura 2: Patrón MVC desde la perspectiva de Angular (primer approach)

Nota: los datos se obtienen del backend (usualmente a través de un API Rest).

donde 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 en el módulo 2, se basa en módulos y componentes. Lo que quiere decir, que la implementación del modelo MVC en Angular más correcto luce como sigue:

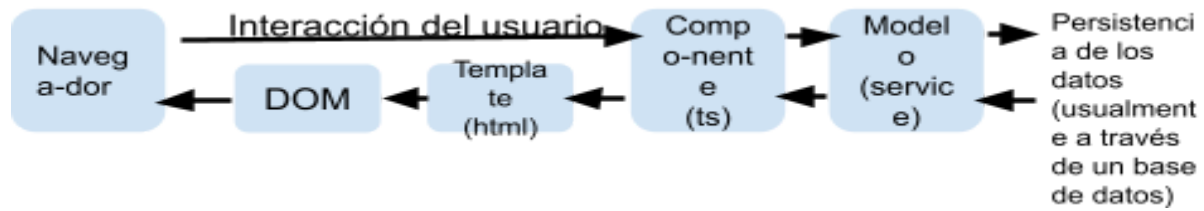


Figura 3: Patrón MVC desde la perspectiva de Angular

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.

Conceptos Previos

Para comprender cómo implementar MVC en Angular, introduciremos algunos conceptos importantes.

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 asincrónica 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:

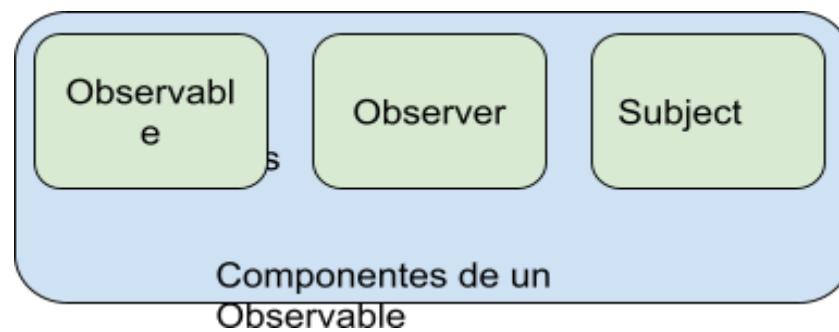


Figura 49: Componentes de un observable.

- **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 .then ()	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.
subscribe() 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>

Consumir un API Rest desde Angular

Hasta ahora hemos visto qué es un servicio y cómo inyectar en el componente y hacer uso del mismo. Los datos por el momento fueron simulados o escritos en duro en el servicio. Sin embargo para acceder a ellos o manipularlos desde una API REST tenemos que hacer llamadas HTTP.

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

Para hacer las llamadas HTTP, Angular nos provee un módulo que facilita esta tarea, el módulo `httpClient`.

Para usar `HttpClient` de Angular en cualquier parte, tenemos que:

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:

```
Url del Api Rest.  
  
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.

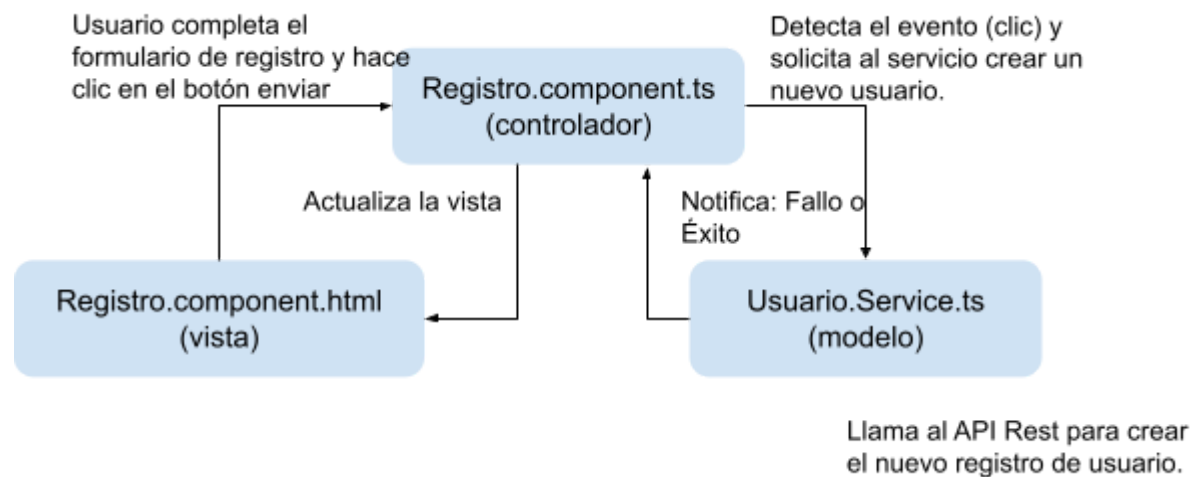
Sintaxis:

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

Objeto Usuario que contiene los datos a enviar al servidor.

Implementación del patrón MVC en Angular

Continuando con la aplicación que venimos desarrollando, y pensando en el patrón MVC desde la perspectiva de Angular, veamos como hacer un registro de usuario.



Como pudimos observar en los gráficos de MVC, el modelo contiene los datos y los mecanismos necesarios para manipularlos. Es entonces el modelo, el punto final del frontend dado que, es a partir de ahí debe conectarse con un servicio web o API Rest para acceder a los datos, es decir con el backend. Los modelos de datos pueden ser simples como por ejemplo para trabajar un CRUD (Create, Read, Update y Delete) o mucho más complejos.

En Angular, como mencionamos previamente, el modelo está codificado en servicios.

Figura 4: Funcionalidad “Crear Registro de Usuario” desde la perspectiva MVC de Angular.

Nota: A fin de poder hacer las pruebas desde el frontend se utilizará el API Rest disponible en <https://reqres.in/>

Crear el Servicio Usuarios (Modelo)

En nuestro ejemplo, el servicio **UsuarioService** es quién nos permitirá acceder y manipular los datos. Para ello,

1. Ir a la consola DOS o “Símbolo del Sistema” del sistema operativo o Terminal de VSCode.
2. Ejecutar el comando: **ng generate service services/usuario**

o su abreviado: **ng g s services/usuario**

Nota: Recuerda que services es la carpeta donde le especificamos a AngularCLI que cree el archivo.

Una vez ejecutado el comando, AngularCLI crea un archivo generará el archivo **usuario.service.ts**

3. Editar el servicio a fin de:
 - a. Importar las clases HttpClient de @angular/common/http y Observable de rxjs.
 - b. Editar el archivo usuario.service.ts a fin de hacer la petición POST.

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

export class Usuario
{
  nombre:string="";
  apellido:string="";
  dni:string="";
  fechaNacimiento:string="";
  password:string="";
  email:string="";
  id:number=0;
  //A modo de ejemplo se deja así pero lo ideal es crear propiedades para acceder a
  los atributos.
}

@Injectable({
  providedIn: 'root'
```

```
    })  
    export class UsuarioService {  
        url="https://reqres.in/api/users/1";  
  
        constructor(private http:HttpClient) {  
            console.log("Servicio Usuarios está corriendo");  
        }  
  
        onCrearUsuario(usuario:Usuario):Observable<Usuario>{  
            return this.http.post<Usuario>(this.url, usuario);  
        }  
    }  
}
```

API Rest de ejemplo:
<https://reqres.in/>

Método al que luego el componente (ts) podrá subscribirse. El mismo, recibe un objeto Usuario como parámetro y devuelve un objeto Usuario (modificado con el id y demás datos que puedan obtenerse del servidor).

Observa que, además de importar **HttpClient**, también hemos importado la clase **Observable**. Esto es porque después necesitamos acceder a los datos desde el componente.

Además, y tal como se definió previamente, se puede observar que el servicio (modelo) contiene datos (objeto usuario) y el mecanismo para acceder y manipular dichos datos (método onCrearUsuario).

Nota: No olvidar importar el módulo **HttpClientModule**, en la sección imports de el **app.module.ts**

Crear el componente Registro de Usuario (Vista y Controlador)

El siguiente paso es crear un componente que hará de Vista (html) y Controlador (ts).

1. Ir a la consola DOS o "Símbolo del Sistema" del sistema operativo o Terminal de VSCode.
2. Ejecutar el comando: **ng generate c pages/registro**
3. o su abreviado: **ng g s pages/registro**

Nota: Recuerda que estos archivos ya los hemos creado previamente (en el módulo 5: Angular)

4. En la vista, archivo **registro.component.html**, editar a fin de crear un formulario de registro:

```

<div class="container">
  <div class="row">
    <div class="col-md-12 registro-form-header">
      <p class="registro-form-font-header">Registro<span> Usuario</span></p>
    </div>
  </div>
  <div class="row m-3">
    <div class="col-md-12 registro-form">
      <form [formGroup]="form" class="form" (ngSubmit)="onEnviar($event, usuario)" >
        <div class="m-3">
          <label for="nombre" class="form-label">Nombre</label>
          <input type="text" id="nombre" class="form-control"
formControlName="nombre" [(ngModel)]="usuario.nombre" [class.border-danger]="NombreValid" >
          <div *ngIf="Nombre?.errors && Nombre?.touched">
            <p *ngIf="Nombre?.hasError('required')" class="text-danger">
              El nombre es requerido.
            </p>
          </div>
        </div>
        <div class="m-3">
          <label for="apellido" class="form-label">Apellido</label>
          <input type="text" id="apellido" class="form-control"
formControlName="apellido" [(ngModel)]="usuario.apellido"
[class.border-danger]="ApellidoValid">
          <div *ngIf="Apellido?.errors && Apellido?.touched">
            <p *ngIf="Apellido?.hasError('required')" class="text-danger">
              El apellido es requerido.
            </p>
          </div>
        </div>
        <div class="m-3">
          <label for="dni" class="form-label">Dni</label>
          <input type="number" id="dni" class="form-control" formControlName="dni"
[(ngModel)]="usuario.dni" [class.border-danger]="DniValid">
          <div *ngIf="Dni?.errors && Dni?.touched">
            <p *ngIf="Dni?.hasError('required')" class="text-danger">
              El dni es requerido.
            </p>
          </div>
        </div>
        <div class="m-3">
          <label for="fecha_nacimiento" class="form-label">Fecha de
Nacimiento</label>
          <input type="date" id="fecha_nacimiento" class="form-control"
formControlName="fechaNacimiento" [(ngModel)]="usuario.fechaNacimiento"
[class.border-danger]="FechaNacimientoValid">
          <div *ngIf="FechaNacimiento?.errors && FechaNacimiento?.touched">
            <p *ngIf="FechaNacimiento?.hasError('required')" class="text-danger">
              La fecha de nacimiento es requerida.
            </p>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>

```

```

<div class="m-3">
  <label for="email" class="form-label">Correo Electrónico</label>
  <input type="email" id="email" class="form-control"
formControlName="email" [(ngModel)]="usuario.email" [class.border-danger]="MailValid">
  <div *ngIf="Mail?.errors && Mail?.touched">
    <p *ngIf="Mail?.hasError('required')" class="text-danger">
      El email es requerido.
    </p>
  </div>
</div>
<div class="m-3">
  <label for="password1" class="form-label">Contraseña</label>
  <input type="password" id="password1" class="form-control"
formControlName="password1" [(ngModel)]="usuario.password"
[class.border-danger]="Password1Valid">
  <div *ngIf="Password1?.errors && Password1?.touched">
    <p *ngIf="Password1?.hasError('required')" class="text-danger">
      El password es requerido.
    </p>
  </div>
</div>
<div class="m-3">
  <label for="password2" class="form-label">Reingrese la Contraseña</label>
  <input type="password" id="password2" class="form-control"
formControlName="password2" [class.border-danger]="Password2Valid">
  <div *ngIf="Password2?.errors && Password2?.touched">
    <p *ngIf="Password2?.hasError('required')" class="text-danger">
      El password es requerido.
    </p>
  </div>
</div>
<div class="m-3">
  <button class="btn btn-primary mt-3" >Enviar</button>
</div>
</form>
</div>
</div>
</div>

```

Nota: Observa que el mismo incluye formulario, validaciones y two way binding por lo que, se deberá además configurar previamente todo lo relacionado a ello (ver módulo 5, formularios reactivos).

5. En el controlador, archivo **registro.component.ts**, editar a fin de:

a. Importar el UsuarioService y Usuario:

```

import { UsuarioService , Usuario} from
'src/app/services/usuario.service';

```

b. Inyectar el UsuarioService en el constructor

```

constructor(private usuarioService: UsuarioService)

```

- c. Editar el evento onSubmit (onEnviar) del formulario a fin de hacer la petición al modelo para que éste finalmente haga la petición http.

```
onEnviar(event: Event, usuario:Usuario): void {
    event.preventDefault();

    if (this.form.valid)
    {
        console.log("Enviando al servidor...");
        console.log(usuario);

        this.usuarioService.onCreateUsuario(usuario).subscribe(
            data => {

                if (data.id>0)
                {
                    alert("El registro ha sido creado satisfactoriamente. A continuación,
por favor Inicie Sesión.");

                    this.router.navigate(['/iniciar-sesion'])
                }
            })
        else
        {
            this.form.markAllAsTouched();
        }
    }
};
```

Controlador solicita al Modelo
Crear un Nuevo Usuario por medio de la ejecución del método onCreateUsuario(usuario). Pasa por parámetros el objeto Usuario.

Subscribe, funciona como las promesas de Javascript permitiendo esperar hasta que la petición termine. Observa que dentro del método se ejecuta una arrow function, la misma permite devolver el objeto data que contiene la respuesta del UsuarioService.

data. Puedes ver que contiene aqui:
<https://reqres.in/api/users/1>

Redirecciona a la ruta de inicio de sesión a fin de obligar al usuario iniciar sesión. Previamente se debe importar router e inyectar en el constructor.

El fuente completo del controlador, archivo usuario.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { UsuarioService, Usuario } from 'src/app/services/usuario.service';
import { Router } from '@angular/router';

@Component({
    selector: 'app-registro',
    templateUrl: './registro.component.html',
    styleUrls: ['./registro.component.css']
})
```

```
export class RegistroComponent implements OnInit {
  form:FormGroup;
  usuario: Usuario = new Usuario();

  constructor(private formBuilder: FormBuilder, private usuarioService:
UsuarioService, private router: Router) {
    this.form= this.formBuilder.group(
      {
        nombre:['', [Validators.required]] ,
        apellido:['', [Validators.required]],
        dni:['', [Validators.required]],
        fechaNacimiento:['', [Validators.required]],
        password1:['', [Validators.required]],
        password2:['', [Validators.required]],
        email:['', [Validators.required, Validators.email]]
      }
    )
  }

  ngOnInit(): void {
  }

  onEnviar(event: Event, usuario:Usuario): void {
    event.preventDefault;

    if (this.form.valid)
    {
      console.log("Enviando al servidor...");
      console.log(usuario);
      this.usuarioService.onCrearUsuario(usuario).subscribe(
        data => {
          console.log(data.id);
          if (data.id>0)
          {
            alert("El registro ha sido creado satisfactoriamente. A continuación,
por favor Inicie Sesión.");
            this.router.navigate(['/iniciar-sesion'])
          }
        })
    }
    else
    {
      this.form.markAllAsTouched();
    }
  };

  get Password1()
  {
    return this.form.get("password1");
  }
  get Password2()
  {
    return this.form.get("password2");
  }
}
```



```
}

get Mail()
{
    return this.form.get("email");
}

get Nombre()
{
    return this.form.get("nombre");
}

get Apellido()
{
    return this.form.get("apellido");
}

get FechaNacimiento()
{
    return this.form.get("fechaNacimiento");
}

get Dni()
{
    return this.form.get("dni");
}

get MailValid()
{
    return this.Mail?.touched && !this.Mail?.valid;
}

get NombreValid()
{
    return this.Nombre?.touched && !this.Nombre?.valid;
}

get ApellidoValid()
{
    return this.Apellido?.touched && !this.Apellido?.valid;
}

get Password1Valid()
{
    return this.Password1?.touched && !this.Password1?.valid;
}

get Password2Valid()
{
    return this.Password2?.touched && !this.Password2?.valid;
}

get FechaNacimientoValid()
{

```

```
return this.FechaNacimiento?.touched && !this.FechaNacimiento?.valid;
}

get DniValid()
{
    return this.Dni?.touched && !this.Dni?.valid;
}
}
```

Finalmente, si ejecutamos `ng serve -o`, en el caso que la aplicación no esté corriendo, podremos contemplar la funcionalidad completa de MVC en Angular funcionando.

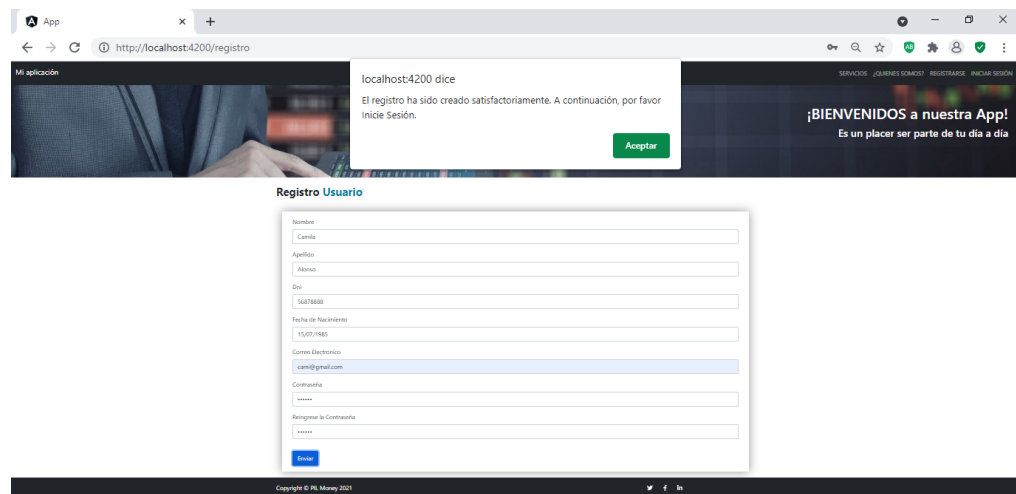


Figura 5: Formulario de Registro - MVC

Autenticación en Angular

Al momento de realizar la autenticación, el usuario ingresa sus credenciales (mail y contraseña), los mismos son enviados al backend y una vez verificada la existencia del mismo, se anexa un token para seguridad, todo el conjunto de datos son enviados nuevamente al Frontend, y dependiendo de estos datos entregados se mostrará el componente correspondiente, como por ejemplo:

- Un tablero de control que permite realizar actividades diferentes según el rol del usuario.
- Un mensaje de bienvenida para diferentes roles.
- etc.

En nuestro ejemplo, el usuario podrá acceder al home, el cual contiene el acceso a últimos movimientos, operaciones, transacciones, etc.

Para generar la autenticación del usuario en nuestra aplicación utilizaremos una librería llamada JWT (JSON WEB TOKEN) en el backend.

Nota: JWT se deberá configurar previamente todo lo relacionado a ello (ver módulo 6, seguridad informática).

Crear el servicios para autenticación

1. Dentro de la carpeta de servicio crear una carpeta contenedora de todos los servicios necesarios para realizar la autenticación de los usuarios. Nosotros la llamaremos "Auth"

Nota: Recuerda que services es la carpeta donde le especificamos a AngularCLI que cree el archivo.

2. Generar el servicio para la lógica de autenticación **auth.service.ts**
3. Editar el servicio a fin de:
 - a. Importar las clases Observable de rxjs, map de rxjs y servicio usuario generado con anterioridad

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { Usuario } from '../usuario.service';
```

- b. Referenciar la misma API utilizada en el registro de usuario.

```
url="https://reqres.in/api/login";
```

- c. Declarar el **BehaviorSubject** y el **observable** como propiedades:

```
currentUserSubject: BehaviorSubject<Usuario>;
currentUser: Observable<Usuario>;
```

- d. Inyectar el HttpClient en el constructor e inicializamos las propiedades creadas previamente:

```
constructor(private http:HttpClient) {
    console.log("Servicio de Autenticación está corriendo");
    this.currentUserSubject = new
    BehaviorSubject<Usuario>(JSON.parse(localStorage.getItem('currentUser') || '{}'));
    this.currentUser = this.currentUserSubject.asObservable();
}
```

- e. Agregar la lógica de inicio de sesión. Método **login**.

Método al que luego el componente (ts) podrá subscribirse. El mismo, recibe un objeto Usuario como parámetro y devuelve un objeto Usuario (modificado con el id y demás datos que puedan obtenerse del servidor).

```
login(usuario: Usuario): Observable<any> {  
  
  return this.http.post<any>(this.url, usuario)  
  
    .pipe(map(data => {  
  
      localStorage.setItem('currentUser', JSON.stringify(data ));  
  
      data. Puedes ver que  
      contiene aquí:  
https://reqres.in/api/users/1  
  
      this.currentUserSubject.next(data);  
  
      this.loggedIn.next(true);  
  
      return data;  
  
    }));  
}
```

Nota: Utilizaremos el API Rest disponible en <https://reqres.in/> como lo hicimos en el Registro de usuario.

- f. Agregar la lógica, para realizar el cierre de sesión

```
logout(): void{  
  
  localStorage.removeItem('currentUser');  
  
  this.loggedIn.next(false);  
  
}
```

- g. Agregar propiedades para acceder a los datos del Usuario autenticado

```
get usuarioAutenticado(): Usuario {  
  
  return this.currentUserSubject.value;  
}  
get estaAutenticado(): Observable<boolean> {  
  return this.loggedIn.asObservable();  
}
```

4. En el controlador, archivo **inicia-sesion.component.ts**, editar a fin de:
- Importar el servicio recién creado

```
import { Router } from '@angular/router';  
  
import { AuthService } from 'src/app/services/auth/auth.service'
```

- Inyectar el servicio en el constructor

```
constructor(private FormBuilder: FormBuilder,  
  
private authService: AuthService, ←  
private router: Router) {  
  this.form = this.formBuilder.group(  
    {  
      password: ['', [Validators.required, Validators.minLength(8)]],  
      mail: ['', [Validators.required, Validators.email]]  
    }  
  )  
}
```

- Crear y editar el evento onSubmit (onEnviar) del formulario a fin de hacer la petición al modelo para que éste haga la petición.

```
onEnviar(event: Event, usuario: Usuario): void {  
  event.preventDefault;  
  
  this.authService.login(this.usuario)  
    .subscribe(  
      data => {  
        console.log("DATA" + JSON.stringify( data));  
  
        this.router.navigate(['/home/movimientos']);  
      },  
      error => {  
        this.error = error;  
      }  
    );  
}
```

Controlador solicita al Modelo la autenticación de Usuario por medio de la ejecución del método login(usuario). Pasa por parámetros el objeto Usuario.

Redirecciona a la ruta de de movimientos al usuario. Previamente se debe importar router e inyectar en el constructor.

- d. Editar la vista (inicio-sesion.component.html) a fin de agregar el evento.

```
<form [formGroup]="form" (ngSubmit)="onEnviar($event, usuario)">
...
</form>
```

- e. Finalmente, ejecutar ng-serve para evaluar el comportamiento. Observaremos que si iniciamos sesión con los datos: email: eve.holt@regres.in y password: cityslicka, la aplicación auténtica y redirecciona al home/movimientos:

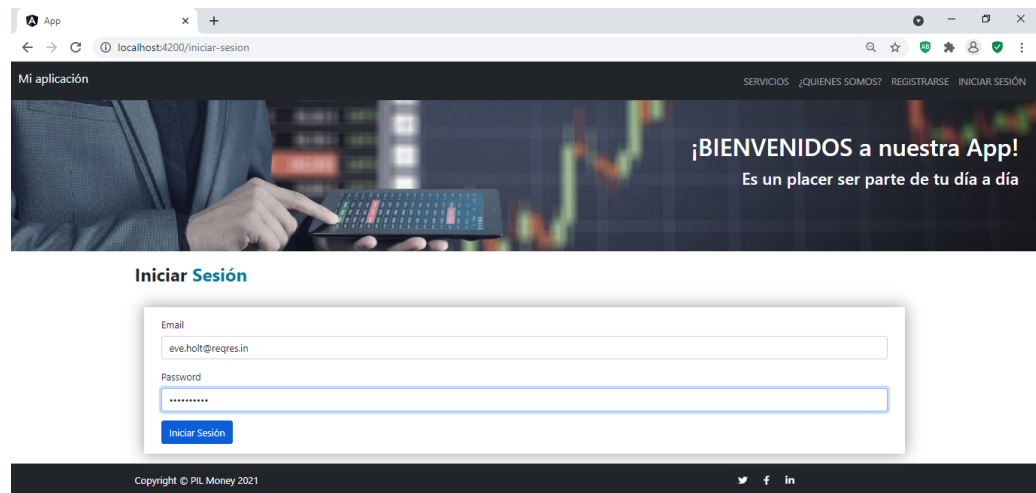


Figura 7: Formulario de inicio de sesión

Nota: El usuario y password son los requeridos por la API de prueba que estamos usando: <https://regres.in/>

Crear Guards

Los Guards son utilizados, cuando requerimos que algunas url (o áreas) de la aplicación estén protegidas de forma que solo puedan ser vistas o accedidas cuando el usuario está autenticado o bien posea un rol específico. Caso contrario, el usuario no tendrá acceso a esta url o área de la aplicación.

1. En el directorio **auth** generar el servicio **auth.guard.ts**. Luego, editar a fin de:
 - a. Importar el servicio previamente creado (**auth.guard.ts**), las clases e interfaces como sigue:

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

Contiene la información sobre una ruta asociada a un componente cargado en una salida en un momento determinado. `ActivatedRouteSnapshot` también se puede utilizar para atravesar el árbol de estado del enrutador.

Fuente:
<https://runbook.dev/es/docs/angular/api/router/activatedroutesnapshot#descripcion>

La función en la ruta es la que hará el llamado del **Guard**, y dependiendo lo que este devuelva, la ruta podrá activarse o mostrarse, o no. Por eso se llama así, **CanActivate**. Pero bueno, empecemos por el principio. Vamos crear nuestro primer **Guard**.

Fuente:
<https://www.codigoincorrecto.com/programacion/angular-2020-como-usar-guard-canactivate-ejemplo-de-uso/>

```
import { Router, ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot } from '@angular/router';

import { AuthService } from '../auth.service';
import { Observable } from 'rxjs';
import { map, take } from 'rxjs/operators';
```

Representa el estado de router en un determinado momento.

Fuente:
<https://angular.io/api/router/RouterStateSnapshot>

- b. Decorar la clase como inyectable e Implementar la interface `CanActivate`

```
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {...
```

- c. Inyectar en el constructor el servicio

```
constructor(
  private authService: AuthService
) { }
```

- d. Editar la lógica de la función de `canActivate`:

```
canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable <boolean> {
  return this.authService.estaAutenticado.pipe(take (1),
    map((isLoggedIn:boolean)=>isLoggedIn));
}
```

- e. En el archivo `app-routing.module.ts`, editar las rutas que requieren autenticación como sigue:

```
const routes: Routes = [
  {path: 'iniciar-sesion', component: IniciarSesionComponent},
  {path: 'home', component: HomeComponent, canActivate: [AuthGuard],
  children:[
    {path: 'operaciones', component: OperacionesComponent},
```

```
{path: 'transacciones', component: TransaccionesComponent},  
{path: 'criptomoneda', component: CriptomonedaComponent},  
{path: 'movimientos', component: MovimientosComponent},  
  ]},  
{path: 'servicios', component: ServiciosComponent},  
{path: 'quienes-somos', component: QuienesSomosComponent},  
{path: 'quienes-somos/:id', component: IntegranteComponent},  
{path: 'registro', component: RegistroComponent},  
{path: '', redirectTo: '/servicios', pathMatch: 'full'},  
...  
}
```

De esta manera, la aplicación no permitirá a los usuarios acceder al home hasta tanto no se hayan autenticado.

Crear Interceptors

Los interceptors proveen un mecanismo para interceptar y/o mutar las solicitudes y respuestas http. Por ende, los interceptors son capaces de intervenir las solicitudes de entrada y de salida de tu app al servidor y viceversa.

Nota: No debe confundirse con los Guards. Los interceptors modifican las peticiones (endpoints, servicios, o como lo quieras llamar) y los guards controlan las rutas de navegación entre páginas de la aplicación web, permitiendo o denegando el acceso por ejemplo (<https://medium.com/@insomniocode/angular-autenticaci%C3%B3n-usando-interceptors-a-26c167270f4>).

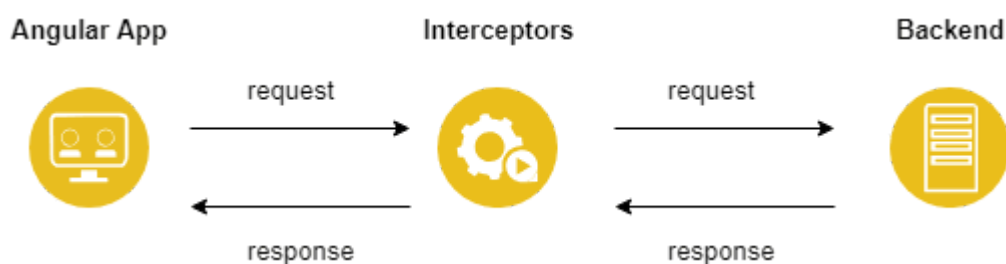


Figura 8: Interceptors

Fuente:

https://res.cloudinary.com/practicaldev/image/fetch/s--skBUyM52--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https://dev-to-uploads.s3.amazonaws.com/uploads/articles/51sp8fpfkexj3u234nwo.png

En nuestro caso, utilizaremos los interceptos de Angular para agregar el token en la cabecera 'Authorization' y de esta manera incrementar la seguridad de nuestra aplicación.

1. En el directorio **auth** generar el servicio **interceptor.ts**. Luego, editar a fin de:

a. Importar el servicio recién creado, las clases e interfaces necesarias:

```
import { Injectable } from '@angular/core';
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from
'@angular/common/http';
import { AuthService } from './auth.service';
import { Observable } from 'rxjs';
```


b. Decorar la clase como inyectable e Implementar la interfaz HttpInterceptor:

```
@Injectable({
  providedIn: 'root'
})

export class JwtInterceptor implements HttpInterceptor {
  ...
}
```

c. En el constructor, inyectar el servicio:

```
constructor(
  private authService: AuthService
) { }
```



d. Agregar la lógica que permite manipular el token que provee el API.

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

  const currentUser = this.authService.usuarioAutenticado;
  if (currentUser && currentUser.token) {
    req = req.clone({
      setHeaders: {
        Authorization: `Bearer ${currentUser.token}`
      }
    });
  }
  console.log("INTERCEPTOR: " + currentUser.token);
  return next.handle(req);
}
```

Un interceptor permite que puedas inspeccionar y/o modificar todas las solicitudes HTTP de un HttpClient. Esto significa que antes de realizar cualquier llamado al servidor, puedes editar su contenido y también la respuesta.

Fuente: <https://lautarocarro.blog/usando-http-interceptors-en-blazor/>

2. Generar el servicio para la lógica de **error.interceptor.ts**.

3. Editar el servicio a fin de:

- a. Importar las clases Observable de rxjs, map de rxjs y servicio usuario generado con anterioridad.

```
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from
 '@angular/common/http';
```

Es un operador de RXJS que se usa para crear un observable que emite una notificación de error.
Fuente: <https://www.concretepage.com/angular/angular-throwerror>

```
import { Observable, throwError } from 'rxjs';
```

El catch es un operador RxJS detecta el error generado por un observable y manipula devolviendo un observable al usuario. Fuente: <https://www.concretepage.com/angular/angular-throwerror>

```
import { catchError } from 'rxjs/operators';
```

- b. Agregar la lógica que mostrará una notificación de error:

```
@Injectable({
  providedIn: 'root'
})
export class ErrorInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(catchError(err => {
      if (err.status === 401) {
        location.reload();
      }
      const error = err.error.message || err.statusText;
      return throwError(error);
    }));
  }
}
```

Devolverá una notificación de error de por falta de credenciales válidas (401).

4. En el archivo app.module.ts

- a. Importar el archivo recién creado **Interceptor** y los servicios necesarios, **interceptor** y **error.interceptor**.

```
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { JwtInterceptor } from '../services/auth/interceptor';
import { ErrorInterceptor } from '../services/auth/error.interceptor';
```

- b. En los providers agregar los interceptors

```
providers: [UsuarioService,
  { provide: HTTP_INTERCEPTORS, useClass: JwtInterceptor, multi: true },
  { provide: HTTP_INTERCEPTORS, useClass: ErrorInterceptor, multi: true },
],
```

- c. Finalmente, ejecutar ng-serve para evaluar el comportamiento.

Nota: Iniciamos sesión con los datos: email: eve.holt@reqres.in y password: cityslicka.

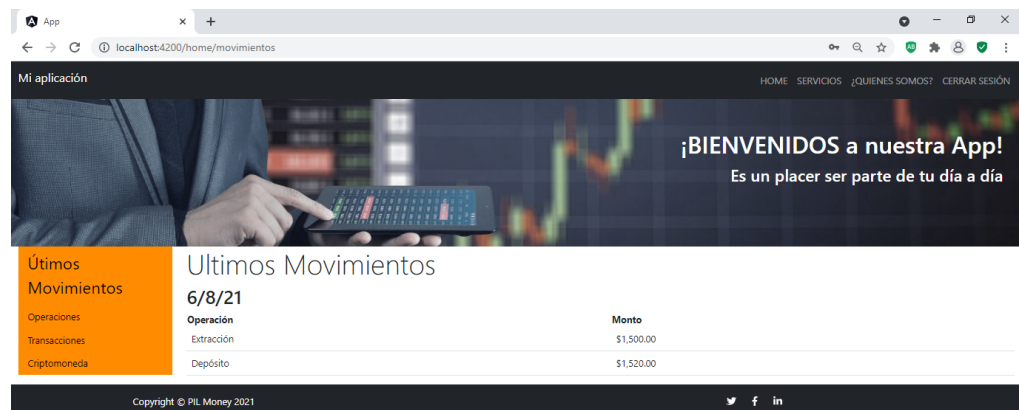


Figura 9: Redireccionamiento luego de una autenticación exitosa.

Nota: Observa que la barra de navegación también ha cambiado en función de la autenticación. Puedes observar el fuente en el componente nav.

Referencias

<https://wuschools.com/what-is-mvc-and-understanding-the-mvc-pattern-in-angular/>
<https://scotch.io/tutorials/mvc-in-an-angular-world>
<https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>

<https://codingpotions.com/angular-servicios-llamadas-http>

<https://codingpotions.com/angular-seguridad>

<https://angular.io/api/router/RouterStateSnapshot>

<https://lautarocarro.blog/usando-http-interceptors-en-blazor/>