Crear una WebAPI y desplegarla en Azure – Guía paso a paso

**Introducción**

En esta guía vamos a indicar cómo construir una Web API con Visual Studio desde cero. En concreto, la API de este ejemplo nos permitirá manipular una lista de servicios de Azure, permitiéndonos añadir nuevos servicios, eliminarlos o modificar sus detalles. Una vez construida, veremos cómo desplegarla en Azure para poder empezar a utilizarla.

Una Web API es un servicio que permite manipular datos haciendo uso de las características que ofrece el protocolo HTTP. Es por ello que el cliente que haga uso de ella puede ser cualquiera que soporte dicho protocolo.

**HTTP**

**HTTP**

**HTTP**

**HTTP**

Te recomendamos que eches un vistazo a los conceptos de API, REST, y HTTP; para tener una ligera idea de lo que estás construyendo. Te recomendamos [este artículo](http://blogs.msdn.com/b/martinkearn/archive/2015/01/05/introduction-to-rest-and-net-web-api.aspx) que hace un repaso a todos ellos. Si prefieres indagar más en Web API, en la Microsoft Virtual Academy puedes encontrar un genial [curso introductorio](https://www.microsoftvirtualacademy.com/en-us/training-courses/web-api-design-jump-start-8689?l=BMxSvaH1_3804984382).

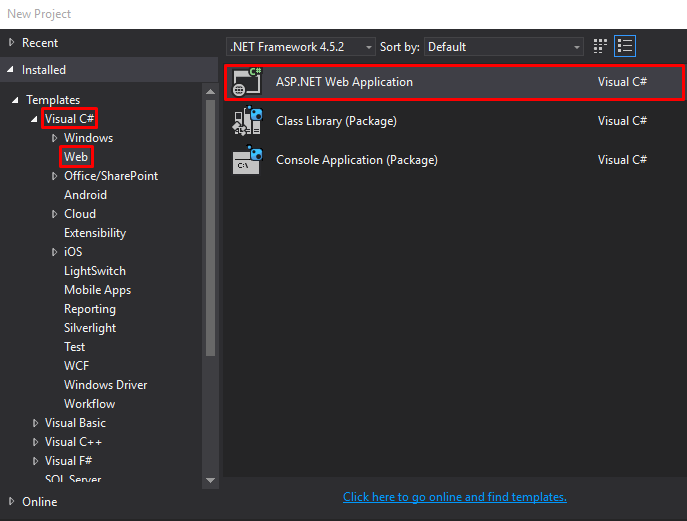
**Requisitos**

Para poder seguir el tutorial necesitas disponer de:

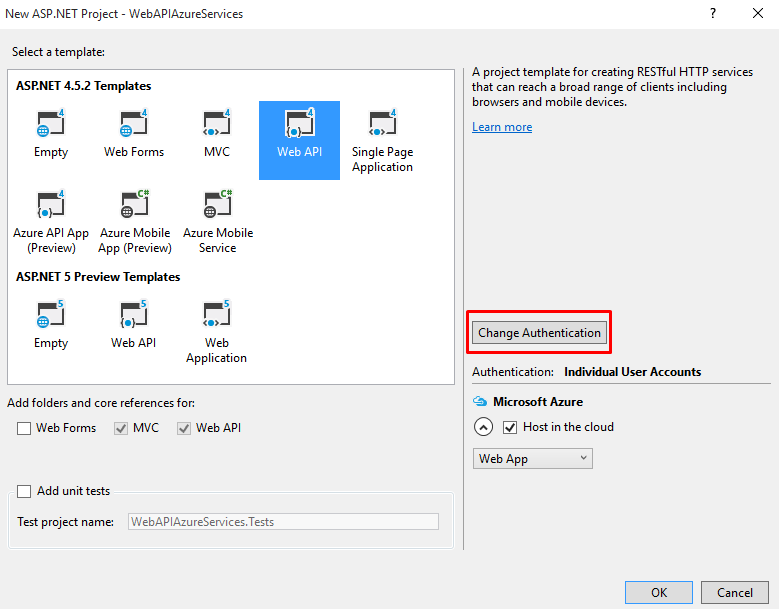
* Visual Studio 2015. La [versión Community](https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx) te sirve perfectamente y no tiene ningún coste.
* Una cuenta en Azure. El tutorial tiene como objetivo desplegar la API en Azure, por lo que necesitarás una cuenta para ello. Puedes utilizar [la suscripción asociada a tu DreamSpark](https://azure.microsoft.com/en-us/pricing/member-offers/dreamspark-students/).

**Comienzo**

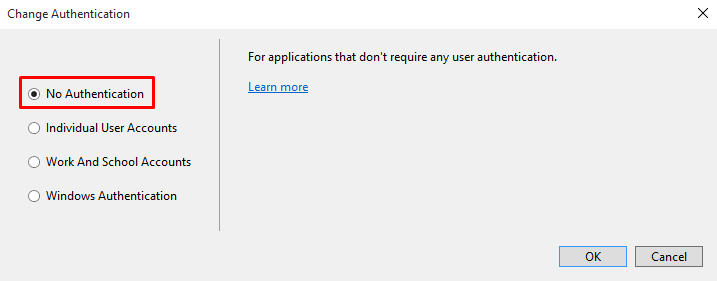
Para comenzar, debemos abrir el Visual Studio y crear un nuevo proyecto. Utilizaremos la plantilla *ASP.NET Web Application*. Le damos el nombre que deseemos al proyecto y pulsamos *OK*.



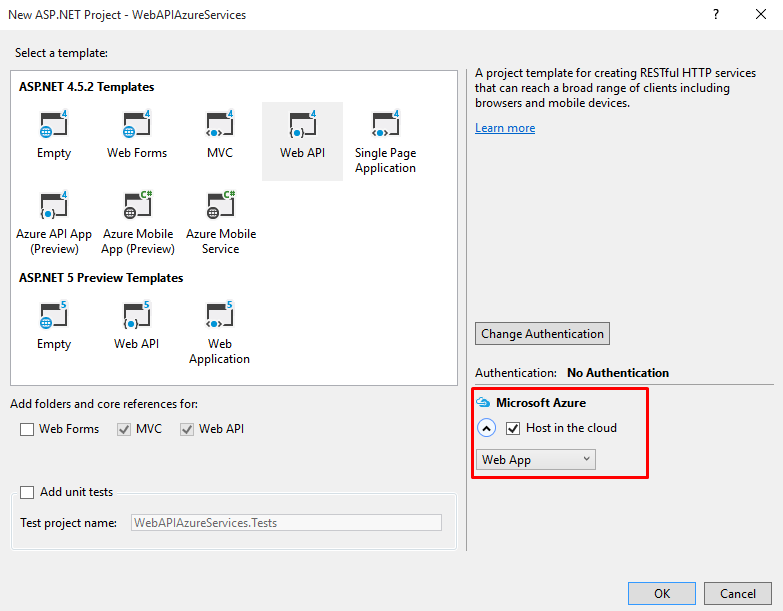
Existen múltiples frameworks que podremos utilizar para hacer una aplicación web de ASP.NET. En nuestro caso, escogemos *Web API* y pulsamos en *Change Authentication*.



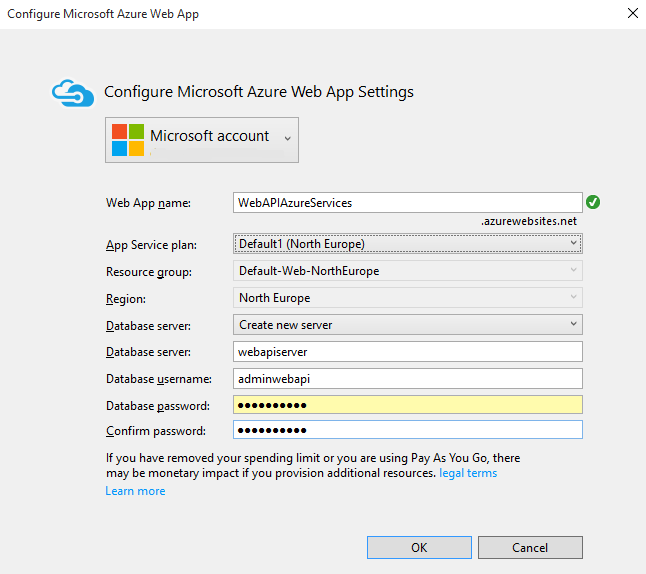
Escogemos *No Authentication*, ya que para este ejercicio crearemos una API pública,y pulsamos en *OK*.



Nos aseguramos de marcar la opción *Host in the cloud* y pulsamos *OK*.



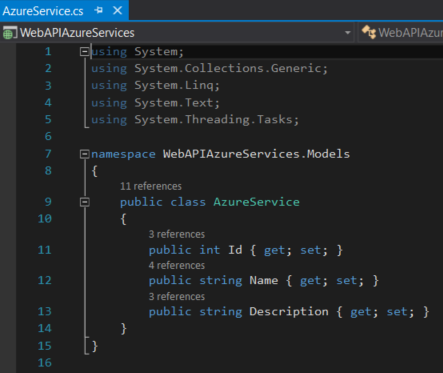
En este momento se nos pedirá que introduzcamos nuestras credenciales de Azure, y nos aparecerá una pantalla de configuración de la que será nuestra aplicación web. Si no tenemos ningún App Service Plan, se nos da la opción de crear uno nuevo escogiendo la opción “Create new App Service Plan”. Hacemos lo mismo con Resource Group y con el servidor de base de datos (Database Server). Hemos escogido North Europe como región ya que era la que menos latencia tenía al momento de crear esta guía. **Es importante que escojamos nombres sean únicos**, ya que sino no se crearán correctamente.



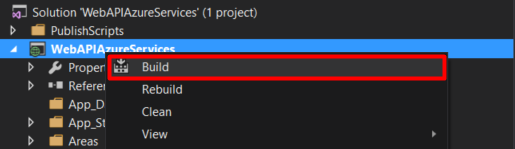
En este momento tendremos nuestro proyecto creado. Dado que queremos crear la vista y los controladores desde cero, borraremos el contenido de la carpeta *Views* y *Controllers*.

Vamos a comenzar creando el modelo para la base de datos. Lo crearemos en la carpeta *Models*. Crearemos una clase: **AzureService**, que modelará un servicio de Azure. La clase será pública. Ambas clases tendrán un ID para identificarlas de manera única en la base de datos.

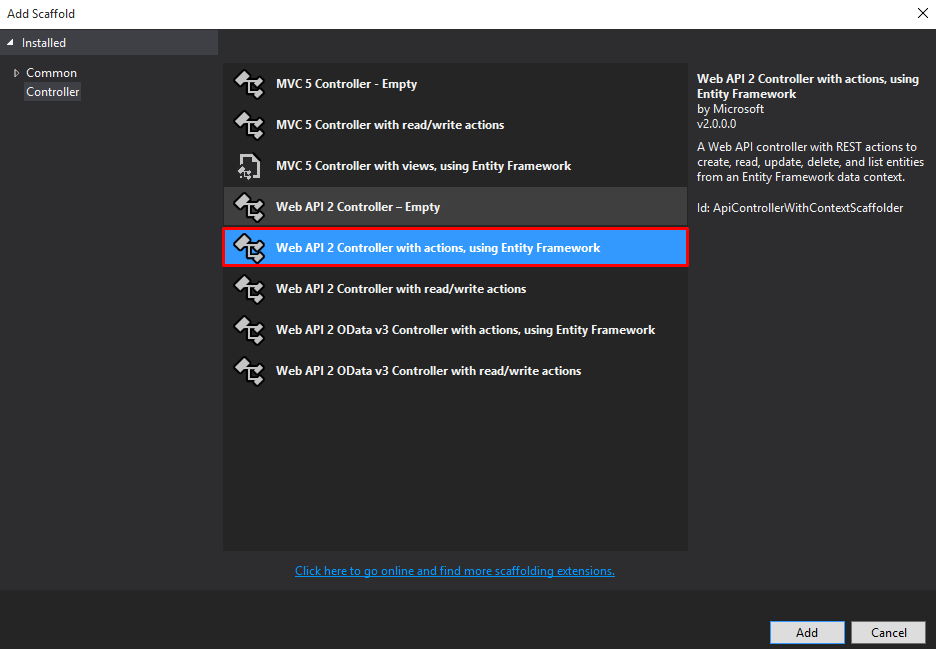
Tendrán el aspecto que se muestra en la imagen:



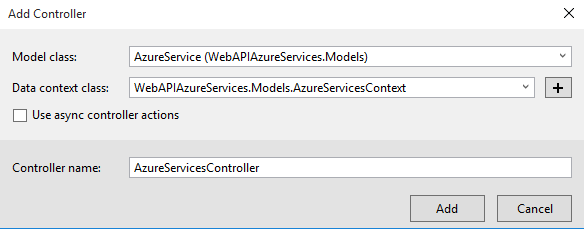
Antes de continuar compilamos la solución, haciendo click derecho en ella y dándole a *Build*.



Ahora añadiremos el controlador. Para ello, hacemos click derecho en la carpeta *Controllers*, y le damos a *Add 🡪 Controller*. Escogemos la opción marcada en la imagen.

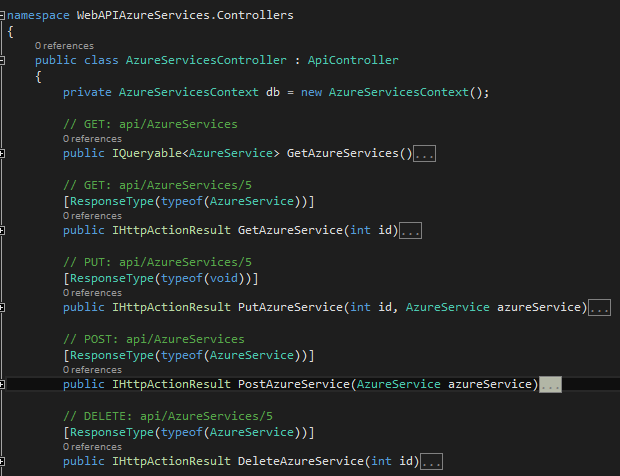


Como modelo escogemos **AzureService**, y creamos un nuevo contexto pulsando en el botón **+** que aparece a la derecha.

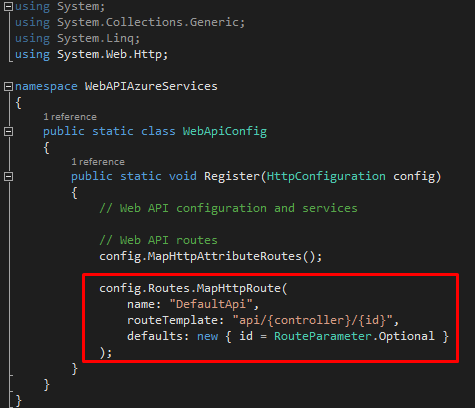


Hacemos click en *Add* y listo. Ya tenemos nuestro controlador preparado con las operaciones implementadas para agregar servicios de Azure, recuperarlos o modificarlos. Vamos a explicar en detalle las operaciones creadas para entender correctamente que es lo que hacen.

En total hay 5 operaciones que responden a peticiones HTTP de los verbos principales: GET, PUT, POST y DELETE.



Para saber la acción que se va a ejecutar, Web API se basa tanto en la configuración de enrutamiento como en una convención que consiste en buscar aquella acción que comience con el nombre del verbo de la petición. Si vemos la configuración de enrutamiento, que se encuentra en el fichero **WebApiConfig.cs** dentro de la carpeta **App\_Start**:



Nos fijamos que, aquellas peticiones que comiencen con *api*, las enrutará al controlador Web API con el nombre *controller*. En nuestro caso, el nombre del controlador sería *AzureServices* ya que la palabra Controller la agrega automáticamente el framework. Por último, el parámetro *id* es opcional, y en caso de ponerse buscará una acción que tenga *id* como parámetro. Se pueden cambiar los enrutamientos si así se desea.

Por otro lado, comentábamos que Web API se basa en una convención basada en los verbos de las peticiones HTTP. Así, si la petición se realiza con el verbo GET, Web API buscará una acción que comience con el nombre Get. No obstante, si queremos dar otro nombre a nuestras acciones, tendremos que usar atributos para enrutar las peticiones correctamente. Estos atributos son *[HttpGet], [HttpPost], [HttpPut]* y *[HttpDelete]*, del namespace *System.Web.Http*.

Por ejemplo, si quisiéramos recuperar todos los servicios Azure existentes, la URL sería:

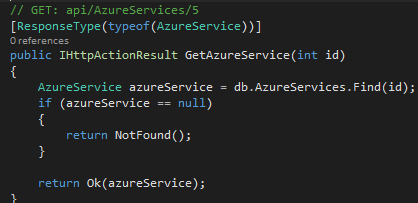
*api/AzureServices*

Si quisiéramos obtener los detalles de un servicio, la URL sería entonces:

*Api/AzureServices/1*

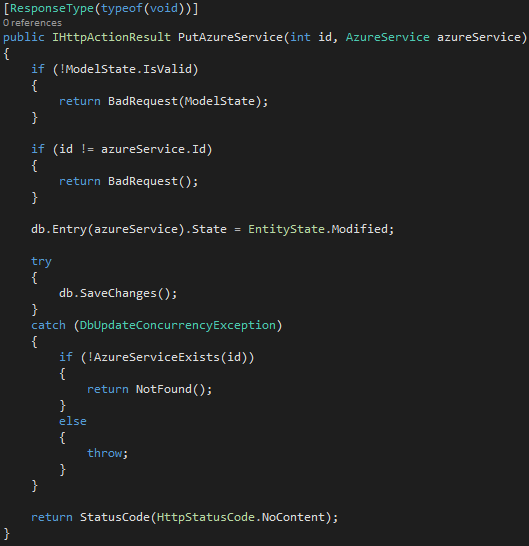
Visto esto, vamos a ver el código de cada operación:

* *GetAzureServices* retorna el listado de todos los servicios Azure que tengamos definidos en la base de datos.
* *GetAzureService* retorna el servicio con el ID que indiquemos. Si nos fijamos, la respuesta es de tipo *IHttpActionResult*. Al momento de retornar, se creará una respuesta de tipo *HttpResponseMessage*. Debido a este tipo de retorno, se declara un atributo *[ResponseType]* indicando el tipo de respuesta que se devolverá en el cuerpo de la respuesta HTTP.



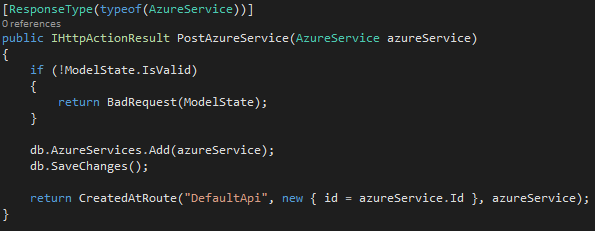
La sentencia *return Ok(azureService)* retornará una respuesta HTTP con código de estado *200 OK* y el servicio en el cuerpo del mensaje.

* *PutAzureService* se encargará de modificar un servicio de Azure. Para ello se le pasa el ID del servicio a modificar y el servicio con las propiedades modificadas.



Si nos fijamos, no retornará contenido, y el código HTTP devuelto será *204 No Content*, por lo que el cuerpo de la respuesta irá vacío. Si por algún motivo el objeto *azureService* no tiene las propiedades seteadas correctamente o el ID que se trata de modificar no coincide con el servicio modificado, se devolverá un *400 Bad Request*.

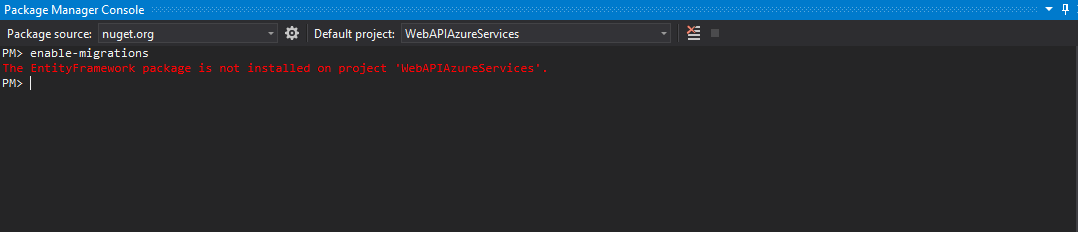
* *PostAzureService* creará un nuevo servicio en la base de datos.



En este caso, se retornará en la respuesta una URL que apunta al nuevo servicio creado.

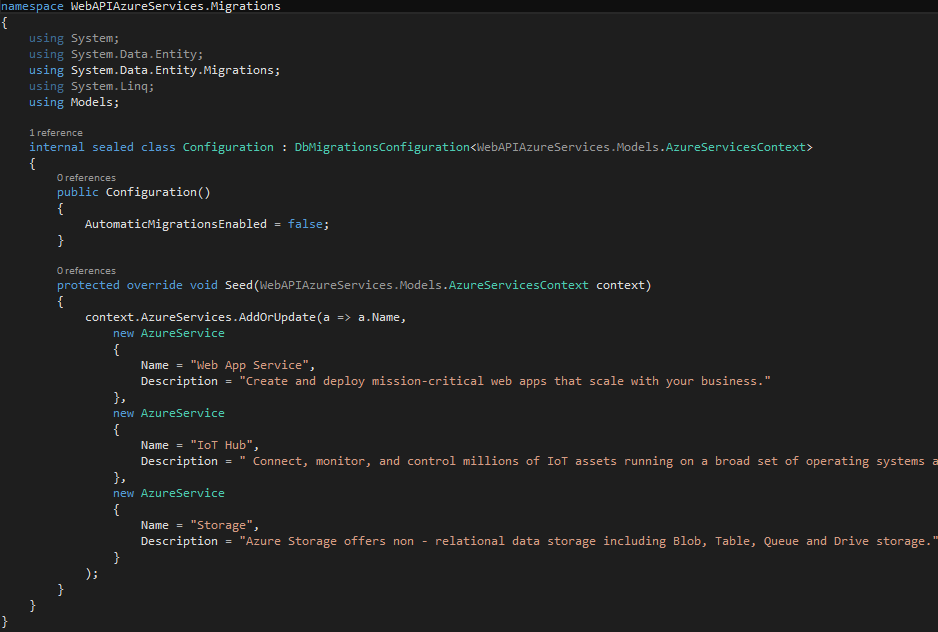
* Por último, *DeleteAzureService* se encargará de eliminar un servicio de la base de datos.

Visto esto vamos a poner tres servicios de ejemplo para empezar a llenar la base de datos. Para ello, abrimos la *Package Manager Console* (*Tools 🡪 NuGet Package Manager 🡪 Package Manager Console*), e introducimos el comando *enable-migrations*. Si nos aparece el siguiente mensaje:



Debemos instalar el *EntityFramework*. Para ello, hacemos click derecho en el proyecto, pulsamos en *Manage NuGet Packages* y lo instalamos.

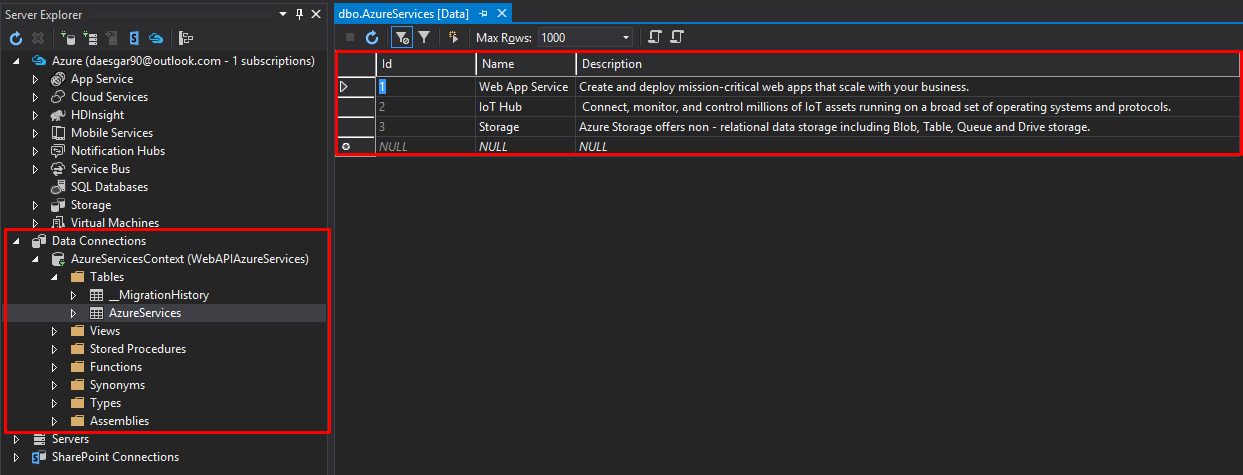
Se nos creará una carpeta llamada *Migrations*, donde se irán guardando las migraciones de base de datos que realicemos, y un fichero denominado **Configuration.cs**. Dentro de dicho fichero existe un método llamado *Seed*. Este método se llamará cada vez que se realice la migración de la base de datos a la última versión, y será el que utilicemos para insertar datos iniciales.



Añadimos tres servicios por el momento.

Creamos la migración inicial, que será la encargada de crear la tabla para los servicios Azure. Para ello, en la *Package Manager Console*, escribimos: *add-migration Initial*. Se nos generará en la carpeta Migrations la migración generada.

Por último, vamos a crear la base de datos. Para ello, introducimos el comando *update-database* (es posible que tarde unos minutos). Si todo ha ido bien, en el *Server Explorer* (*View 🡪 Server Explorer*), veremos el contexto de nuestra BBDD. Además, si hacemos click derecho en la tabla *AzureService* y pulsamos en *Show Table Data*, podemos ver los tres servicios que habíamos puesto en el método *Seed* anterior.



¡Ya tenemos el backend de la aplicación! Ahora falta crear una vista para poder consumir la API (recordemos que la API podríamos publicarla ya a Azure lista para ser utilizada. En nuestro caso crearemos una vista para ver cómo se puede utilizar).

Para ello, hacemos click derecho en la carpeta *Views*, y vamos a *Add 🡪 HTML Page*. En la carpeta Scripts añadimos un fichero Javascript al que enlazaremos en la página HTML.

En el fichero Javascript es donde realizaremos todas las llamadas. Utilizaremos la librería jQuery (ya incluida en el proyecto) para realizar las peticiones asíncronas a la API. Nosotros le hemos llamado **AzureServicesScript.js**, y lo hemos importado en el fichero HTML mediante la etiqueta *<script>*.

La página HTML será muy básica y servirá para demostrar que la API funciona correctamente. Los ficheros tienen el siguiente aspecto:

HTML:



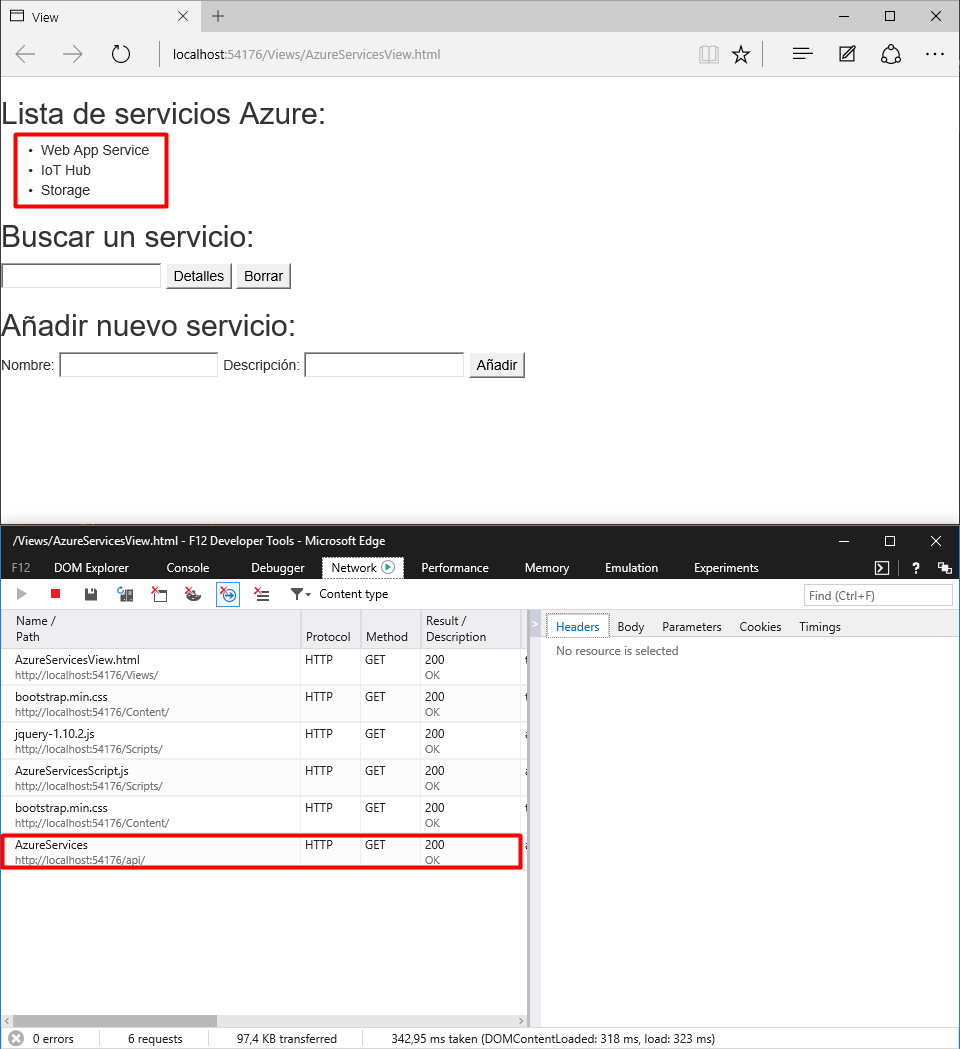
Javascript:



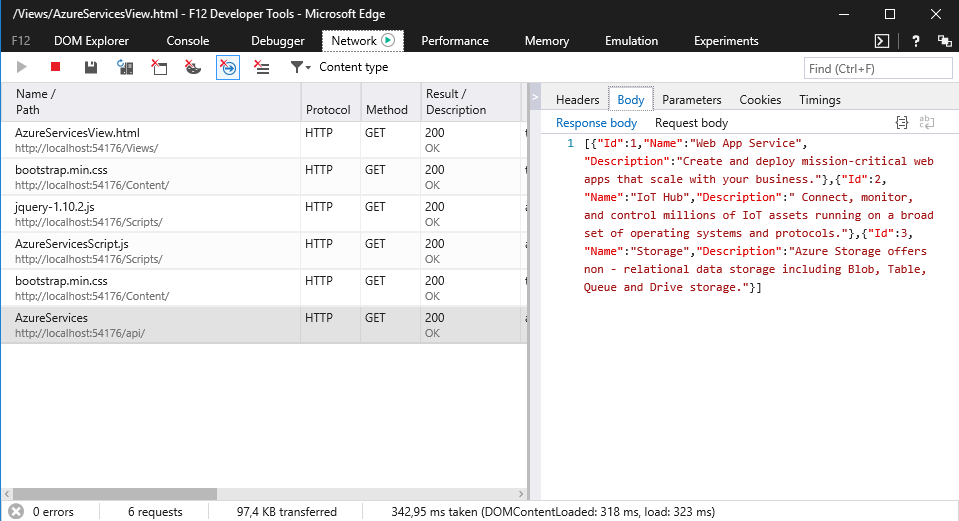


Pulsamos F5 para poner a funcionar la Web API y cargar el HTML. Nos dará un error 404 al entrar dado que en la raíz no tenemos nada. Tenemos que acceder a *Views/AzureServicesView.html*.

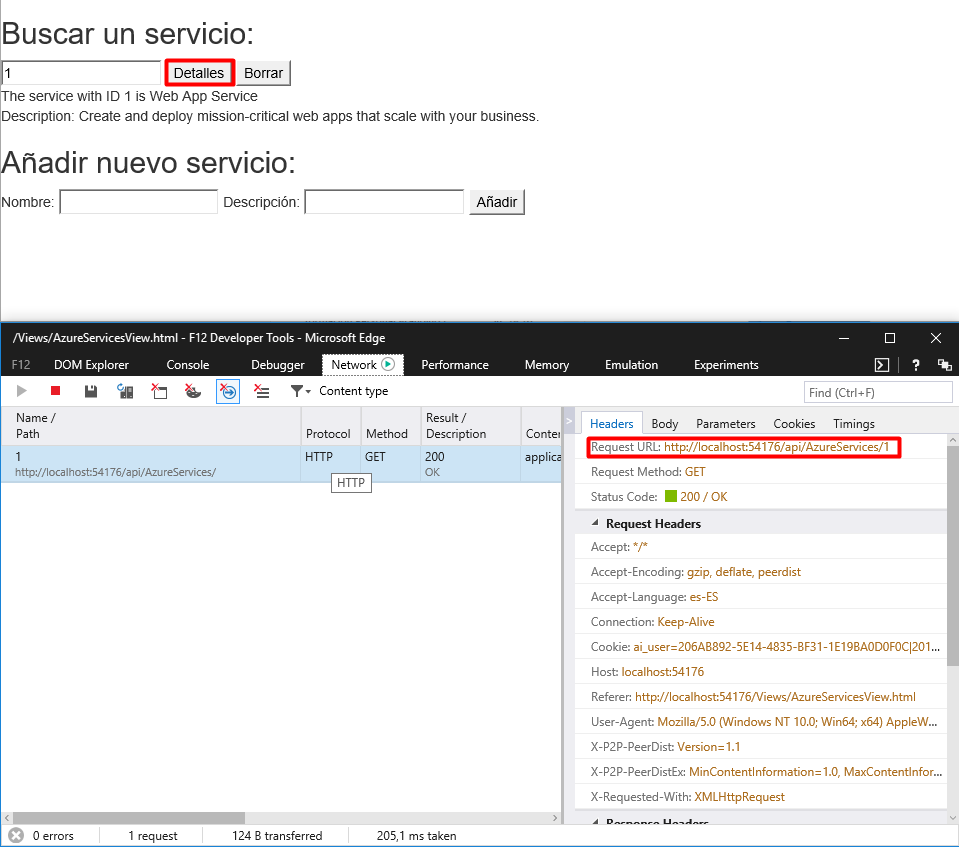
La primera vez que abrimos la página, se genera la primera petición a la API para mostrar el listado de servicios existentes.



Si hacemos click en ella, y vamos a *Body 🡪 Response Body*, podremos ver la respuesta generada por la API.

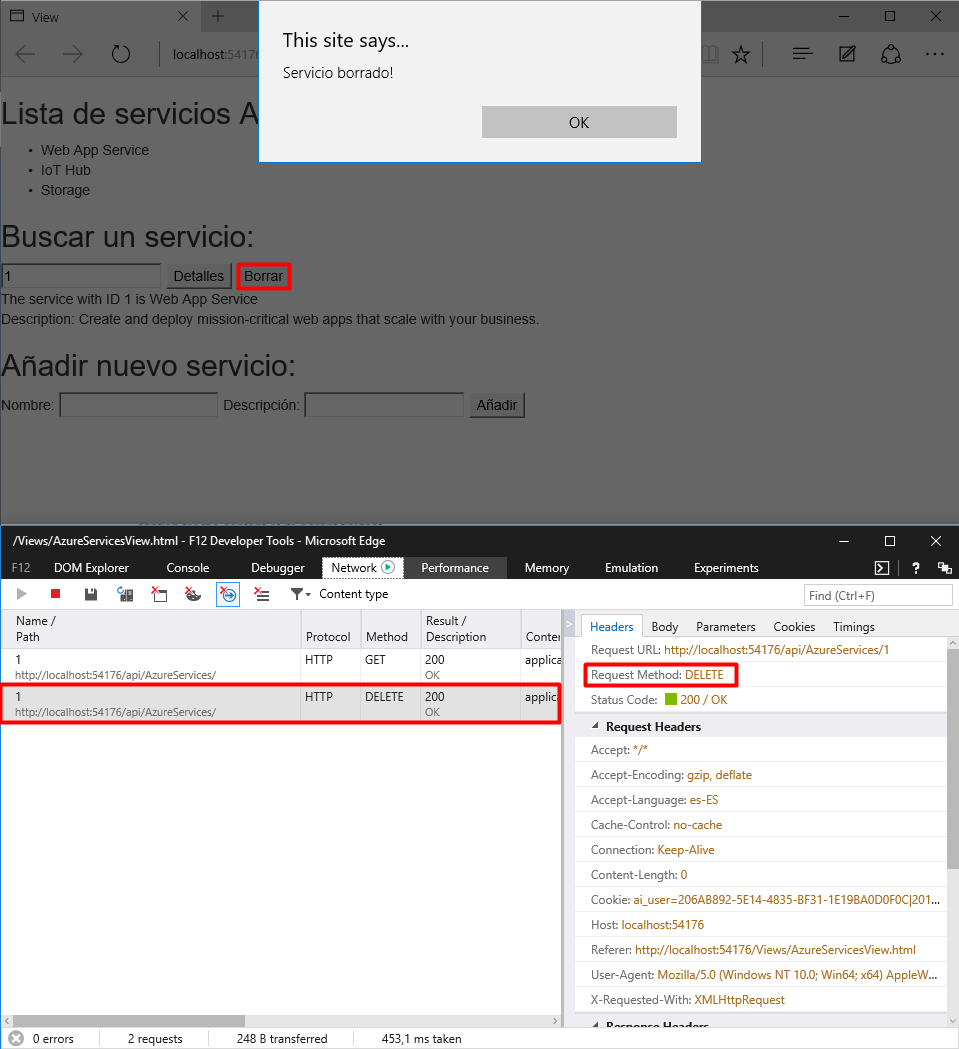


Como vemos es una representación JSON de todos los servicios Azure que se encuentran almacenados en la base de datos. Esa respuesta la procesamos en el método *cargarServicios()* que hemos puesto en el fichero Javascript para luego insertar en la lista los nombres de los servicios únicamente.



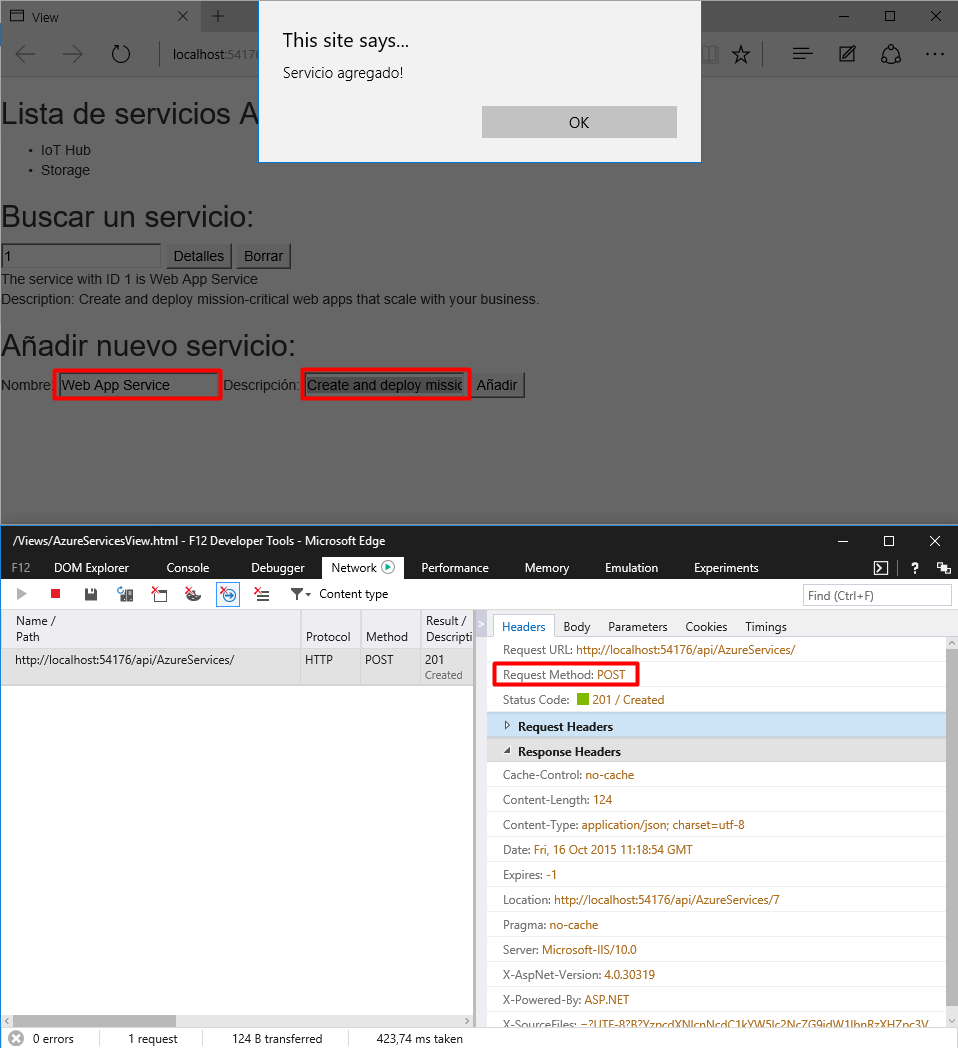
Si queremos obtener el detalle de un servicio determinado, introducimos su ID y le damos a detalle. Se llamará a *detalleServicio()* y se generará la request que vemos en la imagen. Si observamos la cabecera, vemos que la URL incluye el ID al final, por lo que Web API lo enrutará hasta la acción *GetAzureService()* del controlador.

No obstante, si pulsamos en borrar:

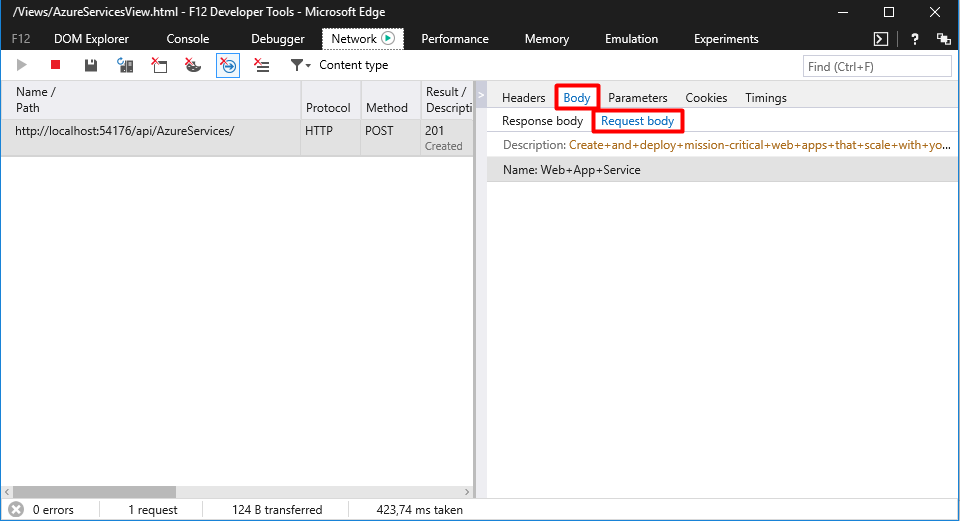


Vemos que la URL es la misma que la generada al pulsar en Detalle, salvo que esta vez el método usado es DELETE.

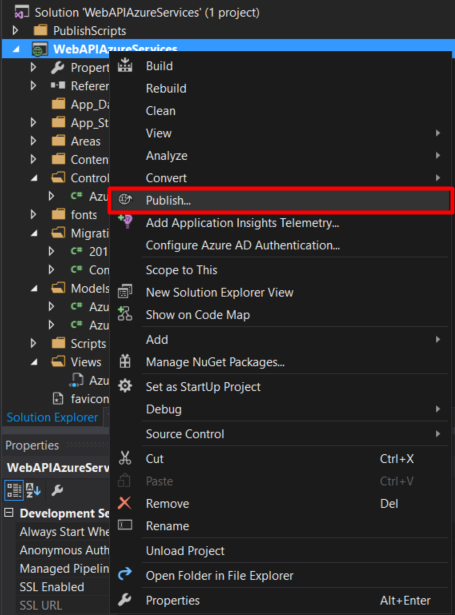
Por último, para añadir un nuevo servicio, introducimos nombre y descripción y pulsamos en Añadir.



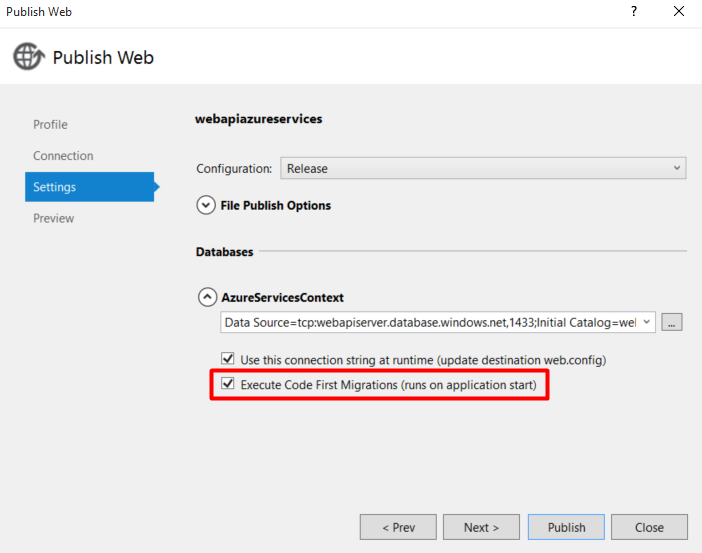
Como vemos, se manda una petición con método POST. Además, como comentamos anteriormente, en la respuesta, vemos que en la cabecera se manda la URL que apunta al servicio Azure recientemente creado (está en la cabecera *Location*). Si queremos ver el detalle de los datos introducidos para crear el servicio, si vamos a *Body 🡪 Request Body*, veremos el nombre y la descripción introducidos.



Comprobado que nuestra API funciona correctamente, es hora de desplegar en Azure. Como anteriormente ya habíamos configurado el servicio de Web App, simplemente tendremos que hacer click derecho en nuestro proyecto y darle a *Publish*.



En la pestaña Settings, marcamos la opción “*Execute Code First Migrations (runs on application start)*”, y le damos a Publish.



Si accedemos a la URL

*/Views/AzureWebServicesView.html*

Si todo ha ido bien, podremos ver la página HTML creada y la API en funcionamiento.

