Tema 8 – Consumo de servicios HTTP

- Acceso a servicios HTTP mediante HttpClientModule
- Llamadas asíncronas y Observables
- Manejo de errores
- Prácticas: Conexión a la API de Datos Deportivos

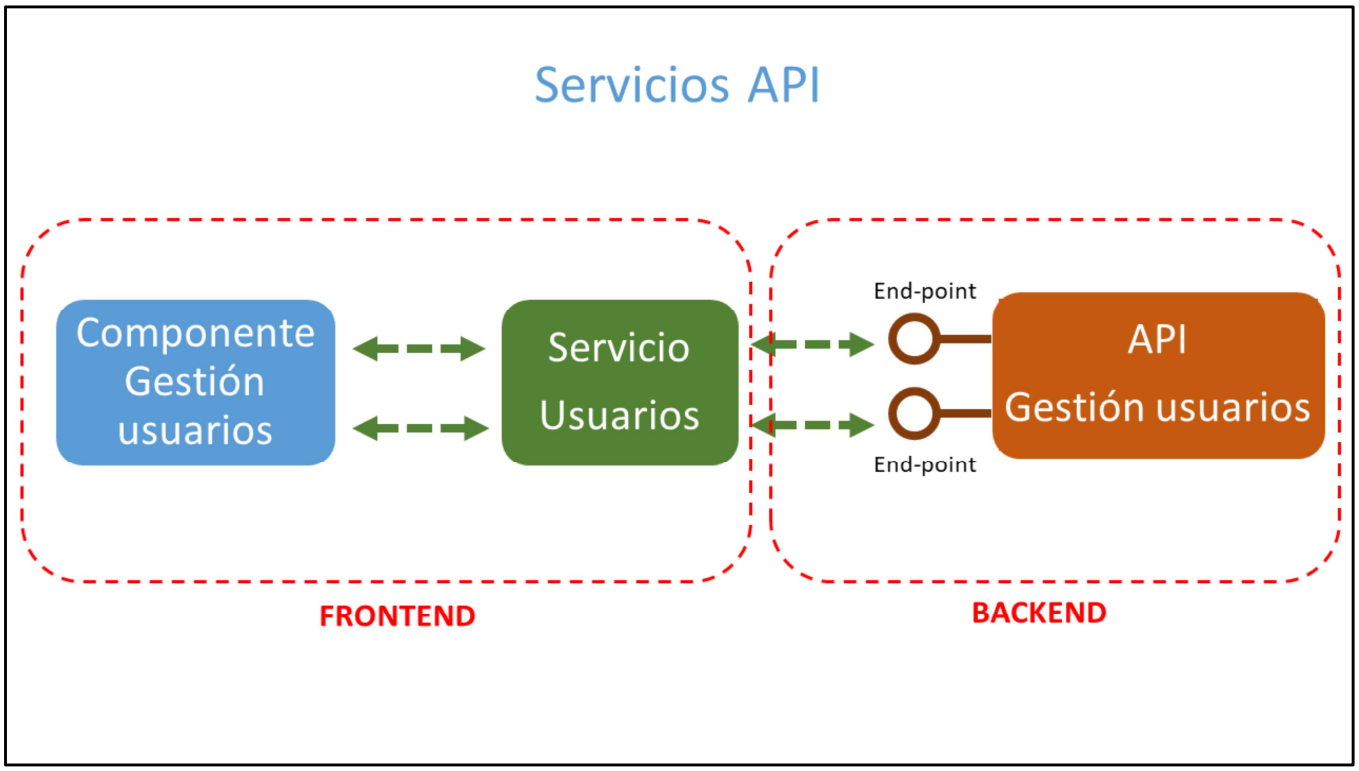
En este tema vamos a aprender a conectar nuestro frontend Angular con un backend remoto a través de HTTP.

Veremos cómo utilizar el componente HttpClient de Angular, que nos hace muy fácil el acceso a API,s. REST.

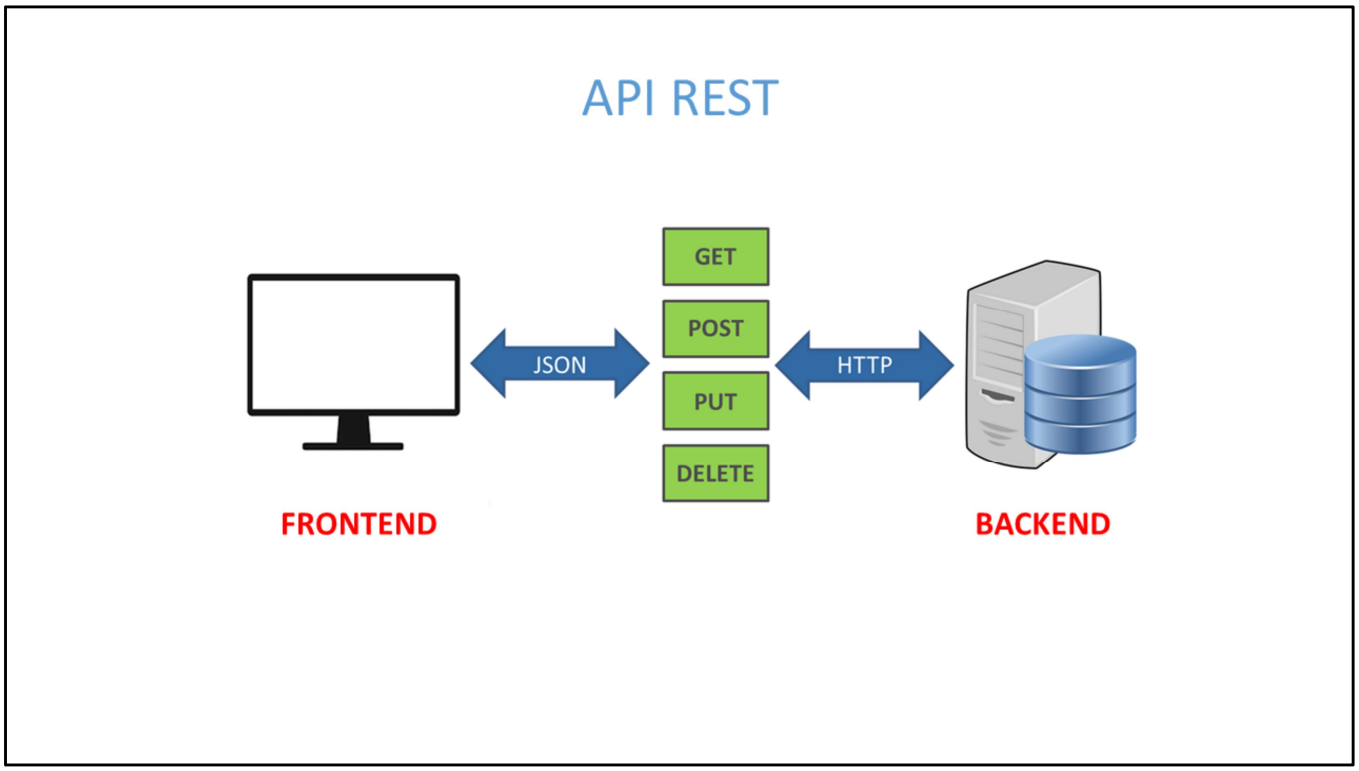
Aprenderemos qué son los Observables y cómo los usamos en Angular para gestionar llamadad asíncronas.

Por último aprenderemos a gestionar los errores en las llamadas a servicios HTTP.

En las prácticas, cambiaremos el servicio de datos local que hemos estado usando hasta ahora, por un servicio que consuma la API de Datos Deportivos desarrollada durante las prácticas de la asignatura de Backend y WS.



En una aplicación típica, el frontend hace llamadas al backend para obtener y modificar datos persistidos en bases de datos, o para consumir alguna funcionalidad especial.



Como ya habéis visto en la asignatura de Backend y WS, una API REST utiliza los métodos estándar HTTP, como son GET, POST, PUT, DELETE..., para las operaciones de lectura y modificación de datos en el backend.

La comunicación se realiza intercambiando mensajes normalmente en formato JSON, a través del protocolo HTTP, entre el navegador del cliente y el servidor remoto.

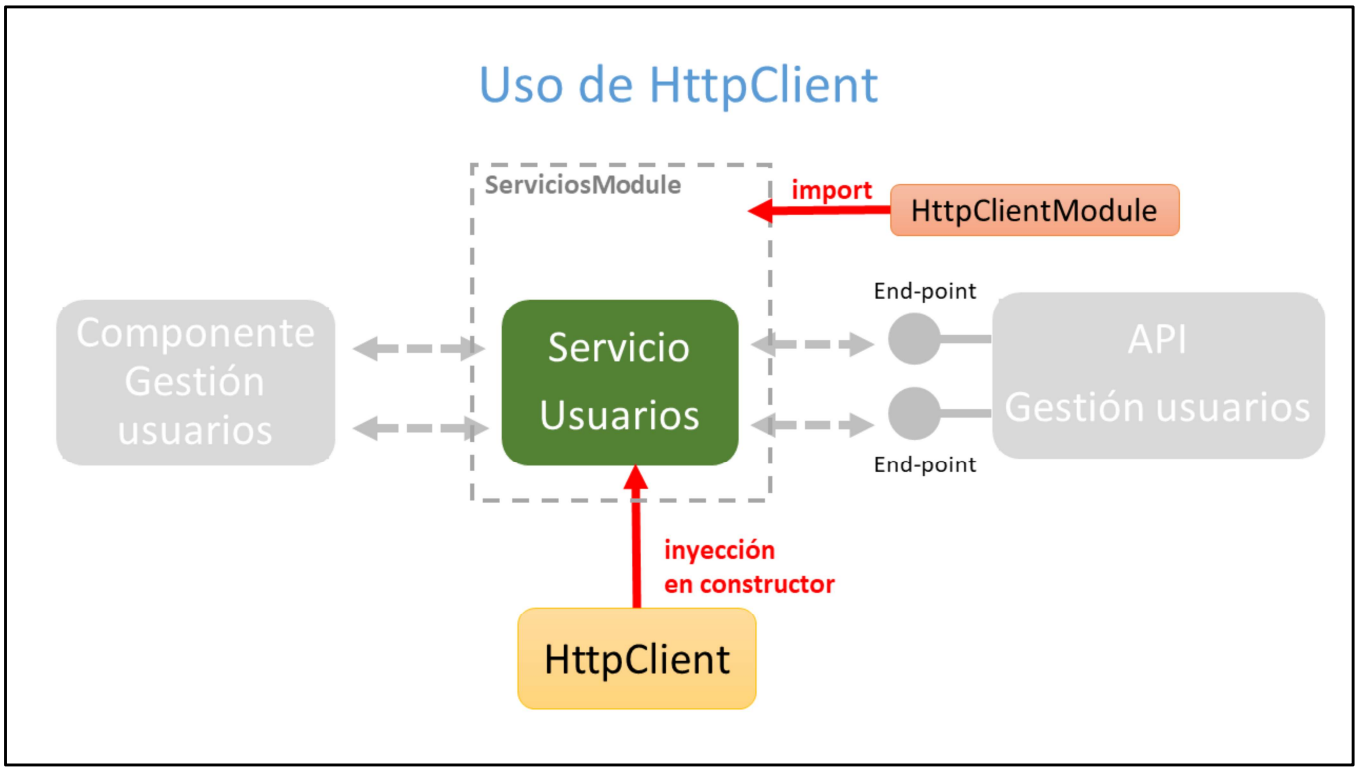
Lo que vamos a ver ahora es la otra parte de la comunicación, es decir, la forma de consumir los servicios REST expuestos en el backend, desde nuestras aplicaciones de frontend desarrolladas en Angular.

El componente HttpClient

- Pertenece al módulo **HttpClientModule** (@angular/common/http)
- Dispone de métodos *get* / *post* / *put*..., orientadas a API REST, y *request* para API,s. genéricas.
- Usa objetos de tipo Request y Response
- Gestiona las llamadas asíncronas al servidor mediante Observables
- Facilita el manejo de errores inesperados (fallo de red, de servidor...)

El servicio angular HttpClient facilita la comunicación con el servidor remoto.

Debe importarse a través del paquete @ angular/common/http



El componente HttpClient lo debemos inyectar en nuestro servicio Angular que suministra datos a los componentes, por tanto necesitaremos importar el módulo HttpClientModule en el módulo donde se ubiquen nuestros servicios.

Uso de HttpClient

mi-servicio.service.ts

TS

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

export class MiServicioService {
   constructor(private http: HttpClient){ ... }

   leer(){
      this.http.get("http://xxx.com", ...); // post, put, delete
   }

}
```

El componente HttpClient se inyecta en el constructor de nuestro servicio, como si fuera una dependencia más.

El propio módulo HttpClientModule proporciona un provider para HttpClient, por lo que no es necesario que lo configuremos manualmente en nuestra aplicación.

API de Datos Deportivos

- API REST general:
  - GET /api/partidos {?page,size,sort}
  - GET /api/participantes {?page,size,sort}
  - GET /api/goles {?page,size,sort}
  - GET /api/tarjetas {?page,size,sort}
  - GET /api/sucesos {?page,size,sort}
  - POST, PUT y DELETE para todas
  - Enlaces a entidades asociadas en cada respuesta (HATEOAS)

- Búsquedas específicas:
  - GET /api/partidos/search/participante {?idParticipante}
  - GET /api/partidos/search/con-nombre-participante {?txt}
  - GET /api/sucesos/search/participante {?idParticipante}
  - GET /api/sucesos/search/despues-de {?instant}
  - GET /api/sucesos/search/entre-fechas {?comienzo,fin}
  - GET /api/sucesos/search/participante-entre-fechas {?idParticipante, comienzo,fin}

Llamadas y funciones asíncronas

- Son aquellas que pueden bloquear la ejecución del código. Ejemplos:
  - Su ejecución lleva mucho tiempo
  - Llama a servicios de red
  - Debe esperar un evento externo
- Si se ejecutan de forma secuencial, paralizan la aplicación hasta que finalizan y/o devuelven la respuesta.
- Solución en Angular → Observables

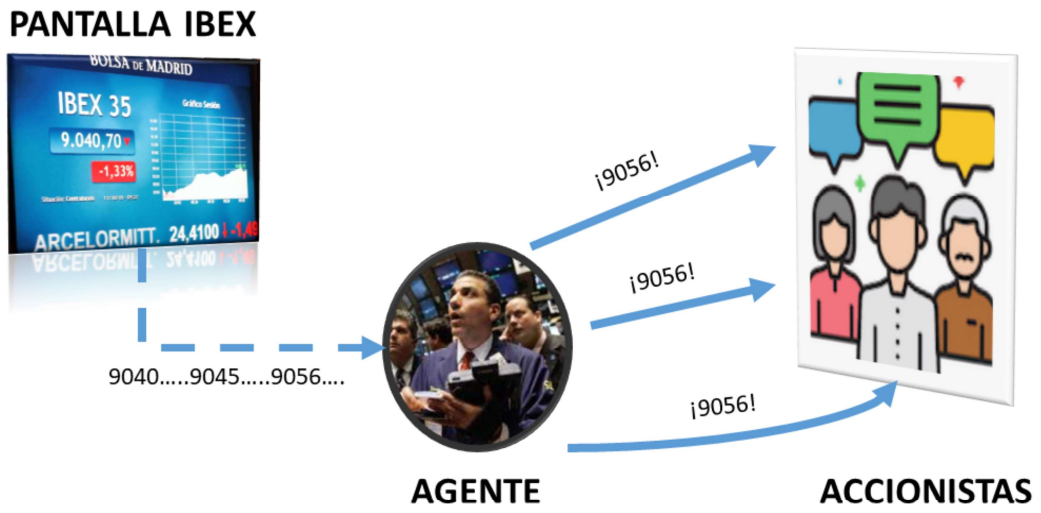
El motivo es que los métodos de la clase HttpClient son asíncronos, es decir, pueden bloquear la ejecución normal del código porque pueden tardar en responder.

Si una llamada a una API tarda 10 segundos en responder, y esa llamada se hace de forma sincronía, es decir, se espera a su respuesta, la interfaz se quedaría "congelada" durante esos 10 segundos, como si el navegador se hubiera quedado colgado.

Para tratar con este problema, se han inventado muchas soluciones a lo largo del tiempo. Angular usa de forma preferente el patrón "Observer" u observador.

Este patrón realmente se usa para muchas otras cosas no relacionadas con llamadas asíncronas, y por otra parte hay otras soluciones posibles para este problema, pero Angular ha decidido usar este patrón, y por tanto es necesario conocerlo y saber cómo se implementa en el código.

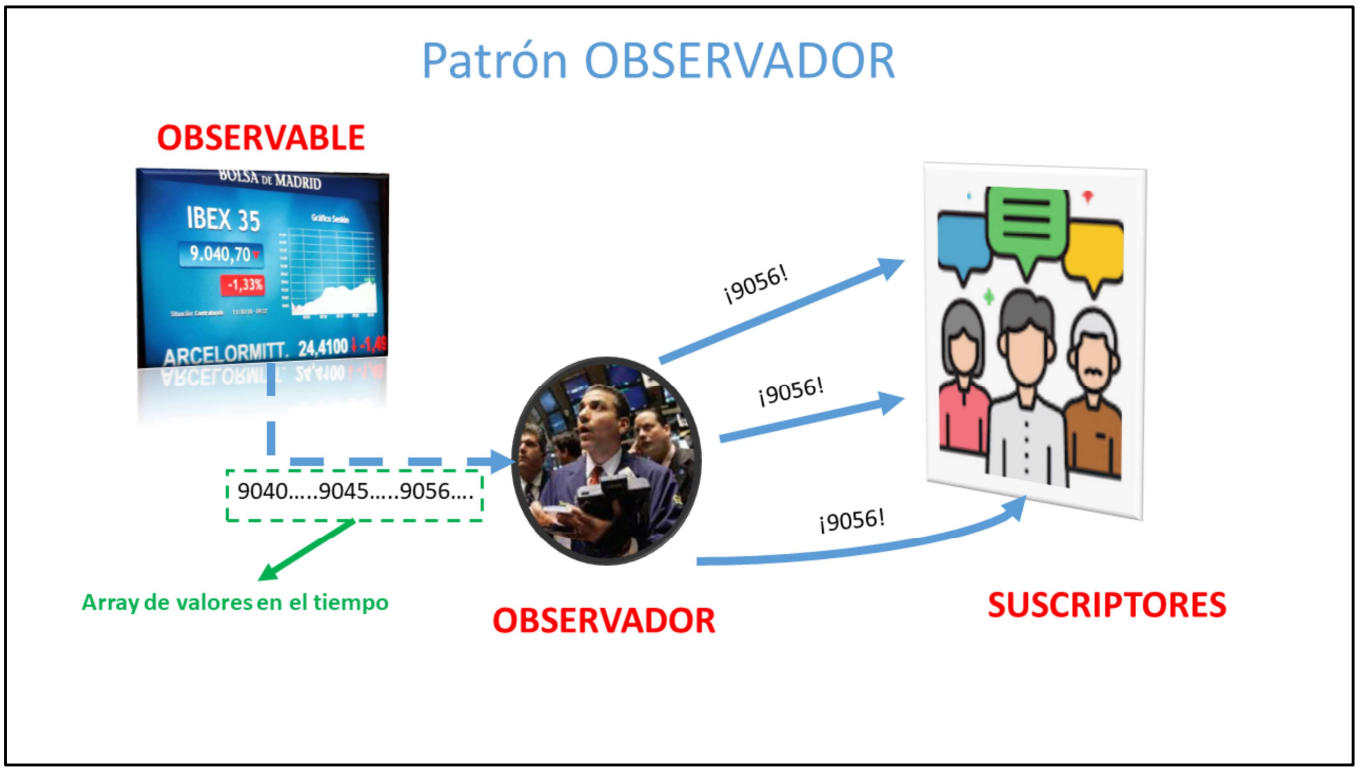
Patrón OBSERVADOR



Para explicar el patrón observador pondremos como ejemplo un agente broker de la bolsa.

Cuando se produce información relevante en la pantalla de la bolsa, el agente, que está pendiente de ello, comunica el evento a los clientes que usan sus servicios, para que estos tomen las decisiones que deseen.

Esto evita que los propios clientes tengan que estar pendientes de los cambios en los valores, no se bloqueen, y puedan realizar mientras otras tareas.



Según el patrón Observador, la pantalla del IBEX sería el Sujeto observable, y los clientes serían los Observadores.

En Angular se le da otra denominación, pero el concepto es el mismo:

- Pantalla IBEX: Observable
- Agente: Observador
- Accionistas: Suscriptores

Angular considera a un Observable, como un array de valores, que se van liberando a lo largo del tiempo, y al cual me puedo suscribir. En el ejemplo, la pantalla del IBEX sería como un array de valores del índice IBEX, que se actualiza cada intervalo. Cada vez que se produce un nuevo dato, este se envía a los suscriptores de la información.

Observables y suscriptores en Angular

- **Observable** = Tipo de objeto que produce valores a lo largo del tiempo (uno o varios), y al que me puedo suscribir.
- **Suscriptor** = Función que es invocada cada vez que el observable al que está suscrita produce un nuevo valor.
- **Observador** = Código que extrae valores del Observable, y se los pasa a los suscriptores.

Librería RxJS

- Todo lo relacionado con el uso de Observables en Angular pertenece a una librería externa, **RxJS**
  - https://angular.io/guide/rx-library
- RxJS implementa la **programación reactiva**:
  - Paradigma de programación basado en reaccionar (react) a flujos de datos variables en el tiempo: llamadas asíncronas, eventos, información cambiante...
- RxJS proporciona el tipo Observable, y varios operadores que permiten manipular los valores que suministra el observable antes de informar a los suscriptores.

Los métodos de HttpClient devuelven Observables

- Los métodos get, post, put, delete, request... de HttpClient, son asíncronos porque conllevan una operación en red.
- Por ello, devuelven un Observable al que puedo suscribirme con una función que recibe la respuesta HTTP, en el momento que se produce.

Los métodos de HttpClient devuelven Observables

- *Consecuencia*: los métodos de mis servicios Angular que llaman a métodos asíncronos, también son asíncronos.
  - *getPartidos()* se apoya en HttpClient.get(), por tanto también es asíncrona, y debe devolver un Observable.

Servicio (PartidosService)

```
getPartidos() {
    return this.http.get("http://localhost:8080/api/partidos")
}
```

Componente (PartidosLista)

```
this.partidosService.getPartidos().subscribe(
      (respuesta) => this.partidosLista = respuesta['_embedded'].partidos
)
```

Observable: El operador “map”

- La librería **rxjs** dispone de varios operadores para ajustar el tratamiento de los valores suministrador por un Observable.
- Se pueden encadenar varios operadores en un Observable mediante la función “pipe”.
- El operador **map** permite transformar cada valor que emite un Observable, antes de pasarlo a sus sucriptores.
- Es necesario importar el operador expresamente, de `'rxjs/operators'`

```
import { map } from 'rxjs/operators';
...

getPartidos() {
    return this.http.get(http://localhost:8080/api/partidos).pipe(
      map(respuesta => respuesta['_embedded'].partidos)
    )
}
```

“Cuando el observable emita un valor, envía a los suscriptores el resultado de: ***valor[‘_embedded’].partidos****”*

Modificamos por tanto el código de getPartidos(), y el de la función que lo consume en PartidosLista.

Servicio PartidosLocalService

- Tras usar Observables, no cumple con la interfaz PartidosService.
- Cualquier dato puede convertirse en un Observable que devuelve dicho valor, con la función **of( )** de RxJS.

Listas de participantes en formulario Partido

- Crear interfaz para entidad Participante
- Crear interfaz ParticipantesService, con método getParticipantes()
- Implementar ParticipantesApiService
- Usar servicio en PartidosForm (selección de equipos local y visitante)
- Evitar errores en consola durante la carga

- Crear interfaz Participante

- Crear clase abstracta ParticipantesService
    - getParticipantes -> Participantes ordenados A-Z

- Crear servicio ParticipantesApiService
    - Implementar getParticipantes
    - Meter “provide” en app.module

- En PartidosForm, usar el servicio para rellenar la lista de equipos locales y visitantes
    - Crear una propiedad local para participantesLista = [ ]
    - En el Init, obtener el Observable del servicio y en la suscripción asignar los valores a la lista
    - Añadir ngFor para rellenar los valores de los desplegables

Manejo de errores - Tipos

- **Tipos de errores** en llamadas a APIs:
  - Errores **esperados**
    - Recursos no existentes
    - Uso incorrecto de la API
  - Errores **inesperados**
    - Fallos de conexión
    - Servidor o servicio caído
    - Excepciones no controladas en la API

Hasta ahora hemos supuesto que todo va bien siempre, y que no se va a producir ningún error en el acceso a la API externa.

La realidad no es así. Se pueden producir 2 tipos generales de errores:

- Errores esperados
- Errores inesperados

Los **errores esperados** son aquellos que están previstos en la propia API, y suelen estar tipificados, con un código de error.

Los **errores inesperados** son los relacionados con fallos de conexión, servicios caídos o errores internos de la API.

Manejo de errores - Estrategias

- Los errores deben ser manejados por la aplicación, de lo contrario pueden lanzarse mensajes inesperados al usuario, e incluso abrir brechas de seguridad.

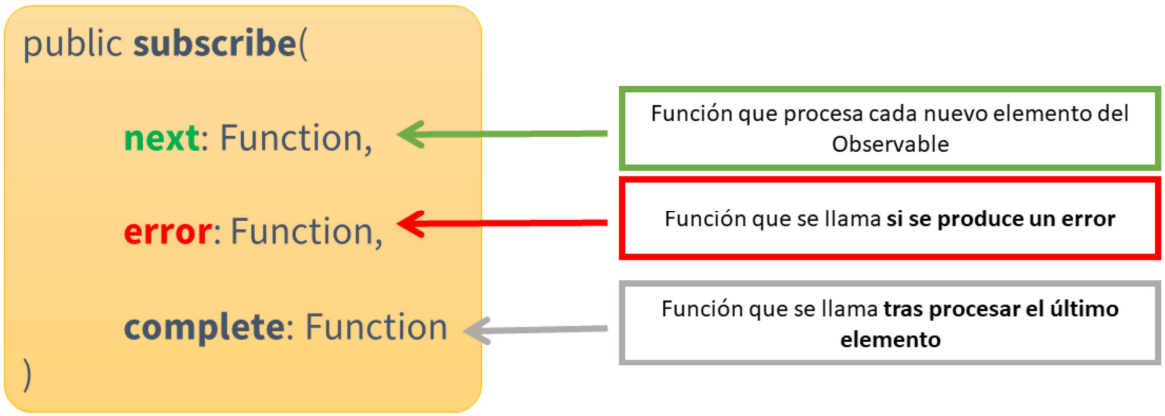
- Hay 2 estrategias para manejar los errores que lanza un Observable:

  A. Que el error **se envíe al suscriptor** para que lo maneje.
  B. Que el error **sea capturado y gestionado por el Observable** y no se envíe a los suscriptores.

- Una estrategía mixta sería que el Observable capture ciertos errores esperados, y el resto los envíe al suscriptor.

Manejo de errores por el suscriptor

- El método *subscribe* de Observable admite hasta 3 parámetros:



Angular pone bastante fácil el manejo de los errores por parte del suscriptor, porque me permite indicar en la suscripción a los Observables, un punto de entrada para los casos en los que se produce cualquier error, sea esperado o no.

- Si vemos los parámetros del método "suscribe()", el primer parámetro, que hemos usado hasta ahora, lo denomina "next()", indicando que es la función que se va a llamar cuando se reciba el siguiente elemento del Observable.
- El segundo lo denomina "error", y es otra función a la que se llamará cuando se produzca un error de cualquier tipo. Si no se especifica este parámetro, si se produce un error, se lanzará hacia arriba sin manejar.
- El tercero se denomina “complete”, y se llama tras emitirse el último valor del Observable sin que haya habido ningún error.

Manejo de errores – Formato del error

- La función que trata el error en el suscriptor, o la que captura el error en el Observable, recibe un objeto de tipo "any", que representa el error producido:

```
let obs = this.http.get(...);

obs.subscribe(
       (respuesta: Object) => { ... },
       (error: any) => { console.log('¡Error!' + error) }
)
```

- Aunque el tipo por defecto del error sea "any", es conveniente especificar el tipo de error esperado en función de quien lo genera.

- Los métodos de **HttpClient** devuelven como error un objeto de tipo **HttpErrorResponse**, con varias propiedades interesantes:
  - status: Valor del estado HTTP tras el error (400, 404, 500, etc)
  - url: URL que ha provocado el error
  - message: string = Texto descriptivo del error

Captura de errores por el Observable

- El operador "**catchError**" de Observable permite capturar y tratar el error en una función.
- La función recibe el error y debe devolver otro Observable, o bien un array vacío.
- Si el error se captura en el Observable, este ya no se lanza al suscriptor.

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

let obs = this.http.get('http://localhost/api/partidos');

obs.pipe(
   catchError(
        (error: HttpErrorResponse) => {
          console.log(error);
          return []; // No se va a emitir ningún valor más en el Observable
        }
   )
)
```

Esta estrategia implica no lanzar los errores al suscriptor, sino tratarlos antes de enviarlos, en el propio Observable.

Esto se realiza con el operador "catchError" de Observable.

El array vacío que se devuelve al tratar el error, significa que el Observable no va a emitir ningún elemento más.

Manejador global de errores

- Los errores no capturados ni tratados son finalmente capturados por un objeto de tipo **ErrorHandler**
- Angular suministra un *ErrorHandler* básico por defecto, que vuelca el error en consola.
- Si deseamos tratamiento personalizado de los errores necesitamos repetir código en múltiples lugares...
- ... O podemos crear un **manejador global personalizado**:
  1. Crear una clase que implemente la interfaz ErrorHandler
  2. Implementar la función "handleError()" a nuestro gusto
  3. Añadir un "provider" en AppModule, que provea nuestra clase cuando se necesite un ErrorHandler.
- A partir de ahí, todo error no tratado será manejado por nuestro método "handleError()"

Los errores que no son capturados por nadie, son finalmente capturados por un objeto especial de Angular, del tipo ErrorHandler.

Este manejador, lo único que hace es dejar rastro en la consola del error que se ha producido.

Por tanto, si queremos que los errores inesperados sean tratados de forma personalizada, debemos manejarlos bien en el suscriptor o en el Observable con catchError. Esto supone repetir mucho código similar.

Para evitar esto, podemos definir nuestro propio objeto de tipo ErrorHandler, que trate los errores inesperados de forma personalizada, pero sin tener que repetirlo en multiples partes del código.