Aplicación web

Es una aplicación cliente-servidor, donde el cliente es un browser y el browser interpreta HTML, JavaScript y CSS.

ASP.NET web api

ASP.NET Web API es un marco que facilita la creación de servicios HTTP disponibles para una amplia variedad de clientes, entre los que se incluyen exploradores y dispositivos móviles. ASP.NET Web API es la plataforma perfecta para crear aplicaciones RESTful en .NET Framework.

Ventajas:

Configuración: No existe mucha configuración necesaria para levantar un servicio de ASP.NET WebAPI. No hay endpoints ni contratos, y los servicios son bastante *holgados* en comparación con un servicio de WCF. Solo es necesaria una url REST, un conjunto de argumentos entrantes y un mensaje de respuesta.

REST por defecto: Principalmente debido a la funcionalidad de ruteo provista por el framework, provee casi todo para una arquitectura REST. Las direcciones de los servicios son rutas RESTful que se mapean con los métodos de los controladores, lo que se presta para realizar APIs REST.

Controller Activation: En cualquier framework que permite el desarrollo de aplicaciones orientadas a servicios, existe el concepto de *service activation*. En WCF o ASP.NET, cada llamada a un servicio es un nuevo pedido, por lo que el ambiente de ejecución activa una nueva instancia del servicio para cada una. Esto es similar a la instanciación de objetos en OO.

Interoperabilidad con JSON, XML y REST: Ya que REST está basado únicamente sobre estándares HTTP, es interoperable con cualquier plataforma capaz de realizar pedidos HTTP. Lo mismo aplica para JSON y XML.

Características

Acciones CRUD basadas en convención: Los verbos HTTP son mapeados automáticamente a los métodos de los controladores según sus nombres. Por ejemplo: si el controller se llama Products, un pedido GET a /api/products invocará automáticamente un método llamado Get en el controlador.

Negociación de Contenido incluida: En este framework, solo alcanza con retornar el tipo de datos crudo, y él se encarga de transformarlo en JSON o XML. Solo es necesario que el cliente utilice un encabezado HTTP Accept o Content-Type para especificar el tipo de contenido deseado para retornar los datos.

Ruteo por atributo y prefijos de ruta: A veces no es deseable seguir ruteos basados en convenciones. En esta nueva versión, es posible utilizar Route, RoutePrefix, y varios atributos Http para definir rutas explícitamente, lo que permite mejorar la manera de manejar las relaciones entre recursos.

Restricciones en rutas: Esto permite incluir las restricciones de los métodos de controllers a reglas de negocio. Por ejemplo, además de {id}, ahora es posible incluir {id:int} , {id:min(10)} , {id:range(1,100)} , o {tel:regex(^\d{3}-\d{3}-\d{3})}

Soporte de CORS El atributo EnableCors permite al desarrollador de la API permitir pedidos *cross-origin* de aplicaciones JavaScript de otros dominios.

Manejo global de errores Todas las excepciones no manejadas ahora pueden capturarse y manejarse desde un mecanismo central. El framework soporta múltiples loggers que tienen acceso a la excepción y el contexto en el que ocurrió.

IHttpActionResult Esta fábrica de la interface HttpResponseMessage provee una manera reusable y test-friendly de encapsular los resultados de los métodos de acción de la Web API. La respuesta creada fluye a través de un proceso de mensajes salientes, por lo que la negociación de contenido es respetada.

¿Por qué surgió ASP.NET?

Aunque durante la mayor parte de las últimas 3 décadas, los sitios web dependían de código del lado del servidor para cualquier cosa además de la manipulación de HTML, la llegada de JavaScript y Ajax han generado algunos cambios. Estas últimas generaron la necesidad de servicios que sean menos acerca de aplicaciones complejas comunicándose entre si y más acerca de páginas web necesitando obtener y enviar pequeñas cantidades de datos. Un ejemplo de este tipo de aplicaciones son las SPA (Single Page Applicactions).

Es aquí donde entran los frameworks para facilitar a creación de estos sitios. Desde el primer lanzamiento de .NET en 2002, MS proveyó varios caminos para construir aplicaciones orientadas a servicios.

API

Una API nos permite implementar las funciones y procedimientos que engloba nuestro proyecto sin la necesidad de programarlas de nuevo. En términos de programación, **es una capa de abstracción**.

Por ejemplo, digamos que estas desarrollando una aplicación web y necesitas hacer peticiones HTTP. En lugar de desarrollar todo el código para hacer una petición HTTP, puedes utilizar una API que se encargue de esto, como por ejemplo Requests de Python.

API REST: REST es una abreviatura de *Representational State Transfer*, o *Transferencia de Estado Representacional* y es un estilo de arquitectura para diseñar aplicaciones en red.

REST está centrada en los recursos, esto quiere decir que las APIs REST utilizan verbos HTTP para actuar sobre u obtener información de recursos. A estos recursos se los conoce como sustantivos.

Una API podría considerarse REST si su arquitectura se ajusta a ciertas reglas o restricciones. Las dos principales son:

Un **protocolo cliente/servidor sin estado**: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión.

Un conjunto de **operaciones bien definidas** que se aplican a todos los *recursos* de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son **POST**, **GET**, **PUT** y **DELETE**.

En conclusión, REST representa un tipo de arquitectura de interfaces de comunicación basado en un protocolo de cliente/servidor sin estado (protocolo HTTP), cacheable, con unas operaciones bien definidas en el que los recursos están identificados de forma única por URIs.

URIS y recursos

A los diferentes elementos de información los denominaremos “recursos”. Un recurso puede ser información sobre libros, clientes, coches, etc. pero un recurso nunca será una acción como por ejemplo “comprar_libro” o “crear_coche”.

Un identificador de recursos uniforme o **URI** —del inglés *uniform resource identifier*— es una cadena de caracteres que identifica los recursos de una red de forma unívoca. La diferencia respecto a un localizador de recursos uniforme (URL) es que estos últimos hacen referencia a recursos que, de forma general, pueden variar en el tiempo.

Reglas para asignar una URI:

- Deben ser únicas, no pudiendo existir más de una URI para identificar un mismo recurso.
- Deben ser independientes del formato en el que queramos consultar el recurso
- Deben mantener una jerarquía en la ruta del recurso
- No deben indicar acciones, por lo que no debemos usar verbos a la hora de definir una URI

Si queremos añadir funcionalidad de filtrado a la hora de obtener un listado de recursos, no debemos definir otra URI especial ni añadir nuevos elementos en la ruta de la URI, sino que podemos usar los parámetros de consulta de la query, que son una serie de pares (clave, valor) separados por el carácter “&” y que se sitúan en la URI tras el símbolo “?”.

Un ejemplo incorrecto a la hora de filtrar clientes sería:

- GET http://miapideejemplo.com/clientes/ciudad/Huelva

El ejemplo correcto sería:

- GET http://miapideejemplo.com/clientes?ciudad=Huelva

Cómo ya establecimos, crear una interfaz RESTful significa terminar con una API céntrica al recurso. Por esto, se debe diseñar la interfaz con los recursos en el centro. Las acciones disponibles para esos recursos están restringidas por el uso de HTTP. Por lo tanto, es necesario mapear los verbos HTTP a la API, y no agregar otras acciones o verbos.

Tenemos dos tipos de URIs: **URIs de colección** y **URIs individuales**. Una URI de colección sería /api/tasks, mientras que la URI individual sería /api/tasks/1234. La sección de /api es la ruta para acceder a la api de la aplicación. tasks es la uri al recurso de Tareas, y si se coloca un id (en este caso 1234), se accede al recurso con ese id.

Ruteo de atributos

Routing es la manera en que Web API conecta una URI a una acción de un controlador.

En Web API 2 se introduce lo que se conoce como ***Attribute Routing*** que, como indica su nombre, utiliza los atributos para definir rutas. Esto brinda más control sobre las URIs. Para activar *Attribute Routing*, hay que llamar a MapHttpAttributeRoutes durante la configuración de la aplicación.

```
Route("/books/{bookId}/authors")]
public IHttpActionResult GetBookAuthors(Guid bookId) { ..... }
```

También existe el ***Convention Routing***, que se basa en definir templates de rutas (strings parametrizados), para que luego el framework empareje una URI con el template y decidir qué acción ejecutar. Sin embargo, este tipo de routing dificultaba utilizar algunos patrones comunes en APIs REST, como recursos con subrecursos.

Ejemplo: Los autores de un libro.

```
Config.Routes.MapHttpRoute(
name: "DefaultApi",
routeTemplate: "api/{Controller}/{id}",
defaults: new { id = RouteParameter.Optional }
);

public IHttpActionResult Get(Guid bookId) { ..... }
```

Verbos HTTP

POST, PUT, GET, DELETE.

Se dice que los verbos GET, PUT y DELETE son idempotentes, ya que no importa cuántas veces se ejecute el pedido el resultado será siempre el mismo. Por ejemplo, si el cliente envía un pedido DELETE sobre un recurso, aunque el recurso ya haya sido borrado, no debería recibir errores. Es importante mantener la idempotencia de los GET, PUT y DELETE consistente.

A diferencia de las otras, POST no es considerada Idempotente. Esto se debe a que este verbo se utiliza para crear una nueva instancia de un recurso por cada invocación del método.

Los métodos por defecto de un controlador en la WebApi son Get, Get(id), Post, Put y Delete.

Tipos de respuesta de los controladores

Un *controlador* es un objeto que maneja los pedidos HTTP y hereda de la clase ApiController.

- HttpResponseMessage: Convierte directamente a un mensaje respuesta HTTP. Esta opción brinda mucho control sobre el mensaje de respuesta, permitiendo ingresar headers particulares, o modificar el formato del contenido.
- IHttpActionResult: Llama a ExecuteAsync para crear la HttpResponseMessage, y luego convertirlo en mensaje de respuesta HTTP. Provee más flexibilidad y hace que la intención de nuestro controller sea mucho más clara, escondiendo los detalles a bajo nivel de construcción de la respuesta.
- Void: retorna código 204 (sin contenido)
- Cualquier otro tipo: escribe el valor de retorno serializado dentro de una respuesta 200 (OK)

Mocking

Los mocks son unos de las varios "test doubles" (es decir, objetos que no son reales respecto a nuestro dominio, y que se usan con finalidades de testing) que existen para probar nuestros sistemas. Los más conocidos son los Mocks y los Stubs, siendo la principal diferencia en ellos, el foco de lo que se está testeando.

Estos dos tipos de pruebas consisten en crear objetos falsos que sustituyan a los objetos reales y de esta manera poder probar un determinado requerimiento. El objetivo de utilizar objetos fake en lugar de reales, es romper las dependencias con otros objetos y de esta manera probar requerimientos de manera independiente.

Antes de hacer énfasis en la diferencia entre Mocks y Stubs, es importante aclarar que nos referiremos a la sección del sistema a probar como SUT -System under test-. Los Mocks, nos permiten verificar la interacción del SUT con sus dependencias. Los Stubs, nos permiten verificar el estado de los objetos que se pasan. Como queremos testear el comportamiento de nuestro código, utilizaremos los primeros. Resumiendo, con el Mock probaremos que desde este método se hace la llamada al servicio correctamente; y con el Stub probaremos que el resultado es el esperado.

Los usamos porque queremos probar objetos y la forma en que estos interactúan con otros objetos. Para ello crearemos instancias de Mocks, es decir, objetos que simulen el comportamiento externo (es decir, la interfaz), de un cierto objeto. Son objetos tontos, que no dependen de nadie, siendo útiles para aislar una cierta parte de la aplicación que queramos probar.

Patrón MVC

Es un patrón que permite la separación de responsabilidades entre los elementos de una estructura de software. Establece que se contará con tres categorías de elementos:

- Modelo: Representado por las clases de dominio, las entidades.
- Vistas: La representación del modelo.
- Controlador: El nexo entre ambas.

Inyección de dependencias

Es un patrón orientado a objetos en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree el objeto.

Inyectar dependencias es entonces pasarle la referencia de un objeto a un cliente, al objeto dependiente (el que tiene la dependencia). Significa simplemente que la dependencia es encajada/empujada en la clase desde afuera. Esto significa que no debemos instanciar (hacer new), dependencias, dentro de la clase.

Esto lo podemos hacer, por ejemplo, pasando al objeto cliente por parámetro al constructor del objeto que tiene la dependencia.

Ventajas de ID:

- Código más limpio. El código es más fácil de leer y de usar.
- Más fácil de Testear.
- Más fácil de modificar. Nuestros módulos son flexibles a usar otras implementaciones. Desacoplamos nuestras capas.
- Permite NO Violar Single Responsability Principle. Ahora nuestra lógica de creación de objetos no va a estar relacionada a la lógica de cada módulo. Cada módulo solo usa sus dependencias, no se encarga de inicializarlas ni conocer cada una de forma particular. (No se hace un new de otra clase en el constructor)
- Permite NO Violar Open Close Principle. Por todo lo anterior, nuestro código es abierto a la extensión y cerrado a la modificación. El acoplamiento entre módulos o clases es siempre a nivel de interfaz.
- Permite NO violar Dependency Inversion Principle. Ya que no es posible invertir las dependencias si tenemos new en el constructor de otra clase. DIP permite conseguir un desacoplamiento de las clases en el código, de tal manera que, si una clase emplea otras clases, la inicialización de los objetos venga dada desde fuera.

Reflection

Reflection es la habilidad de un programa de autoexaminarse con el objetivo de encontrar ensamblados (.dll), módulos, o información de tipos en tiempo de ejecución. En otras palabras, a nivel de código vamos a tener clases y objetos, que nos van a permitir referenciar a ensamblados, y a los tipos que se encuentran contenidos.

Se dice que un programa se refleja en sí mismo (de ahí el término "reflexión"), a partir de extraer metadata de sus assemblies y de usar esa metadata para ciertos fines. Ya sea para informarle al usuario o para modificar su comportamiento.

Al usar Reflection en C#, estamos pudiendo obtener la información detallada de un objeto, sus métodos, e incluso crear objetos e invocar sus métodos en tiempo de ejecución, sin haber tenido que realizar una referencia al ensamblado que contiene la clase y a su namespace.

Específicamente lo que nos permite usar Reflection es el namespace System.Reflection, que contiene clases e interfaces que nos permiten manejar todo lo mencionado anteriormente: ensamblados, tipos, métodos, campos, crear objetos, invocar métodos, etc.

Para usar Reflection es importante que nuestro código referencie a una Interfaz, que es la que toda .dll externa va a tener que cumplir. Tiene que existir entonces ese contrato previo, de lo contrario, no sería posible saber de antemano qué métodos llamar de las librerías externas que poseen clases para usar loggers.

Ejemplo de uso:

Supongamos que necesitamos que nuestra aplicación soporte diferentes tipos de loggers (mecanismos para registrar datos/eventos que van ocurriendo en el flujo del programa). Además, supongamos que hay desarrolladores terceros que nos brindan una .dll externa que escribe información de logger y la envía a un servidor. En ese caso, tenemos dos opciones:

1. Podemos referenciar al ensamblado directamente y llamar a sus métodos (como hemos hecho siempre)
2. Podemos usar Reflection para cargar el ensamblado y llamar a sus métodos a partir de sus interfaces.

En este caso, si quisiéramos que nuestra aplicación sea lo más desacoplada posible, de manera que otros loggers puedan ser agregados de forma sencilla y SIN RECOMPILAR la aplicación, es necesario elegir la segunda opción.

Por ejemplo, podríamos hacer que el usuario elija (a medida que está usando la aplicación), y descargue la .dll de logger para elegir usarla en la aplicación. La única forma de hacer esto es a partir de Reflection. De esta forma, podemos cargar ensamblados externos a nuestra aplicación, y cargar sus tipos en tiempo de ejecución.

Código ejemplo:

```
(1) Assembly assembly = Assembly.LoadFile(dll);
(2) foreach (Type type in assembly.GetTypes())
{
    (3) Object instancia = Activator.CreateInstance(type, new Object[]{ parámetros  });
    (4) List<Type> interfaces = type.GetInterfaces().ToList();
    (5) Type checkInterface = interfaces.Find(x => x.Name == "IImportProducts");}
```

1- Levanta la dll con todos sus tipos dentro.
2- Para obtener todos los tipos que podemos instanciar es con GetTypes
3- Para crear una instancia es con el CreateIstance en donde le tenemos que pasar el type.
4- Para obtener que interfaces implementa un tipo es con GetInterfaces
5- Buscar una interface dentro de las que implementa.

Single Page App

Las *SPAs*, si bien interactuan de la forma cliente-servidor, toman un enfoque diferente. En la petición inicial, un HTML inicial se envía al browser, pero las interacciones del usuario generan requests a través de **AJAX** para pequeños fragmentos de HTML/datos que se insertan en lo que se le muestra al usuario.

El documento HTML inicial nunca se recarga, y el usuario puede seguir intercalando con el HTML existente mientras las requests ajax terminan de ejecutarse asincrónicamente.

Características de las SPAs:

- Este tipo de Web Apps es conocida porque tienen la posibilidad de redibujar la UI sin tener que realizar una nueva petición (Round-Trip) al servidor.
- Mejoran la UI por el hecho de que los usuarios tienen una experiencia ininterrumpida, sin que la página se refresque, y sin agregar complejidad.
- Son ideales para el mundo tanto web y mobile: no se agrega complejidad desde el lado del servidor para poder servir a diferentes dispositivos o plataformas. La lógica de lograr que nuestras web apps sean "responsive" siempre va desde el lado del cliente (browser), no se cargan nuevas páginas todo el tiempo.
- Estan 100% alineadas al concepto de las APIs REST, debido a que estas simplemente exponen puntos para transaccionar y/o recibir o devolver datos, de una forma totalmente separada de la forma en que se van a mostrar.

Angular

La meta de angular es traer las herramientas y capacidades que han estado disponibles para el desarrollo de backend al cliente web, facilitando el desarrollo, test y mantenimiento de aplicaciones web complejas y ricas en contenido.

Angular hace que nuestro HTML sea más expresivo, permitiéndole embeber/agregar features y lógica al HTML para lograr un data-binding con nuestros modelos. Esto nos permite mostrar campos que tengan valores de nuestros modelos/datos de forma sencilla, y tener un seguimiento de los mismos (actualización en tiempo real).

Promueve la modularidad desde su diseño, siendo fácil crear y lograr reuso de los componentes y del contenido.

Angular a su vez tiene soporte ya incluido para comunicación con servicios de backend (es fácil que nuestras webs apps se conecten a nuestros backends y ejecuten lógica del lado del servidor).

En Angular, una aplicación se define a partir de un conjunto de componentes, del mismo modo que también de servicios subyacentes que son comunes a ellos y permiten el reuso de la lógica.

Un **componente** es una unidad modularizada que define la vista y la lógica para controlar una porción de una pantalla en Angular. Cada componente se compone de:

1. Un **template** (que es el HTML para la UI, también llamado la View). Sin los datos, por eso un template. Los datos serán inyectados de forma dinámico.
2. Una **clase que es el código asociado a la View**, teniendo properties/datos que están disponibles para el uso de las Views, y métodos que son lógica o acciones para dichas views. Por ejemplo: responder a un click de un botón, o a un evento.
3. **Metadata,** la cual provee información adicional del componente a Angular. Es lo que identifica a la clase asociada al componente.

Los componentes se agrupan en **módulos** o Angular Modules. Estos nos permiten organizar nuestros componentes en funcionalidad cohesiva. Los modulos en Angular son clases anotadas con el decorator @NgModule

Los **decorators** son simplemente funciones que van a modificar nuestras clases de JavaScript. Angular define un montón de decoradores que agregan metadata a las clases que vayamos definiendo, de manera que podamos agregarle funcionalidad extra a nuestras clases.

TypeScript

Es un lenguaje que debe ser transpilado ya que es basado en ES2015. Esto significa que podemos usar todas las features que ofrece ES2015 y luego esto será transpilado a otra sintaxis para que el browser lo ejecute, pero nosotros no debemos abstraernos a lo que el navegador soporta.

Ejecución de Angular

Al arrancar nuestra app usamos el comando `npm start`. Esto lo que hace es levantar un Web Server para que nuestro navegador pueda consumir los archivos desde ahí, es simplemente un ambiente local que funciona como un ambiente real.

A su vez, vemos como se ejecuta tsc, el TypeScript Compiler, el cual compila nuestros .ts y los transpila a .js. Como vemos en el VisualStudioCode, vemos que por cada `.ts` que nosotros tenemos, se nos crean un `.js` y un `.js.map`, que fueron generados por el typescript compiler, que transpiró todos nuestros .ts para que el browser pueda entenderlos.

También vemos que el TypeScript compiler y el FileServer que levantamos, “watchea” cualquier cambio a nivel de código, de manera que cada vez que hacemos un cambio en nuestro código, TypeScript lo recompila y podemos ver los cambios en el navegador.

Data binding

Data binding es un mecanismo mediante el cual podemos enlazar los elementos de la interfaz de usuario con los objetos que contienen la información a mostrar. Por ejemplo, en nuestra clase de logica podemos tener una variable pageTitle de manera que cuando el HTML se renderiza, el HTML muestra el valor asociado al modelo pageTitle.

El Data Binding va de la mano del concepto de **interpolación**, la cual es la habilidad de poner datos dentro de un HTML (interpolar). Esto es lo que logramos con las llaves dobles {{pageTitle}}.

Property Binding: cuando el binding es hacia una property particular y no a una expresión compleja como puede ser la interpolación. Setea el valor de una property a un a expresíon en el template. Por ejemplo: <img [src]='imagen.jpg'>

Event Binding: es binding hacia funciones o métodos que se ejecutan como consecuencia de eventos. Por ejemplo: <button (click)='showImage()'>

Two-Way Binding: Es un ida y vuelta entre el template y una property entre un component. Muestra el valor de la property en la vista, y si en la vista/template dicho valor cambia, la property también se ve reflejada (por eso es de dos caminos). Por ejemplo: <input [(ngModel)]='listFilter' />

Directivas

A su vez, también podemos enriquecer nuestro HTML a partir de lo que se llaman **directivas**, pudiendo agregar **ifs** o **loops** (estilo for), sobre datos en nuestro HTML y generar contenido dinámicamente.

Pipes

Cuando los datos no están en el formato apropiado que queremos para mostrarlos, usamos Pipes. Estos permiten aplicar cierta lógica sobre las properties de nuestra clase antes de que estas sean mostradas (por ejemplo, para que sean más user friendly). Angular ya provee varios Pipes built-in para diferentes tipos de datos (date, number, decimal,json, etc), e incluso permite crear pipes propios para realizar lógica particular como lo es manejar el filtrado.

`Ejemplo:` {{username | uppercase}}

Life hooks

Todo componente de Angular, tiene un ciclo de vida. Dicho ciclo de vida (lifecycle) es gestionado mismo por Angular:

1. Angular **crea** los componentes.
2. Luego **renderiza** los componentes.
3. Una vez esto, Angular **crea y renderiza** los componentes hijos.

4. Durante la operativa normal, Angular **procesa los cambios** sobre un componente cuando sus "properties enlazadas" (data bound properties) cambian.
5. Y finalmente **destruye** el componente.

Angular provee un conjunto de *lifecycle hooks* o simplemente 'ganchos' para que los desarrolladores de componentes podamos hacer algo en cada etapa del ciclo de vida, cuando lo deseemos. Esto nos permite realizar operaciones a demanda.

OnInit: Inicialización del componente. Es un buen lugar para obtener los datos del backend.

OnChanges: Ejecutar acciones después del cambio de input properties.

OnDestroy: Para hacer un limpiado de los recursos que usa el componente (un CleanUp) antes de que el componente muera.

Para usar un *lifecycle hook*, necesitamos hacer que nuestro componente implemente la interfaz existente para el gancho que se desee usar. Haremos una implementación de estas interfaces (donde cada interfaz tiene simplemente un método), y le daremos nuestra lógica particular.

Cada hook/interfaz define un método que es `ngNombreHook`. Por ejemplo, la interfaz `OnInit` define el hook `ngOnInit`.

Servicios

Para conectarnos con nuestra API construiremos **servicios**. Y a su vez, usaremos **inyección de dependencias** para poder meter/inyectar esos servicios en dichos componentes.

Por ejemplo, así inyectamos el servicio en el constructor de una clase:

```
constructor(private _petsService : PetService) {
 // esta forma de escribir el parametro en el constructor lo que hace es:
 // 1) declara un parametro de tipo PetService en el constructor
 // 2) declara un atributo de clase privado llamado _petService
 // 3) asigna el valor del parámetro al atributo de la clase
}
```

Angular trae un `Injector` *built-in*, que nos permitirá registrar nuestros servicios en nuestros componentes, y que estos sean Singleton. Este Injector funciona en base a un contenedor de inyección de dependencias, donde una vez estos se registran, se mantiene una única instancia de cada uno.

Routing

Como sabemos, nuestra app en Angular es una SPA, lo que nos permitirá tener decenas o cientos de páginas manejando un solo HTML (`index.html`); las diferentes vistas se "van turnando" entre sí para mostrarse en los momentos adecuados.

La idea es que configuremos la **ruta** o *route* para que dicha vista aparezca. Como parte del diseño de nuestra app, vamos a estar usando botones, toolbars, links, imagenes, o lo que sea que queramos usar para disparar acciones y cambiar de página. Estas acciones, tendrán una ruta asociada, permitiendo que cuando dicha acción se dispare, el componente de la acción asociada se muestre.

Ejemplo:

En nuestro menú principal, tenemos una serie de opciones donde cada una de ellas tendrá atada una ruta sobre la cual queremos rutear.

<a routerLink='/home'>Welcome

Cuando el usuario cliquee en dicho link, la ubicación del browser cambiará a: `localhost:3000/home`.

Cuando dicha ubicación cambia, el Angular Router buscará una definición de ruta que se matchee con dicha nueva ruta. Dicha definición de ruta incluirá el componente a cargar cuando tal ruta sea activada:

{ path: 'home', component: HomeComponent}

Observables

Flujo de acción de una petición HTTP:

1. La aplicación envía una request a un servidor/web service (HTTP GET http://page.com/api/products/2)
2. Ese Web Service obtiene los datos, seguramente accediendo a una base de datos.
3. El Web Service le contesta a la aplicación, con los datos obtenidos, en forma de una HTTP Response.
4. La aplicación procesa entonces los datos (por ej: los muestra en una View).

Los Observables nos permiten manejar datos asincrónicos, como los datos que vendrán de nuestro *backend* o de algún *web service*. Los mismos tratan a los eventos como una colección; podemos pensar a un Observable como un array de elementos que van llegando asincrónicamente a medida que pasa el tiempo.

Una vez tengamos nuestros Observables, un método en nuestro código puede suscribirse a un Observable, para recibir notificaciones asincrónicas a medida que nuevos datos vayan llegando. Dicho método puede entonces "reaccionar", sobre esos datos. Dicho método a su vez es notificado cuando no hay más datos, o cuando un error ocurre.

Ejemplo de código:

```
// some.service.ts

@Injectable()
export class SomeService {
      getSomeObservable(): Observable<any> {
            // retorna un observable
      }
}

// some.component.ts

import { SomeService } from './some.service';
export class SomeComponent {
      constructor(private _someService : SomeService) {
      }

      ngOnInit(): void {
            this.someService.getSomeObservable().subscribe(
                  ((data : any) => this.doSomethingWitData(data),
                  ((error : any) => this.handleError(error))
            )
      }
}
```

Observables vs Promises:

Otra forma bastante común de obtener datos a través de HTTP es usando **promises**. Las promises/promesas son objetos de JavaScript que sirven para hacer computación asincrónica, representando un cierto valor que puede estar ahora, en el futuro o nunca.

Tanto Observables como Promises sirven para lo mismo, pero los Observables permiten hacer más cosas:

- Los Observables permiten cancelar la suscripción, mientras que las Promises no.
- Las Promises manejan un único evento, cuando una operación asincrónica completa o falla. Los Observables son como los Stream (en muchos lenguajes), y permiten pasar cero o más eventos donde el callback será llamado para cada evento.
- En general, se suelen usar Observables porque permiten hacer lo mismo que las Promises y más.
- Los Observables proveen operadores como *map*, *forEach*, *reduce*, similares a un array.

Frameworks

Es una aplicación reusable y semicompleta que se puede especializar para hacer aplicaciones más específicas.

Es un esqueleto de aplicación adaptable por el desarrollador.

Se pueden clasificar según alcance:

- De infraestructura: servicios de bajo nivel como comunicaciones y acceso a APIs del sistema operativo.
- "Middleware" de operación: permiten integrar componentes y sistemas.
- De aplicación en dominios particulares: seguros, finanzas, control industrial, etc.

Y se pueden clasificar según diseño:

- Caja Negra: definen un conjunto de interfaces que permiten asociar el framework a componentes intercambiables que deben respetar determinada interfaz. Son fáciles de usar, pero difíciles de implementar.
- Caja Blanca: son extensibles mediante la utilización de herencia y polimorfismo (Template Method). El usuario del framework define sub clases y redefine determinados métodos. Son flexibles y adaptables, pero requieren que el usuario tenga conocimientos de la implementación del framework.

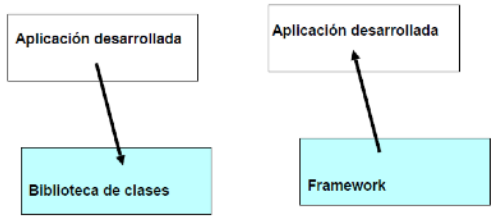
Los frameworks ayudan a reducir el tiempo y costo de desarrollo, disminuir el número de defectos en las clases y contar con diseños más reusables.

Sin embargo, tienen algunos problemas. Son difíciles de entender (y a veces es necesario conocer los mecanismos internos del framework), difíciles de desarrollar (es muy difícil de evolucionar ya que los cambios impactan mucho en las aplicaciones que lo usan) y difíciles de combinar. La falta de estándares conocidos para diseñar, desarrollar, documentar y actualizar frameworks también es un obstáculo grande.

Frameworks vs Bibliotecas de clase

Las bibliotecas de clase son independientes del dominio, mientras que los frameworks son desarrollados para determinado dominio o problemas particulares.

El uso de la biblioteca de clases se caracteriza por que el flujo de control es en un solo sentido: de la aplicación a la biblioteca. Mientras que en los frameworks existe la inversión de control, que significa que el flujo de control es principalmente en el sentido Framework – aplicación.



Normalmente un framework tiene asociado una o más bibliotecas de clase que proveen componentes utilizados por el framework.

Frameworks y patrones

En el diseño de un framework se utilizan patrones para diseñar y documentar el framework.

Principales elementos utilizados en el diseño y desarrollo de frameworks:

Conceptos:

- Herencia
- Polimorfismo
- Clases abstractas
- Composición y delegación

Análisis, diseño y método:

- Patrones
- Análisis de dominio
- Refactoring
- Documentación de frameworks (orientada tanto a usuarios del framework como a desarrolladores)
- Uso de frameworks

Desarrollo de frameworks

Hot Spot: son aquellos aspectos de un dominio de aplicación que deben mantenerse flexibles. Determinan aquellos lugares donde el framework debe permitir adaptación.

Frozen Spots: son aquellos aspectos del framework que no se diseñaron para permitir adaptación.

Ambos son importantes ya que de la buena definición depende la utilidad y flexibilidad del Framework.

En la implementación de frameworks se distinguen los diferentes tipos de métodos:

- Template: implementan los Frozen Spots.
- Hook: implementan los Hot Spots. Pueden ser métodos abstractos, comunes, template methods.