



# Informe de Entrevista de PedidosYa

Prueba Técnica Backend

Entrevistada: Verónica Seoane.

Fecha: 28-06-2018

# Índice

|   |           |
|---|-----------|
| <b>Índice</b>   | <b>2</b>  |
| <b>Qué se necesita para probar?</b>                             | <b>3</b>  |
| <b>Qué hay en el repositorio?</b>                               | <b>3</b>  |
| <b>Por qué se eligió la tecnología utilizada?</b>               | <b>3</b>  |
| <b>Publicación local del servicio para probarlo con Postman</b> | <b>4</b>  |
| <b>Configuración en el Web Config</b>                           | <b>5</b>  |
| <b>Probar la aplicación</b>                                     | <b>7</b>  |
| Flujo de prueba - Caso de prueba básico                         | 8         |
| <b>Cosas que se hicieron bien</b>                               | <b>9</b>  |
| <b>Cosas que podría mejorar</b>                                 | <b>10</b> |
| <b>Comentarios (Cómo me sentí?)</b>                             | <b>11</b> |
| <b>Anexo</b>  | <b>12</b> |
| <b>Activar IIS en Windows para publicar el servicio</b>         | <b>12</b> |

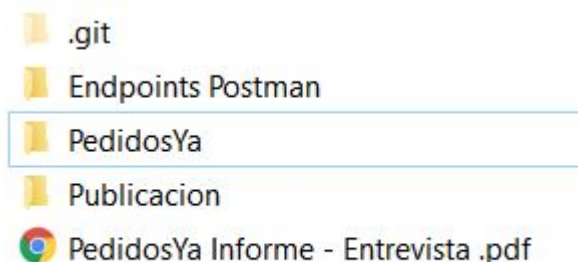
## Qué se necesita para probar?

En primera instancia:

- IIS Activado
- Postman
- Microsoft SQL Server Management
- Visual Studio

Voy a intentar de conseguir publicarlo en algún sitio de la nube para que no haya que instalar todo.

## Qué hay en el repositorio?



En la carpeta Endpoints Postman se encuentra el environment ( donde se define el ip y el puerto ) y la collection con todos los requests para probar la WebAPI.

En la carpeta PedidosYa se encuentra el código en C#.

En la carpeta Publicacion se encuentran los archivos necesarios para publicar el servicio en el iis.

PedidosYa Informe - Entrevista es este documento.

## Por qué se eligió la tecnología utilizada?

Elegí C# ya que es un lenguaje que tiene gran soporte en la comunidad (especialmente en mi sitio favorito StackOverflow).

Además ofrece un montón de soluciones a través de su paquete NuGet.

En mi trabajo desarrollo con esta tecnología por lo que es el lenguaje donde tengo más práctica y ya tenía todo lo necesario instalado en mi notebook.

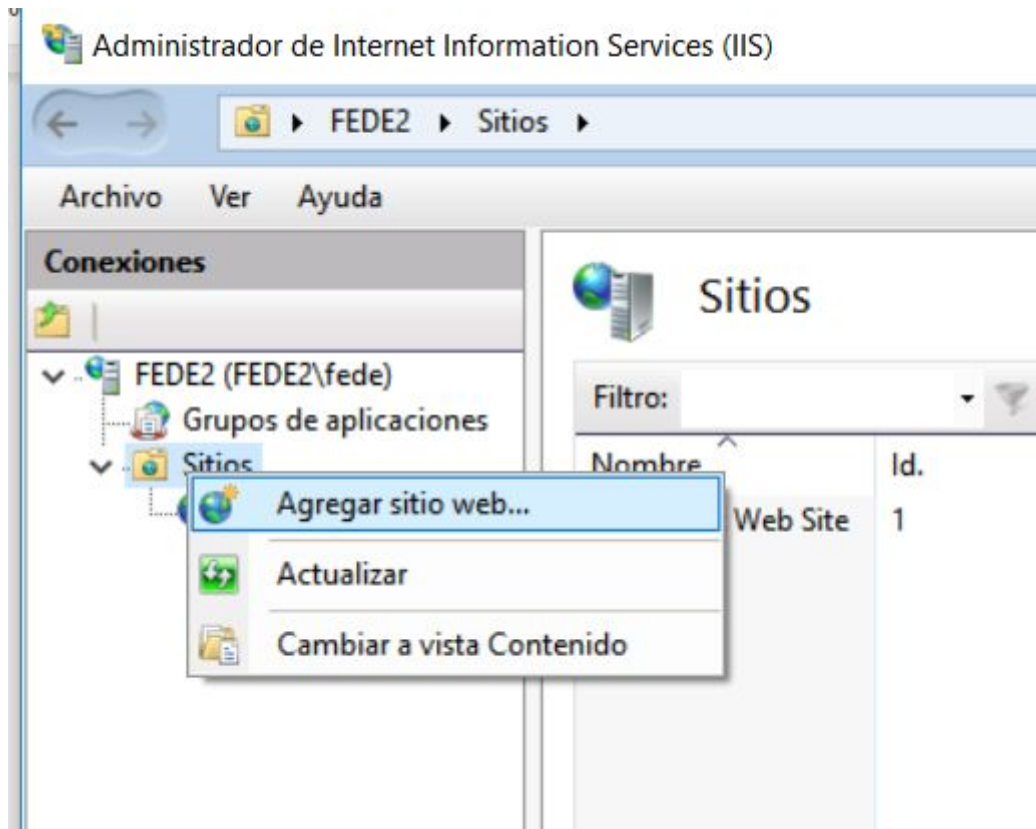
Elegí Entity Framework Code First porque pensé que era la forma más portable, pero después me di cuenta que para probarlo en ambiente de desarrollo van a necesitar instalar herramientas extras. No soy muy fan del Code First, ya que pierdes gran control de la base de datos, ejemplo el crear índices, etc.

# Publicación local del servicio para probarlo con Postman

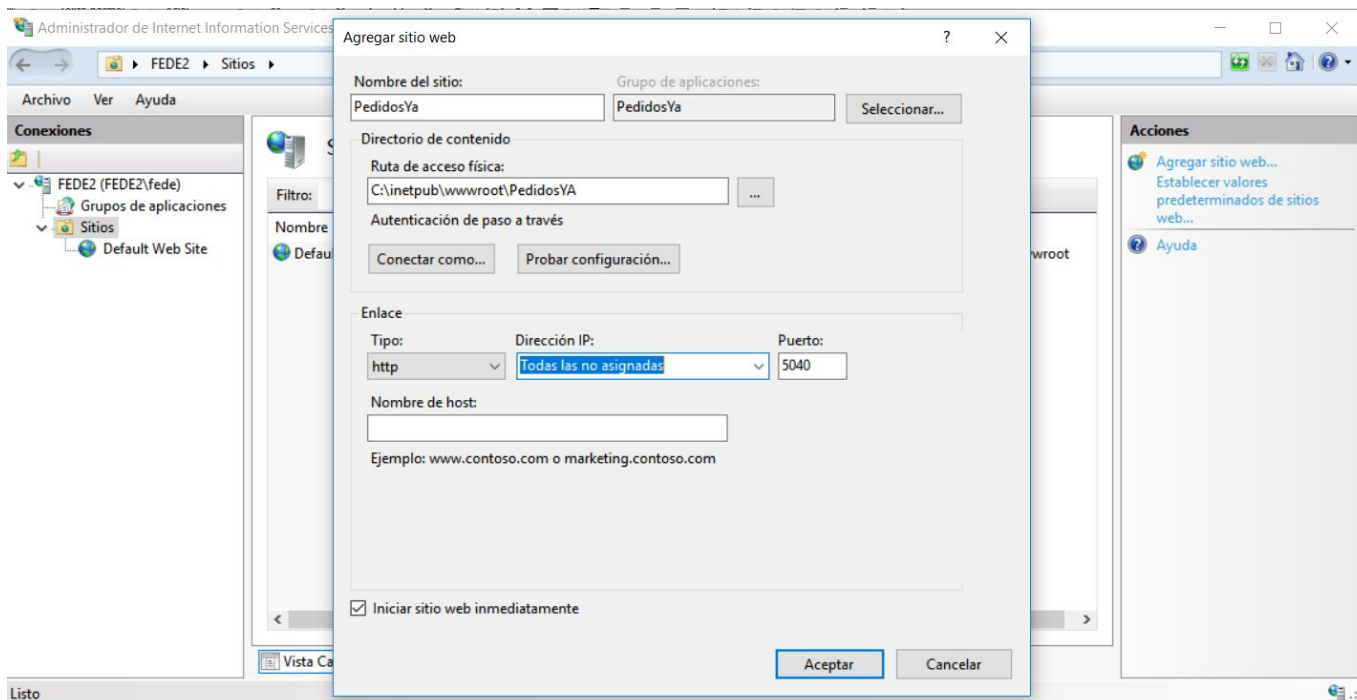
Para configurar el IIS en la PC local: [Ir a Anexo - Activar IIS para publicar el servicio](#)

Una vez activado el IIS, se procede a hacer la publicación.

1) Ingresar al IIS



2) Se agrega el sitio



Luego copiar en ese sitio lo que está dentro del repositorio en la carpeta “Publicacion”.

## Configuración en el Web Config

Allí se puede encontrar las siguientes variables:

Country indica el país en este caso se ingresó 1 (por el ejemplo que se proveyó de la API ).

La idea es que se pase a través de la request.

Por otro lado está la variable “maxResultsFromCompetitors” que es la mayor cantidad de competidores que se quiere retornar, por letra se indicó 3, pero se parametrizó para poder modificarlo sin tener que re compilar el código para cambiarlo.

“userToGetToken” y “passwordToGetToken” es para indicar user y password para acceder al token dentro de la API provista.

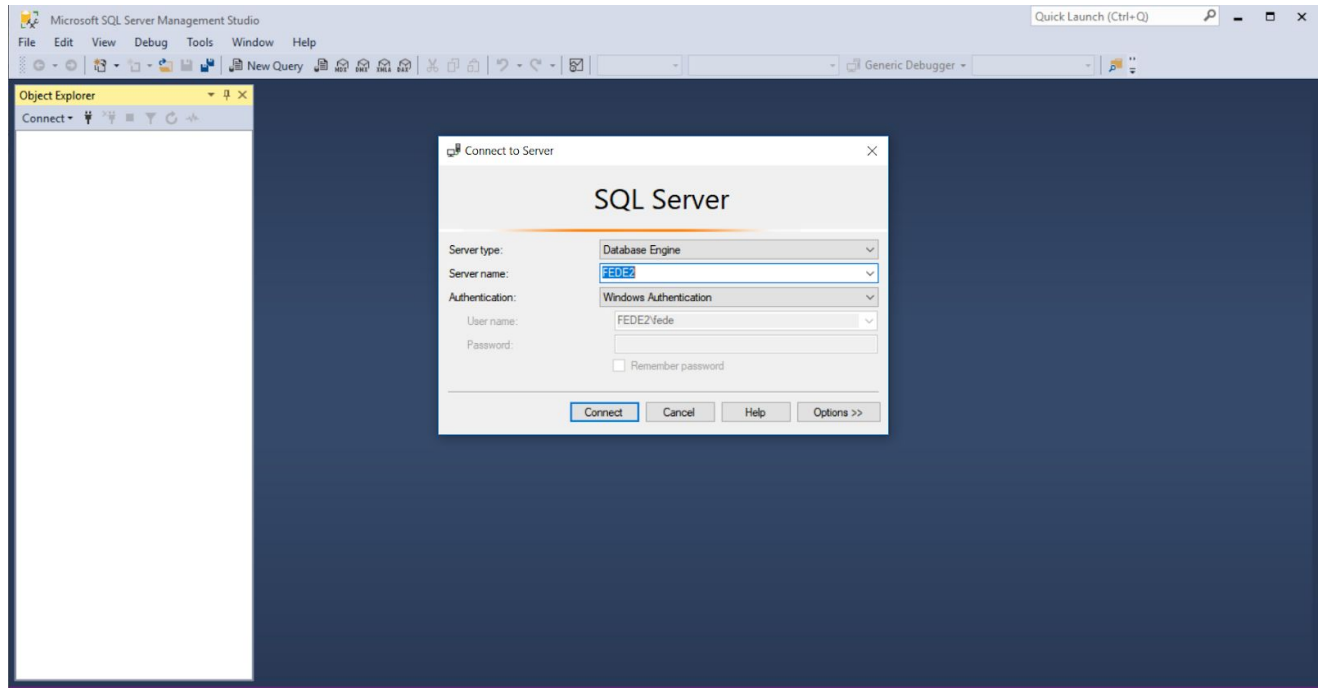
**Conclusión:** Lo único que hay que agregar para que funcione en esta parte es la password provista: PeY.....

```
<appSettings>
  <add key="country" value="1" />
  <add key="maxResultsFromCompetitors" value="3" />
  <add key="userToGetToken" value="test" />
  <add key="passwordToGetToken" value="PeY" />
</appSettings>
```

Por otro lado, una parte importante a modificar es en nombre del Server Name de SQL Server en el connectionString:

```
<connectionStrings>  
  <add name="PedidosYa" connectionString="Data  
Source=NombreDelServerNameDeSQLServer;Initial Catalog=PedidosYa;Integrated  
Security=true" providerName="System.Data.SqlClient" />  
</connectionStrings>
```

Para ello, una de las formas es fijándose en el Microsoft SQL Server Management:  
En mi caso es Fede2




Con eso alcanzaría para que la WebApi esté pronta para probar.

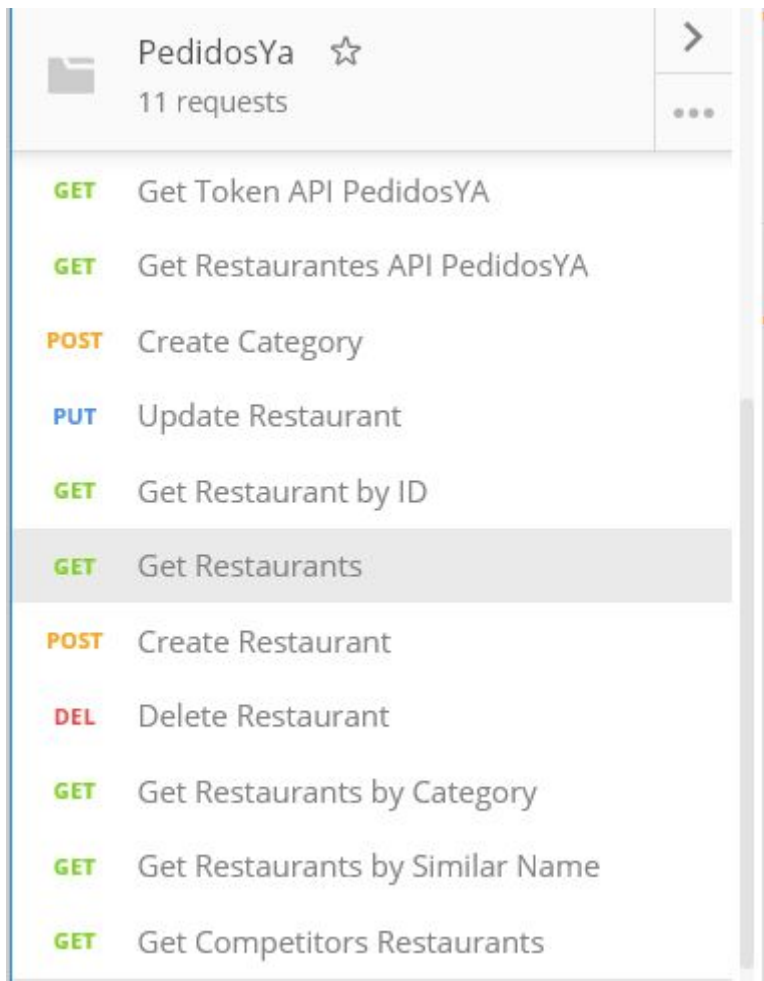
Con el primer request que se haga se creará la base de datos, ya que se utilizó Entity Framework con Code First, por lo que la base de datos se crea a partir del código, por ende luego de que se haga una request.

## Probar la aplicación

Dentro del repositorio hay una carpeta que dice "Endpoints Postman", allí hay una collection donde se encuentran todos los requests para ejecutar todas las funcionalidades de la WebAPI. Por otro lado se encuentra el environment, que sería el ambiente en Postman que sólo contiene la IP y el puerto para que sea parametrizable en todas las requests.

| arios > fede > Documentos > PedidosYa > Endpoints Postman  |                      |           | ▼ ↺ |
|--|----------------------|-----------|-----|
| Nombre   | Fecha de modifica... | Tipo      |     |
|  PedidosYa.postman_collection.json  | 28/06/2018 9:27      | JSON File |     |
|  PedidosYa.postman_environment.json | 28/06/2018 9:27      | JSON File |     |

Una vez que importan ambas cosas en Postman, van a poder probar la WebAPI. Se van a encontrar algo como:



Donde los dos primeros es para probar el servicio que me proveyeron, el resto es para probar la WebAPI.

## Flujo de prueba - Caso de prueba básico

- 1) Crear una Category llamada "Chivitos" (porque hay competidores con esa categoría, así salen resultados cuando se ejecute Get Competitors Restaurants). Se generará una categoría con id 1.
- 2) Crear otra categoría con cualquier nombre. Ejemplo : Pizzas. Se generará una categoría con id 2.
- 3) Allí se crean dos categorías
- 4) Crear Restaurant con las dos categorías anteriores o sea (1 y 2), tal como está en el request:





Y de nombre “El club de la papafrita”.

Es importante mantener también la latitude y longitude indicada ahí para la búsqueda de competidores.

- 5) Probar crear otro restaurante con el mismo nombre
- 6) Actualizar el restaurante, cambiarle cualquier campo, recordar poner el id del restaurante que se desea modificar en el parámetro del request.
- 7) Crear otro restaurante
- 8) Eliminar el restaurante recién creado.
- 9) Mostrar todos los restaurantes y ver que el eliminado no está.
- 10) Realizar el Get by id de un restaurante en particular
- 11) Buscar restaurantes por Categoría con id 1 (Chivitos)
- 12) Crear un restaurante con nombre parecido al primero que se creó
- 13) Buscar por la palabra “pap”, ver que aparece “El club de la papafrita”
- 14) Buscar competidores por el restaurante 1 que tiene como categoría a chivitos y longitud y latitud igual a competidores con esa categoría.

## Cosas que se hicieron bien

- Se parametrizó diferentes variables para no tener código hardcodeado. Ejemplo: Country, cuyo valor se puso en el app config.
- Se usó un diseño separado en capas para separar por comportamiento y no por funcionalidad (Por un lado las controllers, por otro la lógica, etc).
- Se programó en inglés

- Se intentó separar y que hayan responsabilidades bien definidas. El repository se debe de encargar del manejo con la base de datos. La controller de recibir los datos y delegar la lógica al paquete de businessLogic
- Se logró terminar la gran mayoría de los puntos (faltó el paginado)
- Se siguió con las buenas prácticas de codificación.
- Se intentó re utilizar la mayor cantidad de código posible. (Igual sé que podría mejorar este punto)

## Cosas que podría mejorar

- Crearía dos modelos uno tipo DTO para transportar los datos que llegan desde el controller hacía las capas de abajo y otro de entidades para la base de datos.
- La búsqueda de nombres similares de restaurantes no la haría case sensitive.
- Haría que la búsqueda de categorías entre los competidores y el restaurante indicado tenga un algoritmo más inteligente y que detecte que pizzas es lo mismo que pizza o Pizza.
- Cargaría en un caché las categorías (ya que no son tantas) para que cuando consulte si las categoría existe no tenga que ir hasta la base de datos. Dando hincapié en la Performance.
- Separaría las capas con interfaces, para poder hacer mocks en los tests y desacoplar de la implementación. Haciendo noción al atributo de calidad de modificabilidad-mantenibilidad.
- Ampliar este servicio monolito a uno de microservicios para que sea extensible/escalable.
- Pondría un manejo adecuado de control a la hora de crear un restaurante, que valide que la hora este en un rango razonable, que la longitud y latitud sean válidos,
- Parametrizar el lenguaje del texto para que soporte multi idiomas, a través de un resource con mensajes en diferentes lenguajes donde se setee el lenguaje de la web api y los mensajes se adapten.
- Más controles en el manejo de excepciones, agregar más excepciones propias y definir las mejor a lo largo del código.
- Crearía una propia base de datos, no usaría Entity Framework Code First, crearía índices para agilizar el acceso. Si los restaurantes van a ser en gran volumen, pensaría en tener otra base de datos auxiliar (alguna columnar) para que cuando se consulte sea más rápido.
- También agregaría índices a los nombres de restaurantes para que la búsqueda de nombres similar sea más rápida o haría uso de una base de datos columnar o de rápido acceso (pensaría en opciones como Elasticsearch ya que es búsqueda de texto, etc).
- Haría un ABM de Categorías. Ahora simplemente se implementó el alta

- No traería los datos a memoria de la API provista con los competidores, tendría que pensar cómo haría para que esto fuera más performante.
- Pondría la password encriptada dentro del app config
- Mejoraría el manejo de excepciones que se tiran para afuera, en el caso de .NET usaría FaultExceptions bien definidos, con una códiguera bien definida para cada tipo de error.
- Aplicaría el patrón Repository para que se reutilice código de acceso a la base de datos en caso de que sea escalable esta API.
- Haría diagramas mostrando cómo fue la arquitectura y diseño elegido, lo analizaría y buscaría otras mejoras.
- Usaría un chequeador de código para que sea más prolijo y me permita seguir con las buenas prácticas de codificación.
- Crearía UNIT TEST!!! Son re importantes. Trataría de llegar a por lo menos el 80 % de cobertura de código.
- Usaría Servicio asincrónico para la llamada de la api y un manejo de excepción específico en caso de que esa api externa no estuviera disponible.
- Pondría un control de sobrecarga para evitar ataques o saturación de la base de datos. (Podría ser una especie de buffer que no permita más de tantos requests en cierto tiempo, y que el sistema vaya procesando cada request almacenada en dicho buffer)
- Disponibilizaría un Ping Echo para que se testee si el servicio está disponible.
- Agregaría seguridad, obligando el acceso a la api mediante token.
- Me hubiera gustado tener más tiempo para sentarme y analizar qué otros patrones se podrían haber usado.
- Exponer una justificación del diseño elegido.

## Comentarios (Cómo me sentí?)

Bueno me gustó mucho la idea de que me hagan hacer una prueba como parte de la entrevista, creo que es una muy buena idea. Lo re disfruté haciéndolo. Lo único que no me gustó es el poco tiempo que tuve para poder hacerlo, quizá en el plano personal no fue el mejor momento para ponerme hacer una prueba de este tipo ya que justo estoy en la última semana de clases terminando 5 obligatorios a la vez, estudiando para parciales, más el trabajo, más la tesis, más temas personales, creo que eso me jugó en contra porque podría haberle sacado más provecho a esta prueba y creo que podría haber entregado algo de mayor calidad si hubiera tenido más tiempo.

## Anexo

### Activar IIS en Windows para publicar el servicio

- 1) Ir a Panel de Control
- 2) Ir a Programas y Características
- 3) Activar o desactivar las características de Windows
- 4) Seleccionar el Internet Information Services:

