DPTO. INFORMÁTICA - I.E.S. LA MARISMA
MÓDULO PROYECTO
C.F.G.S.
DESARROLLO DE APLICACIONES WEB

Aplicación FoodLack

Autor/es: Jesica Beltrán Cordero
Fecha: 16/06/2020
Tutor: Javier Alonso

Indice

1.Introducción

1.1 Descripción general

Con la llegada de la pandemia mundial provocada por el virus Covid-19, hemos sufrido numerosas consecuencias, como la falta de abastecimiento en los supermercados debido a la alarma social creada en su inicio. Por ello esta aplicación es una herramienta que nos puede ayudar a la hora de hacer una compra de alimentos en este tipo de situaciones en las que lo primordial es no perder el tiempo en la búsqueda de esos alimentos. Así pues, con esta aplicación el usuario puede consultar en su localidad la falta de algún producto que desee comprar en su habitual supermercado.

Así mismo quien crea las notificaciones, donde se puede ver el producto agotado y el supermercado al que pertenece, es el mismo usuario que tendrá la posibilidad de notificarlo, para facilitar las consultas de los demás usuarios.

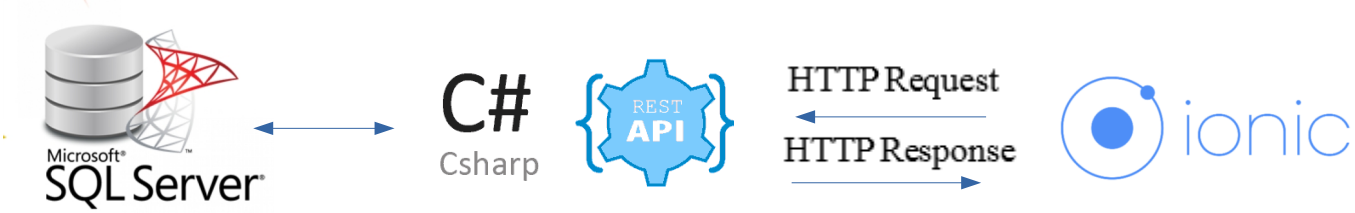
1.1.1 Objetivos generales

El objetivo principal es la reducción del tiempo del usuario al realizar su compra, ya que es muy importante estar el menor tiempo posible fuera de casa en pleno confinamiento.

Otro objetivo es la comodidad del usuario al consultar la disponibilidad del producto para evitar movilizaciones innecesarias.

1.2 Descripción general a nivel de desarrollo

FoodLack es una aplicación realizada con la versión actual del framework .NET Core 3.1 de Microsoft escrito en C#. Con ello he creado una API REST para realizar las peticiones HTTP pertinentes a la base de datos, en este caso a SQLServer. El cliente encargado de hacer las peticiones a la API está desarrollado en Ionic.



Está organizado en una arquitectura por capas, cada capa está desacoplada una de la otra, es decir, cada una tiene sus propias responsabilidades. Esto conlleva a una mejor organización del proyecto y se convierte en un código más legible. Todo esto se explicará con mas detenimiento a lo largo de esta documentación.

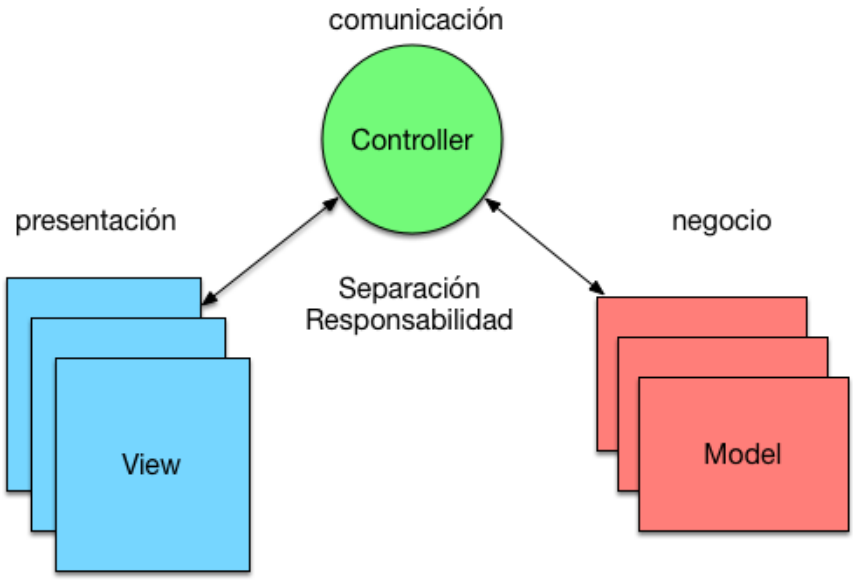
2.Arquitectura del proyecto

2.1.Arquitectura Modelo-vista-Controlador

MVC es un patrón en el diseño de software comúnmente utilizado para implementar interfaces de usuario, datos y lógica de control. Enfatiza una separación entre la lógica de negocios y su visualización. Esta "separación de preocupaciones" proporciona una mejor división del trabajo y una mejora de mantenimiento.

Las tres partes del patrón de diseño de software MVC se pueden describir de la siguiente manera:

- Modelo: Maneja datos y lógica de negocios.
- Vista: Se encarga del diseño y presentación.
- Controlador: Enruta comandos a los modelos y vistas.



Principales ventajas:

- La separación del Modelo de la Vista, es decir, separar los datos de la representación visual de los mismos.
- Es mucho más sencillo agregar múltiples representaciones de los mismos datos o información.
- Facilita agregar nuevos tipos de datos según sea requerido por la aplicación ya que son independientes del funcionamiento de las otras capas.
- Crear independencia de funcionamiento.
- Facilita el mantenimiento en caso de errores.
- Ofrece maneras más sencillas para probar el correcto funcionamiento del sistema.
- Permite el escalamiento de la aplicación en caso de ser requerido.

2.2.Arquitectura en capas

La arquitectura de una aplicación es la vista conceptual de la estructura de esta. Toda aplicación contiene código de presentación, código de procesamiento de datos y código de almacenamiento de datos. La arquitectura de las aplicaciones difieren según como esta distribuido este código.

Lo que se consigue con esta arquitectura es dividir el sistema en partes diferenciadas, de tal forma que cada capa solo se comunique con la inferior.

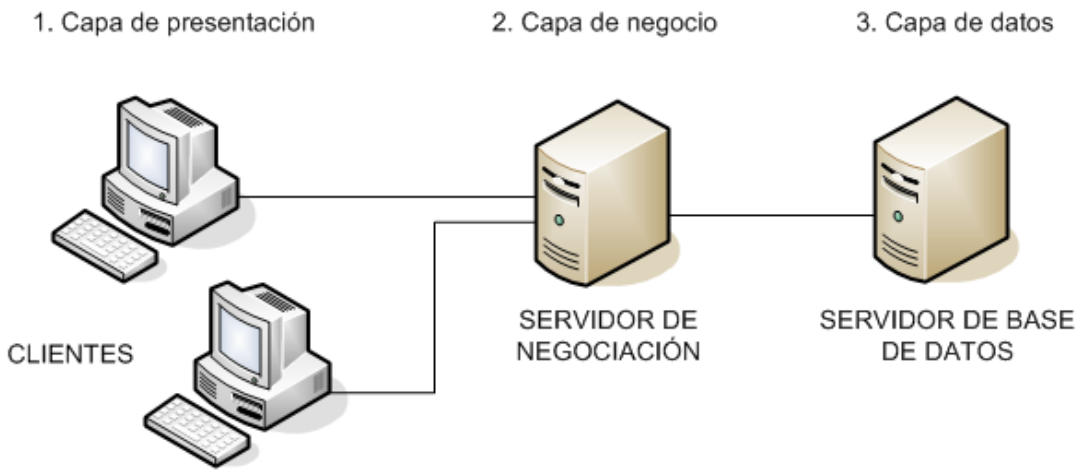
La mayor ventaja de este patrón de arquitectura es que en el caso de que exista algún error o la necesidad de algún cambio, solo es necesario cambiar el nivel en cuestión, sin afectar el correcto funcionamiento del resto del sistema. Se simplifica la comprensión del desarrollo del proyecto.

Estas capas se denominan:

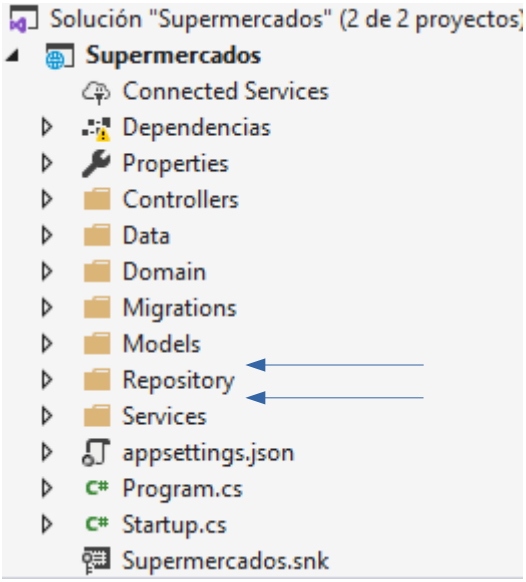
Capa de persistencia: Esta capa se encarga de guardar los datos. Es la única capa que tiene comunicación directa con la base de datos, por lo que gestiona todo lo relativo a ella, su creación y modificación.

Capa de negocio: En esta capa se gestiona la parte lógica de la aplicación. Es la que realiza las operaciones pertinentes con los datos que recoge de la capa de persistencia.

Capa de presentación: Esta capa es la que crea la interfaz del usuario. Su función es pasar las acciones que realice el usuario a la capa de negocio.



Este proyecto está organizado con una arquitectura por capas de la siguiente manera:



La capa de **persistencia** en este caso es nuestro repositorio, carpeta "Repository", donde almacenamos los datos recogidos de la base de datos y así modificarlas.

La capa de **negocio** es nuestra capa encargada de la lógica y operaciones de la aplicación, en este caso se trata de la carpeta "Services".

La capa de **presentación** es la parte de la interfaz, que conseguimos crear con Ionic.

En resumen, esta arquitectura conlleva muchas ventajas para el desarrollo de la aplicación:

- Reutilización de capas.
- Facilita la estandarización.
- Dependencias se limitan a intra-capa.
- Contención de cambios a una o pocas capas.
- Más seguridad y versatilidad al sistema, porque con objetos es mas fácil hacer crecer la aplicación.
- Es más ordenado.
- Clara distribución de las responsabilidades.
- Es más fácil trabajar en equipo con otros desarrolladores y hasta armar equipos de desarrolladores para cada capa.
- Podemos cambiar de repositorio de datos sin impacto en el resto de la aplicación.

Una vez que sabemos cómo va a ir organizado nuestra aplicación pasaremos a crear nuestra API REST en C#.

3.Desarrollo en la capa servidor

3.2.Creación de API REST con ASP .NET CORE

3.2.1.¿Qué es una API REST?

El término REST (Representational State Transfer) se originó en el año 2000, descrito en la tesis de Roy Fielding, padre de la especificación HTTP. Un servicio REST no es una arquitectura software, sino un conjunto de restricciones con las que podemos crear un estilo de arquitectura software, la cual podremos usar para crear aplicaciones web respetando HTTP.

Hoy en día la mayoría de las empresas utilizan API REST para crear servicios. Esto se debe a que es un estándar lógico y eficiente para la creación de servicios web.

Las principales características que define a una API REST son:

Cliente-servidor: esta restricción mantiene al cliente y al servidor débilmente acoplados. Esto quiere decir que el cliente no necesita conocer los detalles de implementación del servidor y el servidor se "despreocupa" de cómo son usados los datos que envía al cliente. Son dos proyectos separados pero que se comunican entre ellos.

Sin estado: aquí decimos que cada petición que recibe el servidor debería ser independiente, es decir, no es necesario mantener sesiones.

Cacheable: debe admitir un sistema de almacenamiento en caché. La infraestructura de red debe soportar una caché de varios niveles. Este almacenamiento evitará repetir varias conexiones entre el servidor y el cliente para recuperar un mismo recurso.

Interfaz uniforme: define una interfaz genérica para administrar cada interacción que se produzca entre el cliente y el servidor de manera uniforme, lo cual simplifica y separa la arquitectura. Esta restricción indica que cada recurso del servicio REST debe tener una única dirección, "URI".

Sistema de capas: el servidor puede disponer de varias capas para su implementación. Esto ayuda a mejorar la escalabilidad, el rendimiento y la seguridad.

Las operaciones más importantes que nos permitirán manipular los recursos son cuatro: **GET** para consultar y leer, **POST** para crear, **PUT** para editar y **DELETE** para eliminar.

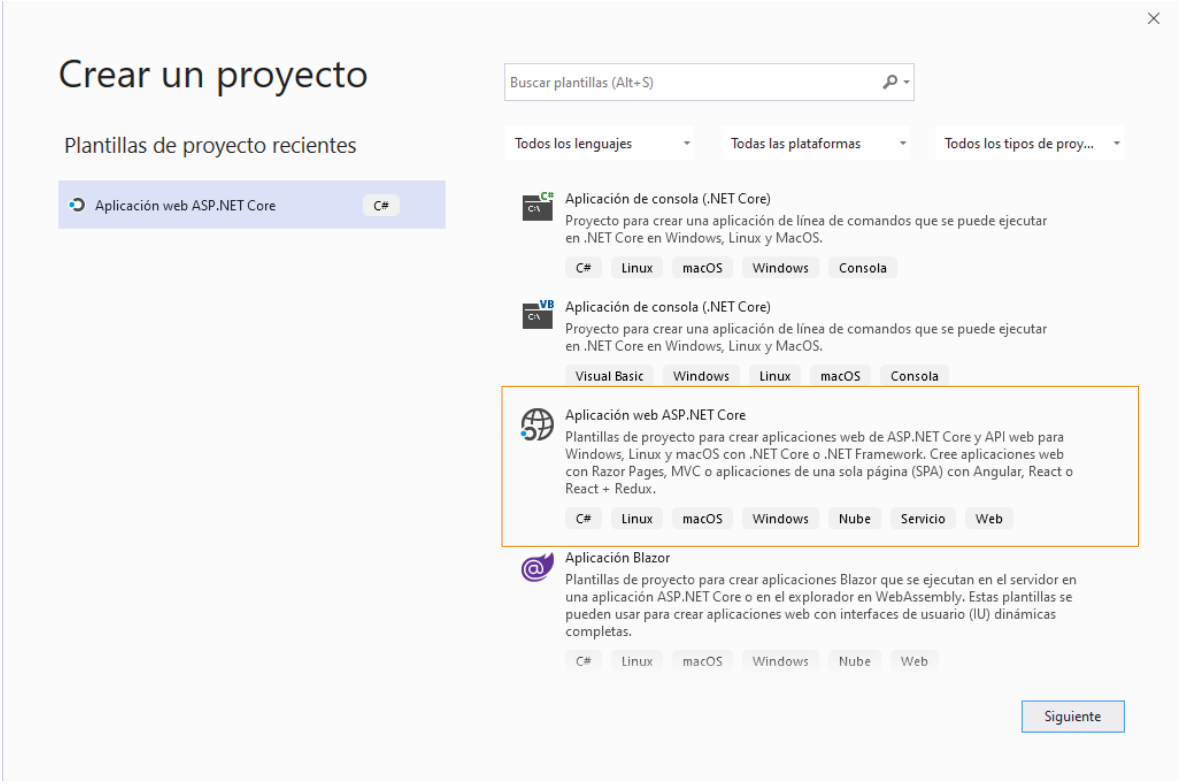
Otra característica es el uso de **hipermedios** (término que en el ámbito de las páginas web define el conjunto de procedimientos para crear contenidos que contengan texto, imagen, vídeo, audio y otros métodos de información) para permitir al usuario navegar por los distintos recursos de una API REST a través de enlaces HTML.

Para terminar, comentar que lo más importante a tener en cuenta al crear nuestro servicio o API REST no es el lenguaje en el que se implemente sino que las respuestas a las peticiones se hagan en XML o JSON, ya que es el lenguaje de intercambio de información más usado. En este proyecto trabajaremos con el intercambio de datos en JSON.

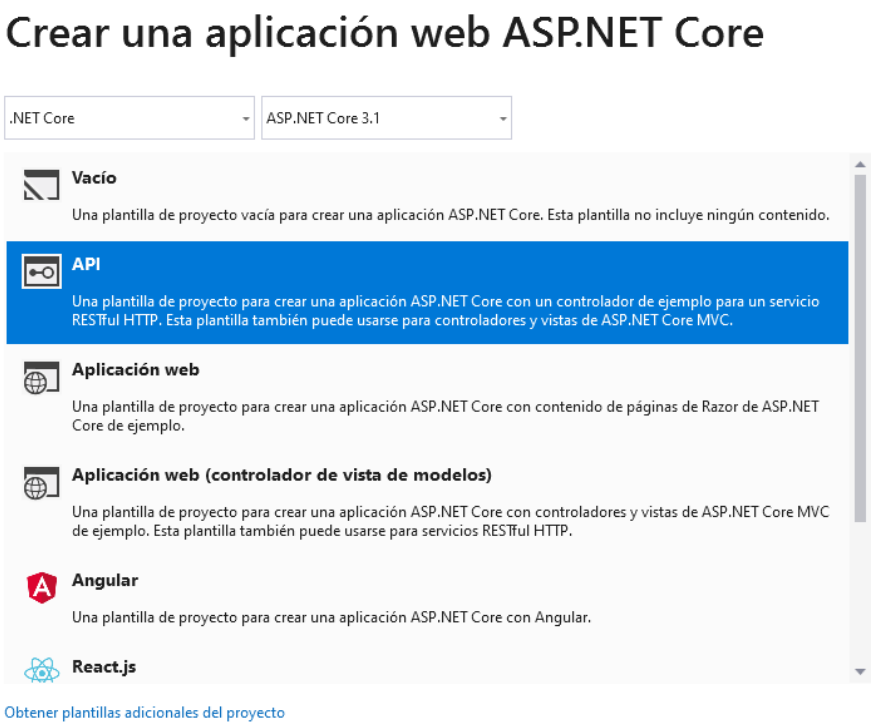
3.2.2.Creación de proyecto WEB API

Para comenzar he utilizado el entorno Visual Studio 2019. Puede descargarse desde su página oficial https://visualstudio.microsoft.com/es/downloads/ .

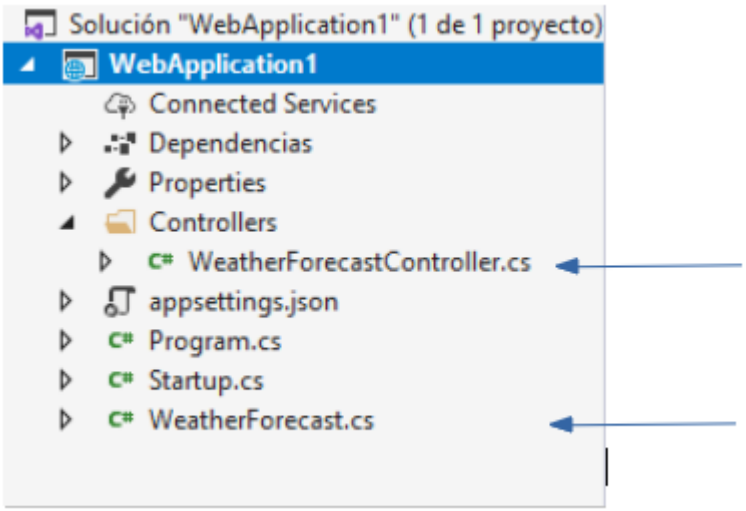
Una vez que lo tenemos instalado procedemos a crear un proyecto, en este caso seleccionaremos la opción “Aplicación web ASP .NET Core”.



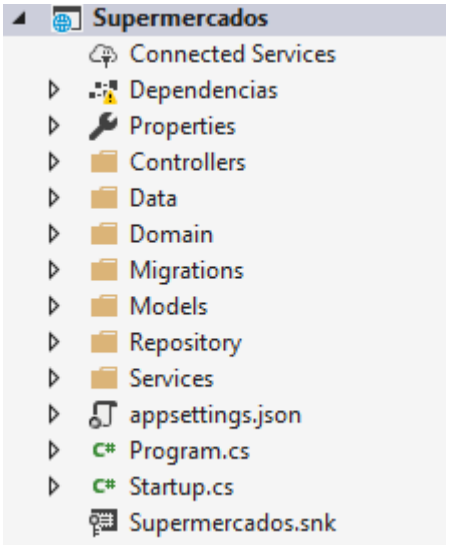
A continuación le damos un nombre al proyecto y una ubicación para guardarlo. Posteriormente seleccionamos una plantilla de API que nos creará un proyecto API por defecto.



Como podemos observar nos ha creado un controlador con una petición GET y un modelo de prueba con sus propiedades.



En nuestro caso crearemos las carpetas pertinentes que necesitaremos para organizar el proyecto con una arquitectura por capas quedando organizado de la siguiente manera:



Pero ¿cómo podemos ver su funcionamiento? Ya que no contamos aún con la parte de presentación vamos a ver un ejemplo del funcionamiento de la API que nos hemos creado de prueba a través de Postman.

En el controlador agregamos una ruta a la que accedemos para hacer las peticiones:

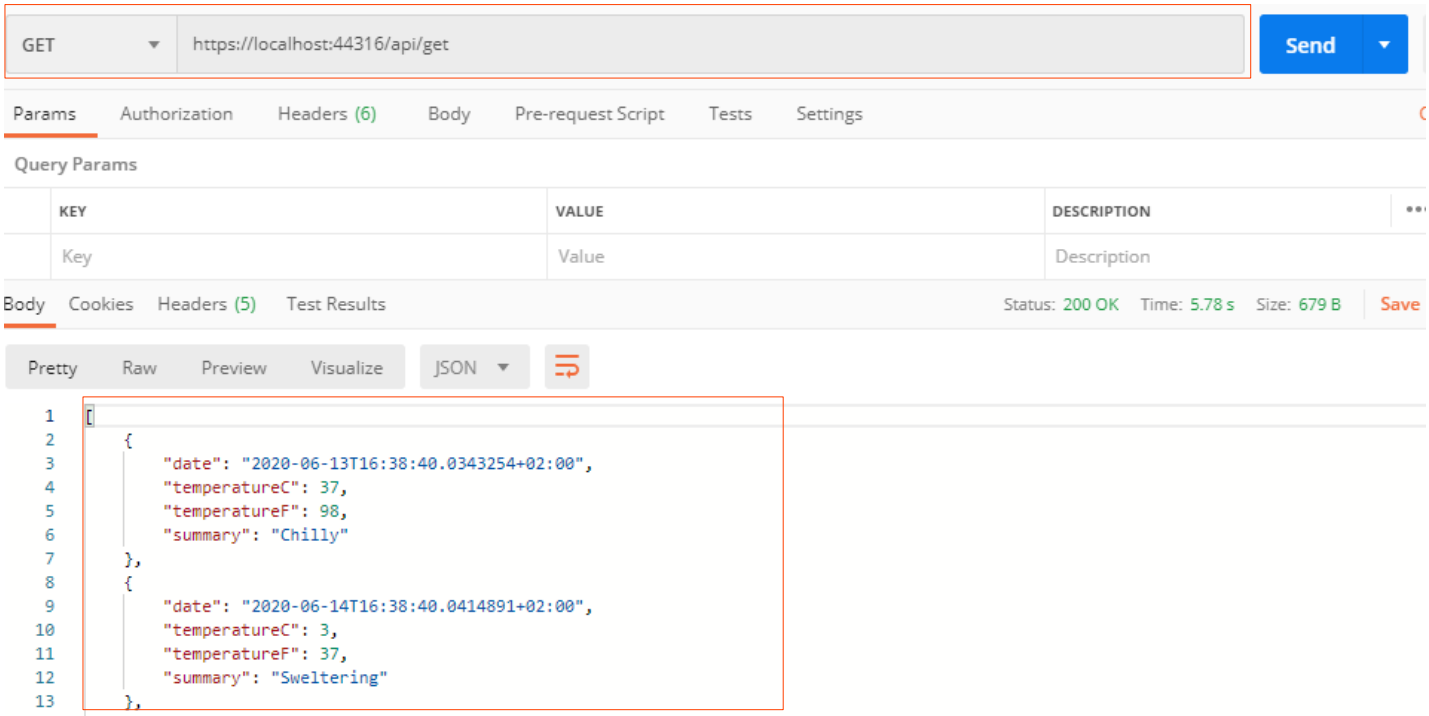
```
namespace WebApplication1.Controllers
{
    [ApiController]
    [Route("api/")]
```

Entonces podemos acceder a cada una de las peticiones con la siguiente url "https://localhost:44316/api/(nombreMetodo)"

Además de añadirle una ruta al método, en este caso a la petición GET:

```
[HttpGet]
[Route("get")]
0 referencias
public IEnumerable<WeatherForecast> Get()
{
    var rng = new Random();
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

Ahora vamos a probar su funcionamiento en Postman y accedemos a la url de la petición:



Podemos observar que nos devuelve los valores del objeto en formato JSON. Estos valores posteriormente serán recibidos en Ionic.

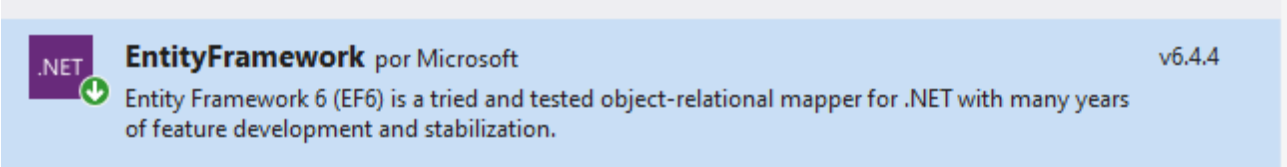
3.3.ADO.NET EntityFramework

Entity Framework es el ORM (Object-Relational Mapper) de Microsoft, con versiones tanto para la plataforma .NET "tradicional" como para. NET Core. Es un asignador relacional de objetos que permite a los desarrolladores de .NET trabajar con datos relacionales utilizando objetos específicos del dominio. Elimina la necesidad de la mayoría del código de acceso a datos que los desarrolladores generalmente necesitan escribir.

Permite crear un modelo escribiendo código o utilizando cuadros y líneas en el Diseñador EF. Ambos enfoques se pueden utilizar para apuntar a una base de datos existente o crear una nueva base de datos. Es el ORM principal que Microsoft proporciona para .NET Framework y la tecnología de acceso a datos recomendada.

3.3.1.Instalación de EntityFramework

Para instalar EntityFramework se necesita un paquete NuGets que podemos encontrar en VisualStudio. Para ello hacemos click derecho en nuestra solución y seleccionamos la opción *Administrar paquetes NuGets…* En esta ventana buscamos el paquete *EntityFramework.Core* y procedemos a instalarlo en nuestro proyecto.



También podemos instalar el framework de entidades utilizando la consola del administrador de paquetes. Para hacerlo, primero debe abrirlo utilizando el menú Herramientas -> NuGet Package Manager -> Package Manager Console y luego ingrese esto:

Install Package EntityFramework

Esto instalará Entity Framework y agregará automáticamente una referencia al ensamblaje en su proyecto.

3.4.CodeFirst

Code First se considera la alternativa más adecuada para aquellos casos en los que hay que crear la base de datos desde cero en conjunto con la aplicación. Usando Code First literalmente no deberemos escribir una sola sentencia SQL (lo cual no implica que no debamos conocer aspectos propios de su arquitectura). Es un enfoque muy popular y nos permite el control total sobre el código en lugar de la actividad de la base de datos. Por lo que podemos realizar todas las operaciones de base de datos desde el código y los cambios manuales a la base de datos se han perdido y todo depende del código.

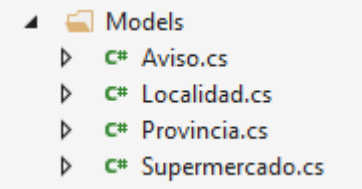
Ventajas que ofrece su uso:

- Máximo control sobre el código de tu modelo de datos en C#, ya que son clases que construyes desde cero.
- Te ofrece control sobre las colecciones de objetos que quieres que se carguen de modo "perezoso" (es decir, a medida que se vayan necesitando).
- El código es el que manda y el "tooling" se encarga de generar la base de datos en función de tu código.
- La estructura de la base de datos es muy fácil de mantener bajo control de código (con Git o similar) ya que no se guarda en absoluto: se guarda el código de nuestras clases del modelo, y se genera la base de datos bajo demanda.

Utilizo este método al tener que crear una base de datos desde cero, y así puedo centrarme en el código del proyecto dejando a un lado la base de datos.

3.4.1.Creación de modelos

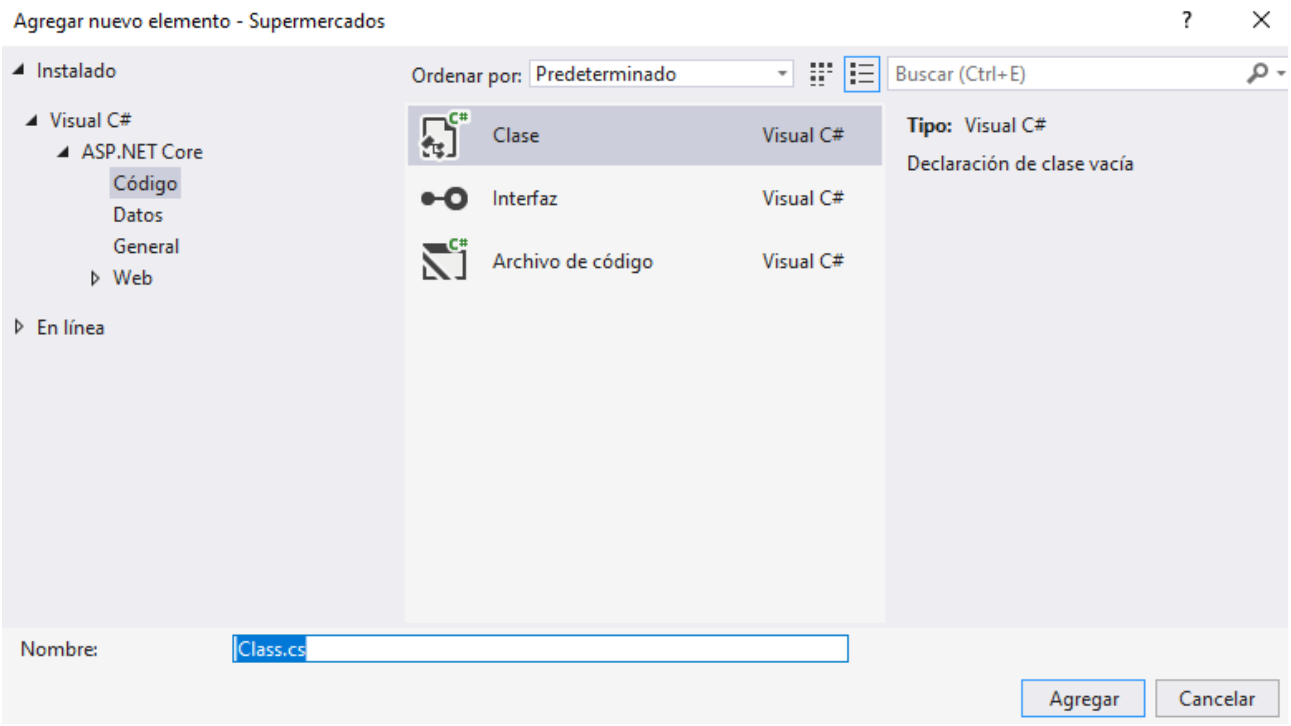
Tenemos que crear nuestros modelos, es decir, nuestro conjunto de clases que representan los datos que la aplicación administra. Los modelos de esta aplicación son:



- Aviso: almacena los datos de cada una de las notificaciones.
- Localidad: almacena las localidades.
- Provincia: almacena las provincias.
- Supermercado: almacena los supermercados existentes.

Para crear los modelos debemos seguir los siguientes pasos:

- En el Explorador de soluciones, haga clic con el botón derecho en el proyecto. Seleccione Agregar > Nueva carpeta. Asigne a la carpeta el nombre Models.
- Haga clic con el botón derecho en la carpeta Models y seleccione Agregar > Clase. Asigne el nombre de la clase y seleccione Agregar.



Este es un ejemplo del modelo *Aviso*:

```
namespace AvisoServices.Models
{
    57 referencias
    public class Aviso
    {
        9 referencias | 1/1 pasando
        public int Id { get; set; }
        [Required]
        18 referencias | 1/1 pasando
        public string Supermercado_id { get; set; }
        [Required]
        19 referencias | 1/1 pasando
        public string Provincia_id { get; set; }
        18 referencias | 1/1 pasando
        public string Localidad_id { get; set; }
        10 referencias | 1/1 pasando
        public string Producto { get; set; }
        8 referencias | 1/1 pasando
        public string Comentario { get; set; }
        11 referencias | 1/1 pasando
        public DateTime FechaCreacion { get; set; }
    }
}
```

En este modelo podemos apreciar que tiene el atributo *ID* que será la clave primaria de nuestra tabla de notificaciones. Además *Supermercado_id, Localidad_id* y *Provincia_id* serán claves foráneas (la relación entre las entidades se explicará más adelante).

3.4.2.Incorporación del contexto de la base de datos

Una vez creados nuestros modelos, debemos crear el contexto asociado a nuestro acceso a datos.

Nos crearemos una carpeta llamada “Data” y dentro de la misma una clase, en este caso se llama *FoodLackContexto*.

Primero debemos definir la clase y que derive de *DbContext*:

```
public class FoodLackContexto : DbContext
{
```

Seguidamente debemos registrar todas las entidades que queremos que se creen en la base de datos, definiéndolas como un tipo DbSet. El nombre de la propiedad será el que tenga la tabla asociada.

```
public DbSet<Aviso> AvisoItems { get; set; }
1 referencia
public DbSet<Supermercado> Supermercado { get; set; }
1 referencia
public DbSet<Provincia> Provincia { get; set; }
1 referencia
public DbSet<Localidad> Localidad { get; set; }
```

Finalmente debemos configurar la clase *startup.cs* de nuestro proyecto indicándole el uso del contexto y su conexión con la base de datos. Por lo que hay que añadir las siguientes lineas:

```
services.AddDbContext<FoodLackContexto>(options =>
 options.UseSqlServer(Configuration.GetConnectionString("ConexionTest")));
```

"ConexionTest" forma parte de un archivo .json que contiene la configuración de la base de datos, por lo que debemos ir al archivo *appsettings.json* y agregarlo de la siguiente manera:

```
"ConnectionStrings": {
  "ConexionTest": "Server=(localdb)\\MSSQLLocalDB;Database=FoodLack;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

- Server=(localdb)\\MSSQLLocalDB: es la instancia a la que nos conectamos.
- Database=FoodLack: es el nombre de la base de datos que queremos crear.

Una vez configurado todo lo anterior procedemos a crear las tablas a partir de nuestros modelos con el contexto.

3.4.3.Creación de tablas

La clase DbContext tiene un método llamado OnModelCreating que toma una instancia de Model-Builder como parámetro. El marco llama a este método cuando su contexto se crea por primera vez para construir el modelo y sus asignaciones en la memoria.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<Aviso>().HasData(new Aviso
    {
        Id = 1,
        Supermercado_id = "01",
        Provincia_id = "01",
        Localidad_id = "01",
        Producto = "leche",
        Comentario = "No queda leche puleva",
        FechaCreacion = DateTime.Now
    },
```

De esta manera a partir del modelo creamos la entidad en nuestra base de datos.

Una vez creadas las entidades procedemos a definir las relaciones entre ellas y la creación de las claves foráneas, mediante el método Fluent API. Ejemplo de creación de relación y clave foránea:

```
modelBuilder.Entity<Aviso>()

            .HasOne<Supermercado>()
            .WithMany()
            .HasForeignKey(p => p.Supermercado_id);

modelBuilder.Entity<Aviso>()
          .HasOne<Provincia>()
          .WithMany()
          .HasForeignKey(p => p.Provincia_id);
```

3.4.4.SQLServer Express Edition

Para descargar e instalar SQLServer Express Edition podemos hacerlo en su web oficial de manera gratuita.

Una vez instalado debemos asegurarnos de que la instancia está activa, en caso contrario no podemos acceder a la base de datos que vamos a crear posteriormente, en consola a través de los siguientes comandos (es recomendable instalar Microsoft SQL Server Management Studio 18 para observar que es correcta la creación de la base de datos).

Ver estado de la instancia:

SqlLocalDB info (nombre de instancia)

```
C:\Windows\system32>sqllocaldb info mssqllocaldb
Nombre:               mssqllocaldb
Versión:            13.1.4001.0
Nombre compartido:
Propietario:              DESKTOP-NITCAHN\jessb
Crear automáticamente:        Sí
Estado:             Detenido
Hora de último inicio:    04/06/2020 15:14:28
Nombre de canalización de instancia:
```

Iniciar instancia:

SqlLocalDB start (nombre de instancia)

```
C:\Windows\system32>sqllocaldb start mssqllocaldb
Instancia "mssqllocaldb" de LocalDB iniciada.
```

Utilizaré la instancia que se crea automáticamente durante la instalación.

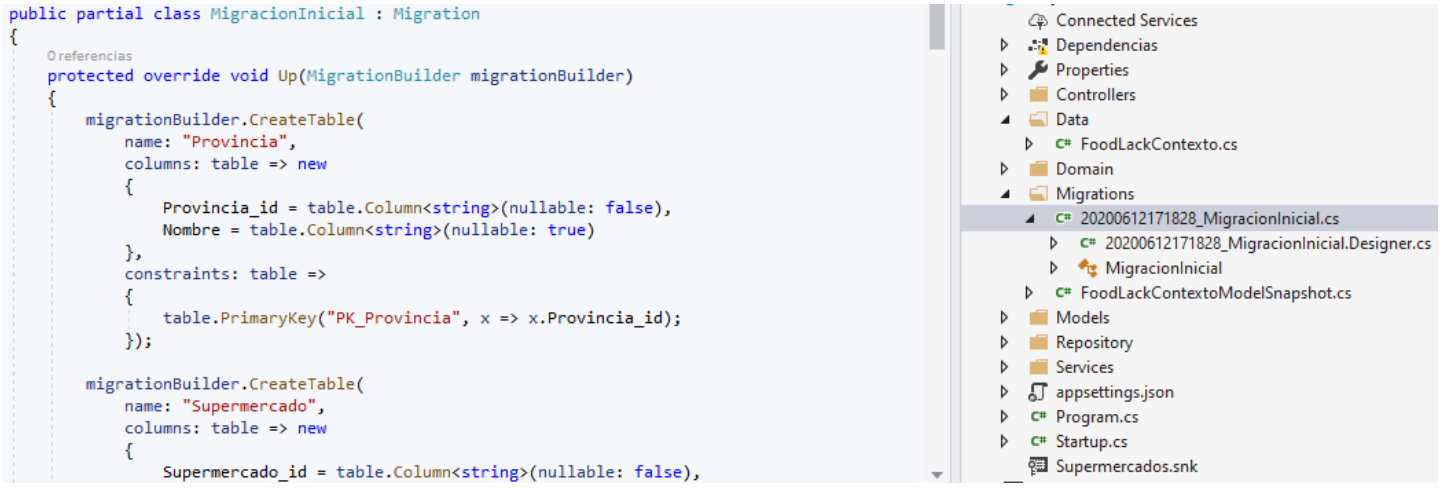
3.4.5.Migrar base de datos

Estando todo listo debemos migrar la base de datos. Para ello abrimos la consola en VisualStudio y escribimos *add-migration (nombre de la migración)*. Cuando realizamos la migración, se crea automáticamente la carpeta *Migrations* con el resultado de la migración.

```
PM> EntityFrameworkCore\add-migration MigracionInicial
Both Entity Framework Core and Entity Framework 6 are installed. The Entity Framework Core tools are running. Use 'EntityFramework6\Add-Migration' for Entity Framework 6.
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

Seguidamente para actualizar los datos en SQLServer utilizamos el comando *update-database* y ya podemos observar que se ha creado correctamente a través de Microsoft SQLServer Management Studio.

```
PM> EntityFrameworkCore\update-database
Both Entity Framework Core and Entity Framework 6 are installed. The Entity Framework Core tools are running. Use 'EntityFramework6\Update-Database' for Entity Framework 6.
Build started...
Build succeeded.
Done.
```



Ejemplo de archivos de migración creados

3.5.Principios SOLID

Solid es un acrónimo inventado por Robert C.Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo. Los costes de mantenimiento pueden abarcar el 80% de un proyecto de software por lo que hay que valorar un buen diseño.

Estos principios nos permite:

- Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
- Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
- Permitir escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

En definitiva, desarrollar un software de calidad.

3.6.Inyección de dependencias

Inyección de Dependencias es un patrón de diseño de software que nos permite desarrollar componentes acoplados libremente o desacoplados para obtener como resultado la fácil gestión de cambios a futuro además de una implementación fácil de pruebas unitarias.

Proporciona un mecanismo para la construcción de gráficos de dependencia independientes de las definiciones de clase, aplicando esta técnica nuestras clases van a depender de interfaces y no de implementaciones, esto se relaciona directamente con el primero de los 5 principios para desarrollar software de calidad, haciendo referencia a SOLID y el principio de responsabilidad única.

Por su parte ASP.NET Core nos permite la utilización del patrón de diseño Inyección de Dependencias sin la necesidad de utilizar librerías de terceros para lograrlo, además ASP.NET Core provee un Contenedor de Inversión de Controles(IoC) quien es el encargado de proveer las instancias de los tipos los cuales le decimos en el inicio de la Aplicación(Startup.cs).

Una vez aprendida la teoría, ¿cómo implementamos las interfaces?

Creamos la interfaz en nuestra capa de negocio, en este caso se llama *IAvisoServices*, en la cuál definimos los métodos a los que el controlador accederá, sin necesidad de acceder a ellos directamente en la clase *AvisoServices*.

```
namespace AvisoServices.Services
{
    4 referencias
    public interface IAvisoServices
    {
        2 referencias
        Task AddAvisoItems(Aviso items);
        3 referencias | 1/1 pasando
        IEnumerable<Aviso> GetAvisoItems();

    }
}
```

Y en la clase *AvisoServices* creamos los métodos, en la cual podemos observar que extiende de su interfaz *IAvisoServices*.

```
public class AvisoServices : IAvisoServices
{
```

Esta clase a su vez llama a los métodos de la interfaz del repositorio, lo que permite que las capas se comuniquen a través de las interfaces de las clases desacoplándolas completamente.

Las interfaces hay que configurar su dependencia en la clase *startup.cs* mediante el método AddScoped.

```
services.AddScoped<IAvisoServices, AvisoServices.Services.AvisoServices>();
```

3.7.Asincronia

Las consultas asincrónicas evitan bloquear un subproceso mientras la consulta se ejecuta en la base de datos. Las consultas asincrónicas son importantes para mantener una interfaz de usuario dinámica en las aplicaciones cliente de gran tamaño. También pueden aumentar el rendimiento de las aplicaciones web donde liberan el subproceso para atender otras solicitudes de las aplicaciones web.

Entity Framework Core proporciona un conjunto de métodos de extensión asincrónicos similares a los de LINQ, que ejecutan una consulta y devuelven resultados. Entre los ejemplos se incluyen ToListAsync(), ToArrayAsync(), SingleAsync(). No hay versiones asincrónicas de algunos operadores de LINQ como Where(...) o OrderBy(...) porque estos métodos solo generan el árbol de la expresión de LINQ y no hacen que la consulta se ejecute en la base de datos.

En esta aplicación se realizan consultas a la base de datos de manera asíncrona, como podemos observar en el siguiente método:

```
public async Task AddAvisoItems(Aviso items)
{
    _context.AvisoItems.Add(items);
    await _context.SaveChangesAsync();
}
```

Ejemplo método asíncrono

Definimos con *async* la asincronía del método y como EF Core no admite que varias operaciones en paralelo se ejecuten en la misma instancia de contexto y siempre debe esperar que se complete una operación antes de iniciar la siguiente usamos la palabra clave *await* en cada una de las operaciones

3.8.Pruebas de integración

Las pruebas de integración evalúan los componentes de una aplicación en un nivel más amplio que las pruebas unitarias. Las pruebas unitarias se usan para probar componentes de software aislados, como métodos de clase individuales. Las pruebas de integración confirman que dos o más componentes de una aplicación funcionan juntos para generar un resultado esperado, lo que posiblemente incluya a todos los componentes necesarios para procesar por completo una solicitud.

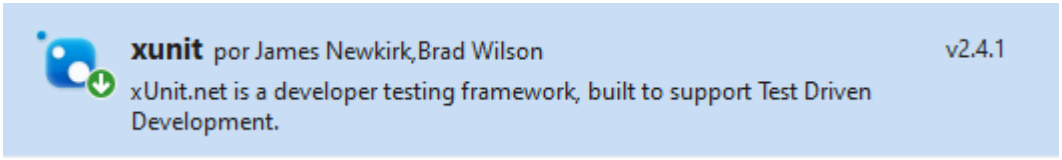
Para realizar las pruebas utilizaremos la herramienta *xUnit* y para la simulación de las clases *Moq*.

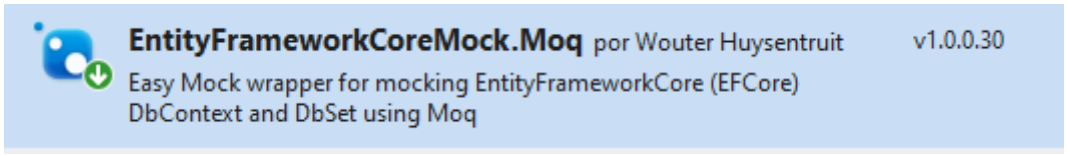
Moq nos sirve para simular el comportamiento de los objetos, nos ayuda aprovechar toda la potencia de C# para crear “mocks” limpios y mantenibles.

Estas pruebas siguen una secuencia de eventos que incluye los pasos de prueba normales Arrange, Act y Assert:

1. Se configura el host web del SUT.
2. Se crea un cliente de servidor de pruebas para enviar solicitudes a la aplicación.
3. Se ejecuta el paso de prueba *Arrange*: la aplicación de prueba prepara una solicitud.
4. Se ejecuta el paso de prueba *Act*: el cliente envía la solicitud y recibe la respuesta.
5. Se ejecuta el paso de prueba *Assert*: la respuesta real se valida como correcta o errónea en función de una respuesta esperada.
6. El proceso continúa hasta que se ejecutan todas las pruebas.
7. Se notifican los resultados de la prueba.

Lo primero que necesitamos es instalar *xUnit* y *Moq.* Son dos paquetes NuGets que están disponibles en VisualStudio. En la herramienta de instalación de NuGets lo encontramos y lo descargamos.





Una vez instalados procedemos a realizar la prueba de integración.

El objetivo de esta prueba es saber si el repositorio y el servicio se comunican correctamente. Comenzamos observando que antes del inicio de la prueba tenemos *[Fact]*, esto indica que el método es una prueba y que debe ser ejecutada. Esto debe definirse siempre para que se ejecute.

Seguidamente creamos un mock de la interfaz del repositorio y se lo pasamos por parámetro a un servicio nuevo creado. Simulando la comunicación entre ambos.

```
public class AvisoServicesTest
{

    [Fact]
    0 referencias
    public void GetAvisoItems_ReturnAll()
    {

        var mockAvisoRepositorio = new Mock<IAvisoRepository>();
```

```
var service = new AvisoServices.Services.AvisoServices(mockAvisoRepositorio.Object,
```

Vamos a crear una lista simulando los datos que contiene el repositorio.

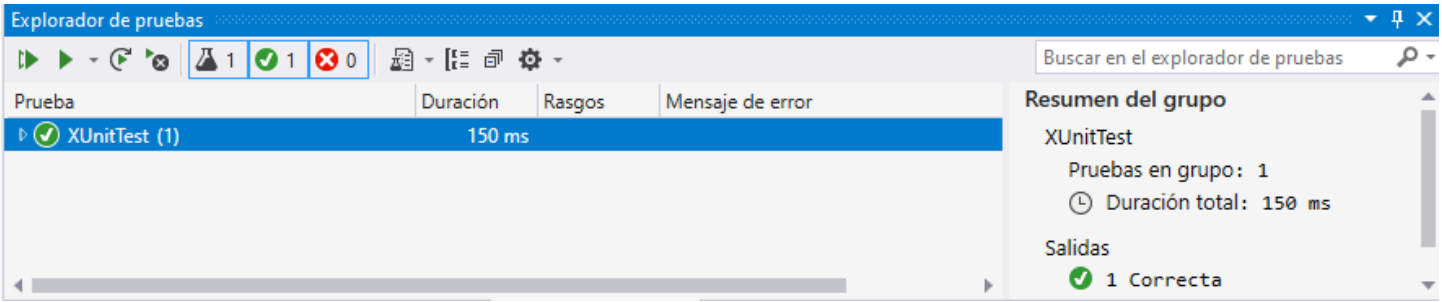
```
var listaAvisoRepo = new List<Aviso>() {
    new Aviso
    {
        Id = 1,
        Supermercado_id = "01",
        Provincia_id = "01",
        Localidad_id = "01",
        Producto = "leche",
        Comentario = "No queda leche puleva",
        FechaCreacion = date
    }};
```

Ahora indicamos que lo correcto es que al realizar el método *GetAvisoItems* del repositorio nos va a devolver la lista que hemos creado, es decir, lo que esperamos obtener como resultado.

```
mockAvisoRepositorio.Setup(r => r.GetAvisoItems()).Returns(listaAvisoRepo);
```

Entonces ya solo faltaría crear una lista con lo que nos devuelve la llamada del servicio, puesto que esta llamada contiene los datos que el repositorio contiene. Si se realiza la comunicación debe cumplir los requisitos que definimos en *Assert* y la prueba es correcta.

```
var avisosService = service.GetAvisoItems();
```



Prueba satisfactoria

4.Desarrollo en la capa cliente

4.1.Ionic

Ionic es un Framework de código abierto que se utiliza en el desarrollo de aplicaciones móviles híbridas, es decir, se combinan el HTML5, CSS y JavaScript dando como resultado aplicaciones con una interfaz amigable e intuitiva para el usuario que luego se comercializan o descargan en plataformas como Android o Ios.

La base de Ionic esta desarrollada sobre AngularJs y Cordova, vio la luz en 2013 con la única intención de que desarrolladores pudieran crear aplicaciones móviles híbridas con la particularidad y beneficios de los dos framework sobre los que fue construida.

Una de las principales ventajas de trabajar con Ionic es que aprovecha todos los plugins (Hardware, software, imágenes, texto, códigos QR, etc) del marco de desarrollo móvil Cordova.

Vamos a trabajar con la versión actual de Ionic 5, que a diferencia de sus versiones anteriores, es una herramienta de desarrollo de aplicaciones móviles y empieza con remplazar AngularJS por Angular moderno.

El conjunto de componentes de esta herramienta utiliza elementos personalizados y las API DOM de Shadow disponibles en todos los navegadores modernos para dispositivos móviles y de escritorio. El desarrollo de aplicaciones móviles con Ionic nos garantiza que la implementación del proyecto sea mucho más estable, sencilla y con una interfaz de usuario óptima.

Con respecto al control del DOM, AngularJS incluye jQLite, una pequeña versión ultraligera de jQuery que permite la gestión de DOM (Document Object Model, Modelo de Objetos del Documento) de forma compatible en la inmensa mayoría de navegadores, dejando una pequeñísima huella en los mismos haciendo la app mucho más ligera y eficiente. Por supuesto, si con jqLite no tuviésemos suficiente, podremos implementar una funcionalidad extendida cargando jQuery con el documento, con la consecuente pérdida de velocidad.

Entramos en el campo fuerte de Ionic, donde destaca su simplicidad para con el desarrollo . De una forma similar (que no igual) a como hace Google con Android, Ionic procura hacer uso de las capacidades que ofrecen HTML5 y CSS3 para ofrecer experiencias de usuario sobretodo rápidas.

Como la interfaz de usuario que por defecto facilita Ionic es prácticamente HTML5 en su estado más puro , la personalización de ésta se realiza mediante SASS junto con las librerías propias que incorpora Ionic, con más de 450 iconos y símbolos de código abierto para su libre uso, además de otras utilidades con las que extender esta capacidad de personalización. Además Ionic aprovecha los principios del diseño web Responsive para adaptar el contenido al tamaño de pantalla o densidad de píxeles ofrecidas por el destino, optimizando así los recursos empleados y prestándose a soportar prácticamente todos los dispositivos del mercado.

4.1.1.Ciclos de vida

Ionic cuenta con eventos que indican el ciclo de vida de las páginas. A continuación se pueden ver algunos consejos sobre los casos de uso para cada uno de los eventos del ciclo de vida.

- **ngOnInit** - Inicializar el componente y carga los datos desde servicios que no necesitan ser actualizados en cada visita posterior al componente.
- **ionViewWillEnter** - Desde ionViewWillEnter se llama cada vez que se navega hacia la View (independientemente de que haya sido inicializada o no), es un buen método cargar datos de los servicios. Sin embargo, si tus datos son cargados durante la animación, puede generar distintas manipulaciones en el DOM, lo que puede causar algunas animaciones se vean lentas o trabadas.
- **ionViewDidEnter** - Si ves problemas de performance usando ionViewWillEnter al cargar datos, puedes hacer tus llamadas de datos en ionViewDidEnter en su lugar. Este evento no se disparará hasta que la página sea visible por el usuario, sin embargo, es posible que quieras usar un indicador de loading o componente de tipo skeleton, para que el contenido no sea visible por un instante de forma no natural después de que la transición esté completa.
- **ionViewWillLeave** - Se puede utilizar para limpiar el contenido del componente, como así también para desusbscribirse a distintos eventos y/o observables. Puesto que ngOnDestroy podría no dispararse cuando navegues desde la página actual, es posible realizar un cleanup aquí si es que no quieres que esté activo mientras la pantalla no está en vista.

- **ionViewDidLeave** - Cuando este evento se ejecute, debes saber que la nueva página ha realizado su transición completamente, por lo que cualquier lógica que no pueda hacer normalmente cuando la vista sea visible puede ir en este lugar.
- **ngOnDestroy** - La lógica de clean up de tus páginas, de las cuales no deseas hacer un clean up ionViewWillLeave.

4.2.Instalación de Ionic

Para su instalación es necesario NodeJs, ya que este cuenta con un manejador de paquetes llamado *npm* el cual nos sera necesario para instalar muchas de nuestras dependencias.

4.2.1.NodeJS

Para instalar NodeJS debemos de dirigirnos al sitio oficial y descargar su instalador.
Para poder corroborar que la instalación fue satisfactoria y además para saber qué versión instalamos de NodeJS y de npm se pueden ejecutar los comandos *node -v* y *npm -v.*

```
C:\Proyect\proyectoIonic>node -v
v12.16.2
```

4.2.2.Instalación Ionic

Una vez instalado NodeJS, procedemos a instalar Ionic ejecutando el siguiente comando:

npm install -g ionic

Nos dará este resultado junto a su versión:

```
C:\Users\jessb\AppData\Roaming\npm\ionic -> C:\Users\jessb\AppData\Roaming\npm\node_modules\ionic\bin\ionic
+ ionic@5.4.16
removed 1 package and updated 15 packages in 15.619s
```

A continuación procederemos a crear el proyecto en Ionic 5.

4.3.Creación de proyecto Ionic

Para crear un proyecto en Ionic ejecutamos el siguiente comando:

ionic start nombredelProyecto

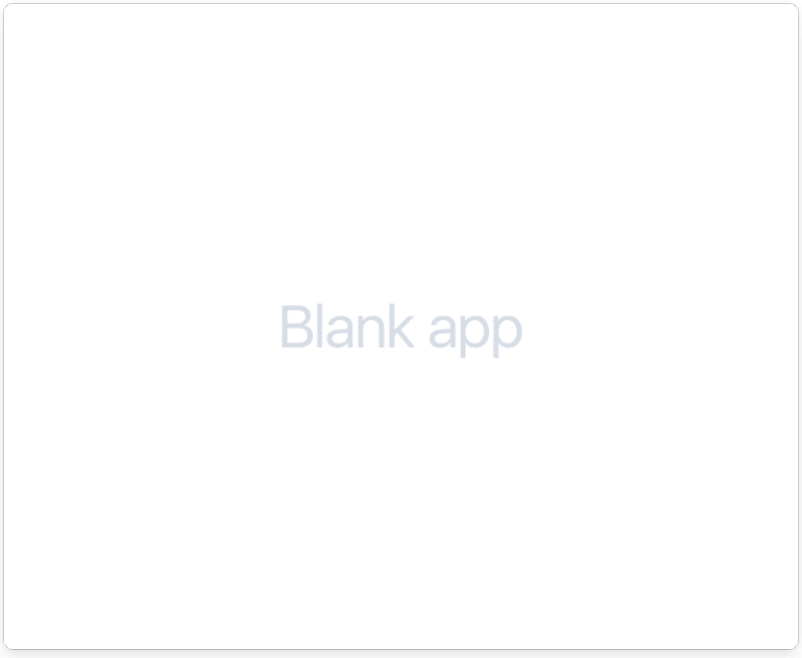
A continuación seguimos los siguientes pasos:

1. Elegir el framework

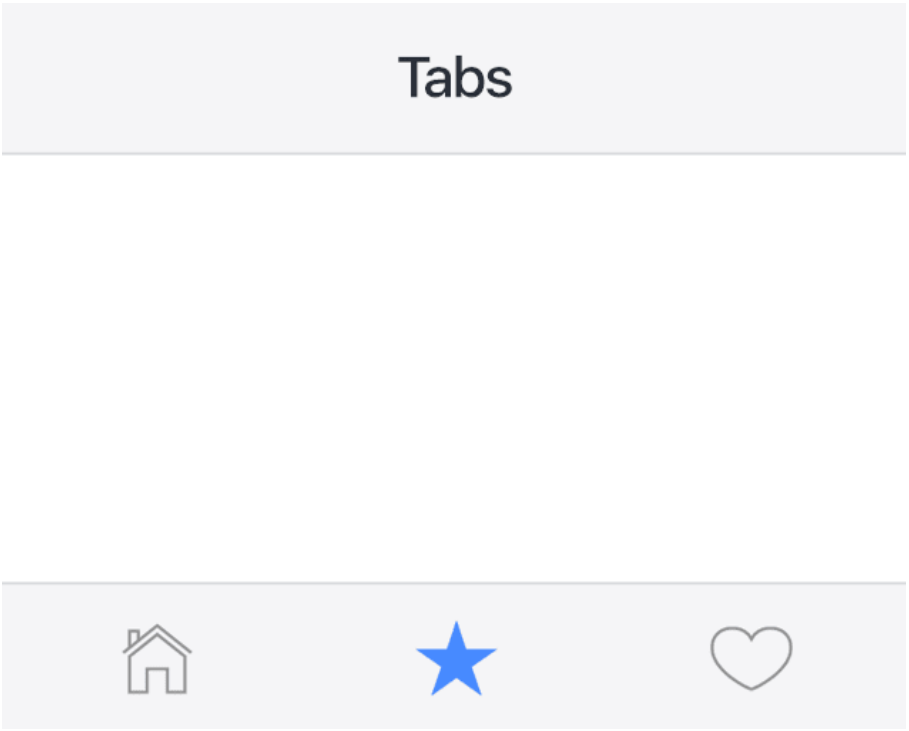
```
Pick a framework!

Please select the JavaScript framework to use for your new app.
--type option.

? Framework: (Use arrow keys)
> Angular | https://angular.io
  React   | https://reactjs.org
```

2. Elegimos la plantilla

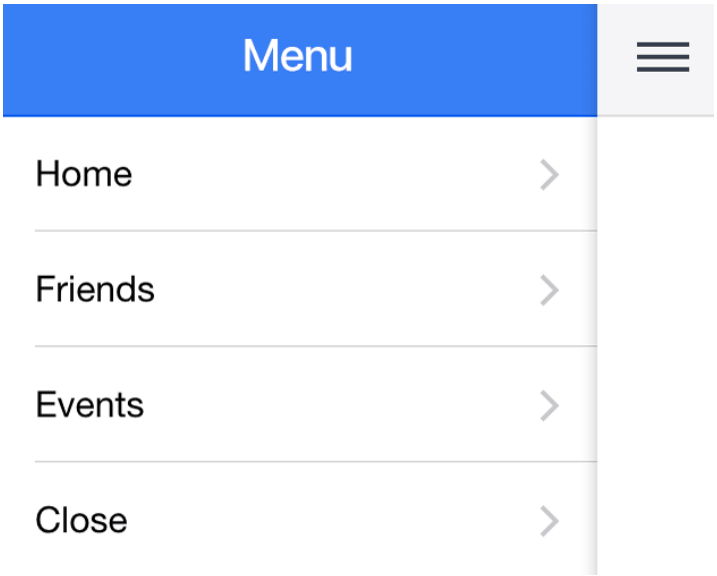
```
Let's pick the perfect starter template!

Starter templates are ready-to-go Ionic apps that come packed with everything you need to
prompt next time, supply template, the second argument to ionic start.

? Starter template: (Use arrow keys)
> tabs          | A starting project with a simple tabbed interface
  sidemenu      | A starting project with a side menu with navigation in the content area
  blank         | A blank starter project
  my-first-app  | An example application that builds a camera with gallery
  conference    | A kitchen-sink application that shows off all Ionic has to offer
```



Plantilla blank: plantilla en blanco



Plantilla Tabs: con tres vistas donde puedes navegar entre ellas muy fácil.



Plantilla sidemenu: Crea un proyecto con un menú lateral

3. Comienza la creación del proyecto. Esto puede durar varios minutos.

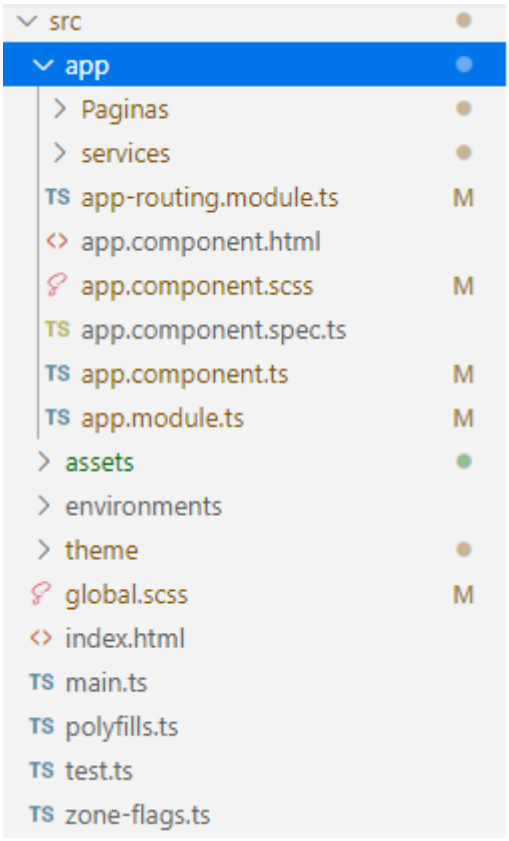
4. Iniciamos el proyecto con el comando *ionic serve* y automáticamente nos abrirá en nuestro navegador predeterminado el resultado del proyecto creado. Ahora cualquier modificación del código podrá hacerse visible sin tener que reiniciar la aplicación.

```
C:\proyectoNuevo>cd proyectoIonic

C:\proyectoNuevo\proyectoIonic>ionic serve
> ng.cmd run app:serve --host=localhost --port=8101
[ng] Compiling @angular/core : es2015 as esm2015
[ng] Compiling @ionic-native/core : module as esm5
[ng] Compiling @angular/compiler/testing : es2015 as esm2015
```

```
[ng] : Compiled successfully.

[INFO] Development server running!

       Local: http://localhost:8101

       Use Ctrl+C to quit this process

[INFO] Browser window opened to http://localhost:8101!

[ng] Date: 2020-06-14T10:05:35.814Z - Hash: d395e1167e25333f3239
[ng] 98 unchanged chunks
[ng] Time: 4235ms
[ng] : Compiled successfully.
```

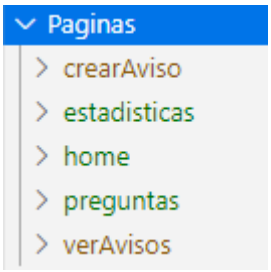
4.4.Estructura de proyecto en Ionic

Este proyecto está organizado en capas, cada una con sus propias funciones.

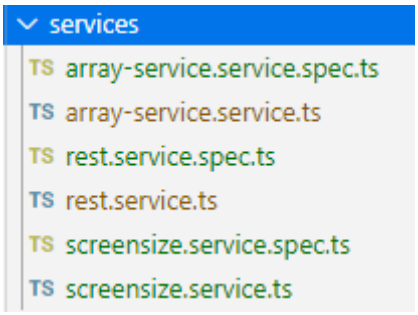


Organización general del proyecto

Paginas: en esta carpeta están organizadas las páginas por las que navegamos en nuestra aplicación.



Tenemos a su vez una carpeta para cada página, en este caso está la página donde creamos las notificaciones, donde vemos las estadísti cas, la página de inicio, la página donde consultamos las preguntas más frecuentes y la página donde vemos las notificaciones que han sido creadas.



En esta carpeta nos encontramos la lógica de la aplicación, el servicio *array-service.service.ts* almacena las operaciones que se realizan con los arrays de la aplicación (explicado más ade lante), el servicio *rest.service.ts* es donde realizamos las peti ciones a la API Rest, y el servicio *screensize.service.ts* es para configurar el tamaño de la pantalla.

4.4.1.Creación de menú lateral vertical

En la documentación oficial de Ionic encontramos todos los componentes que se pueden utilizar en Ionic. En este caso nos vamos a centrar en la creación del menú lateral vertical.

Este menú lo crearemos en nuestro *app.component.html.*

Utilizaremos la etiqueta *<ion-split-pane>* asignandole un *contentId.* Y procedemos a crear el menu con la etiqueta *<ion-menu>*. Le indicamos con *side="start"* que es un menu que se esconde en las pantallas pequeñas.

```
<ion-split-pane when="md" contentId="principal">
<ion-menu side="start" menuId="first" contentId="principal">
```

Y finalmente antes de cerrar la etiqueta *<ion-split-pane>* agregamos la ruta de salida indicándole el *contentId* que hemos creado anteriormente.

```
  </ion-menu>
  <ion-router-outlet id="principal"></ion-router-outlet>
</ion-split-pane>
```

El enrutador de salida <ion-router-outlet /> extiende la pagina de Angular<router-outlet />con algu- na funcionalidad adicional para habilitar mejores experiencias para dispositivos móviles.

Cuando una aplicación está encapsulada en <ion-router-outlet />, Ionic navega de de una manera di- ferente. Cuando navegas a una pagina nueva, Ionic mantendrá la pagina existente en el DOM, pero

la ocultara de tu vista y procederá a ejecutar la transición de la nueva pagina. La razón por la que hacemos esto es por que existen dos motivos:

- Podemos mantener el estado de la página anterior (datos en la pantalla, posición de desplazamiento, etc.)
- Podemos proporcionar una transición más suave a la página ya que ya está allí y no necesita ser recreada.

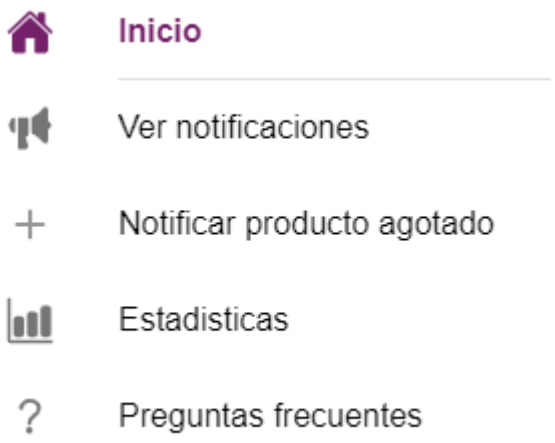
Por último necesitamos crear los *Items* que contiene este menú, la estructura es la siguiente:

```
<ion-menu-toggle autoHide="false">
  <ion-item lines="none" routerLink="verAvisos">
    <ion-icon slot="start" name="megaphone"></ion-icon>Ver notificaciones
  </ion-item>
</ion-menu-toggle>
```

En estas líneas le asignamos al item una ruta a la que accedemos al seleccionarlo con la propiedad *routerLink*, nos llevaría a una página que más adelante crearemos.

También le añadiremos un icono. Los iconos disponibles para Ionic se pueden ver en la página web oficial de Ionic en el apartado de Icons.

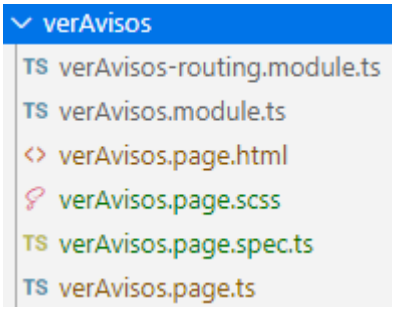
Y finalmente el menú tendría este aspecto:



4.4.2.Creación de páginas

Con el menú accedemos a diferentes páginas de nuestra aplicación, pero ¿cómo creamos estas páginas? Se crean ejecutando el comando *ionic g page nombreDePagina*

Cuando termina el proceso de creación se nos ha añadido varios archivos:



Podemos observar que nos ha creado el módulo de la página, su archivo HTML, el CSS pertinente y el archivo ts de la página. Nosotros modificaremos principalmente el archivo VerAvisos.page.ts.

También nos ha modificado el archivo de rutas *app-routing.module.ts* donde indicamos las rutas que vamos a utilizar, con su nombre y ubicación:

Esta modificación la realiza de manera automática, podemos apreciarlo en la siguiente ilustración:

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', loadChildren: () => import('./Paginas/home/home.module').then( m => m.HomePageModule)},
  {
    path: 'crearAviso',
    loadChildren: () => import('./Paginas/crearAviso/crearAviso.module').then( m => m.ExistenciasPageModule)
  },
  {
    path: 'verAvisos',
    loadChildren: () => import('./Paginas/verAvisos/verAvisos.module').then( m => m.VerExistenciasPageModule)
  },
```

4.4.3.Creación de servicios

Los servicios permiten administrar la aplicación y la interfaz mediante programación. Sus métodos realizan operaciones necesarias en la lógica de la aplicación.

Crearemos un servicio específico para las llamadas a la API REST, al cual llamaremos *rest*.

Para crear un servicio hay que ejecutar el siguiente comando:

ionic -g service nombreDeServicio

Este servicio lo guardaremos en la carpeta *Services*.

Cuando deseemos utilizar un servicio desde una página, lo primero que hay que hacer es importar el servicio.

```
import { RestService } from '../../services/rest.service';
import { ArrayServiceService } from '../../services/array-service.service';
```

Y posteriormente agregarlo al constructor de la página para poder utilizar sus métodos.

```
constructor(public servicioGet:RestService,public arrayService:ArrayServiceService) {
}
```

4.5.Peticiones a API REST

Mantener una aplicación backend en un servidor remoto es muy funcional, ahí se va a alojar la lógica de negocio y la lógica funcional más pesada y reutilizables en todos los dispositivos donde nos conectamos.

Los servicios en una aplicación híbrida suelen ser muy comunes, ya que no disponen de toda la potencia que una aplicación nativa y entre otras características, debe servirse de un servidor remoto para ejecutar la lógica más compleja, además de mantener la información centralizada y disponible siempre en cualquier dispositivo, poder compartir esa información con otros usuarios, incluso poder utilizarla en común, ya que esto, con toda la información repartida en dispositivos seria mucho más difícil.

Una vez que tenemos las páginas creadas y los servicios vamos a definir los métodos que realizan las llamadas a la API REST.

Peticiones realizadas:

Petición GET

Para hacer la petición debemos crear un método con el siguiente código:

```
getAviso(){
      var api_url="https://localhost:44394/api/avisos/GetAvisoItems";
            return new Promise (resolve =>
            this.Http.get(api_url).subscribe(data => {
                  resolve(data);
                  }, err => {
       console.log(err);
      })
      )
}
```

- *var api_url="https://localhost:44394/api/avisos/GetAvisoItems"* es la url de la petición, la cuál utilizabamos en Postman como ejemplo.
- *Promise* es una función que devuelve los datos una única vez cuando estos son recibidos.
- *Data* son los datos que se recibe de la API.

Petición POST

Para realizar una petición post debemos enviar los datos necesarios que recoge nuestra API.

```
postAviso(arrayNewAviso: any){

       let datos = arrayNewAviso;
       var url = 'https://localhost:44394/api/avisos/AddAvisoItems';
       return new Promise(resolve => {
            this.Http.post(url,JSON.stringify(datos))
            .subscribe(data => {
             resolve(data);
```

- *var url = 'https://localhost:44394/api/avisos/AddAvisoItems'* es la url de la petición a la API.
- *Promise*, tenemos de nuevo una promesa.
- *Data* son los datos que se recibe al terminar la petición.

Una vez que hemos creado las peticiones, ¿cómo accedemos a ellas desde la página?

En nuestro archivo ts de la página llamamos al método de la siguiente manera:

```
this.servicioGet.getAviso().then(data => {

  this.arrayService.crearArrayAvisos(data);
  this.arrayAvisos=this.arrayService.getAvisos();

}).catch(data => {
})
```

- *.then* se utiliza para recoger los datos que se han solicitado cuando la promesa de la petición se ha resuelto.
- Entonces, recogemos los datos obtenidos en la variable *data*

5.Errores durante el desarrollo

Como todo desarrollo, surgen complicaciones y errores por el camino y en esta aplicación han aparecido varios, en este punto mostraré con cuáles me he ido encontrando y solucionando.

5.1.Permisos CORS

El Intercambio de Recursos de Origen Cruzado (CORS) es un mecanismo que utiliza cabeceras HTTP adicionales para permitir que un user agent obtenga permiso para acceder a recursos seleccionados desde un servidor, en un origen distinto (dominio) al que pertenece. Un agente crea una petición HTTP de origen cruzado cuando solicita un recurso desde un dominio distinto, un protocolo o un puerto diferente al del documento que lo generó.

El W3C Grupo de Trabajo de Aplicaciones Web recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, por sus siglas en inglés). CORS da controles de acceso a dominios cruzados para servidores web y transferencia segura de datos en dominios cruzados entre navegadores y servidores Web.

Para solucionar este problema he necesitado dar los permisos desde el backend al frontend desde la clase *startup.cs* permitiendo que se puedan recibir los recursos desde el origen del frontend.

```
app.UseCors(
options => options.WithOrigins("http://localhost:8100").AllowAnyMethod().AllowAnyHeader()
);
```

Además en la configuración de servicios agregar la línea: *services.AddCors().*

Lo que hace los permisos CORS es añadir nuevas cabeceras HTTP que permiten al servidor describir el conjunto de orígenes que tienen permiso para leer la información usando un explorador web. Adicionalmente, para métodos de solicitud HTTP que causan efectos secundarios en datos del usuario (y en particular, para otros métodos HTTP distintos a GET, o para la utilización de POST con algunos tipos MIME), la especificación sugiere que los exploradores "verifiquen" la solicitud, solicitando métodos soportados desde el servidor con un método de solicitud HTTP OPTIONS, y luego, con la "aprobación" del servidor, enviar la verdadera solicitud con el método de solicitud HTTP verdadero. Los servidores pueden también notificar a los clientes cuando sus "credenciales" (incluyendo Cookies y datos de autenticación HTTP) deben ser enviados con solicitudes.

Con respecto al frontend, también puede solucionarse este problema añadiendo en la peticion las cabeceras de la siguiente manera:

```
let options = {
  headers: {
   'Content-Type': 'application/json'
  }
 };
var url = 'https://localhost:44394/api/avisos/AddAvisoItems';
return new Promise(resolve => {
 this.Http.post(url,JSON.stringify(datos),options)
   .subscribe(data => {
    resolve(data);
   });
});
```

5.2.Importación de entidades desde archivo .csv

Como hemos visto, en la parte del servidor podemos crear las entidades desde el contexto. Esto lo hacemos manualmente añadiendo valores a las entidades. Por ejemplo:

```
modelBuilder.Entity<Provincia>().HasData(
    new Provincia { Provincia_id = "01", Nombre = "Huelva" },
    new Provincia { Provincia_id = "02", Nombre = "Sevilla" },
    new Provincia { Provincia_id = "03", Nombre = "Cadiz"}
    );
```

Pero si se trata de un conjunto de datos con muchos datos se convierte en una tarea ardua. Entonces mi propósito fue exportar esos datos desde un archivo .csv que contiene todos los datos. Pero ¿cuál fue el problema?

Instalé el paquete necesario para realizar la exportación llamado *csvHelper.* Al importar los datos de la siguiente manera:

```
var reader = new StreamReader("Domain/SeedData/provincias.csv");
var csv = new CsvReader(reader, System.Globalization.CultureInfo.InvariantCulture);

csv.Configuration.HasHeaderRecord = false;
csv.Configuration.MissingFieldFound = null;

while (csv.Read())
{
        var records = csv.GetRecords<Provincia>();
        modelBuilder.Entity<Provincia>().HasData(records);
}
```

El contexto me crea las entidades con esos valores, pero el orden es importante. Primero me creaba las localidades y seguidamente las provincias. Se exportan los datos, pero no se realiza adecuadamente ya que si no hay provincias no se pueden realizar las relaciones de las provincias con las localidades. Por falta de tiempo y la mayor importancia de otras cuestiones decidí seguir introduciendo los valores de las entidades creandolas en el contexto y no exportandolas desde el .csv. Encontré la cuestión del problema pero no la solución. Aún así he decidido mostrarlo en este apartado de errores ya que me parece interesante que se conozca aun sin haberse resuelto.

6.Mejoras futuras

6.1.Caducidad de notificaciones

Debido a que solamente he realizado peticiones GET y POST, una opción para utilizar la peticion DELETE es en la eliminación de las notificaciones.

Como mejora en el futuro creo que es importante la caducidad de las notificaciones para que no se acumulen una vez que se han solventado. La idea es que después de cierto tiempo, quizás unos tres días que es la media en la que se puede reponer un producto, las notificaciones caduquen y se eliminen.

6.2.Reportar notificaciones

La eliminación de las notificaciones pueden hacerse por su caducidad, como ya he explicado, o con la ayuda de los usuarios si la notificación realizada por otro usuario no es correcta. Para ello cuando vemos las notificaciones, debajo a su derecha e incorporado un botón para realizar una reportación. Entonces, cada “x” reportaciones la notificación pasa automáticamente a ser eliminada.

6.3.Control de sesiones

No he visto importante la incorporación de un sistema de sesiones, pero podría implantarse. Al utilizarse el usuario tendrá la posibilidad de modificar la notificación que ha creado en caso de que ésta estuviese incorrecta o tenga un error. Además podría recibir en la misma aplicación notificaciones de la existencia de un producto que esté buscando.

6.4.Notificar de existencias

Sería interesante que, ya que podemos ver el producto que está agotado, nos avisen de la disponibilidad del mismo. Esto podría hacerse mediante correo electrónico, recibiendo un correo cuando haya existencia del producto que buscamos.