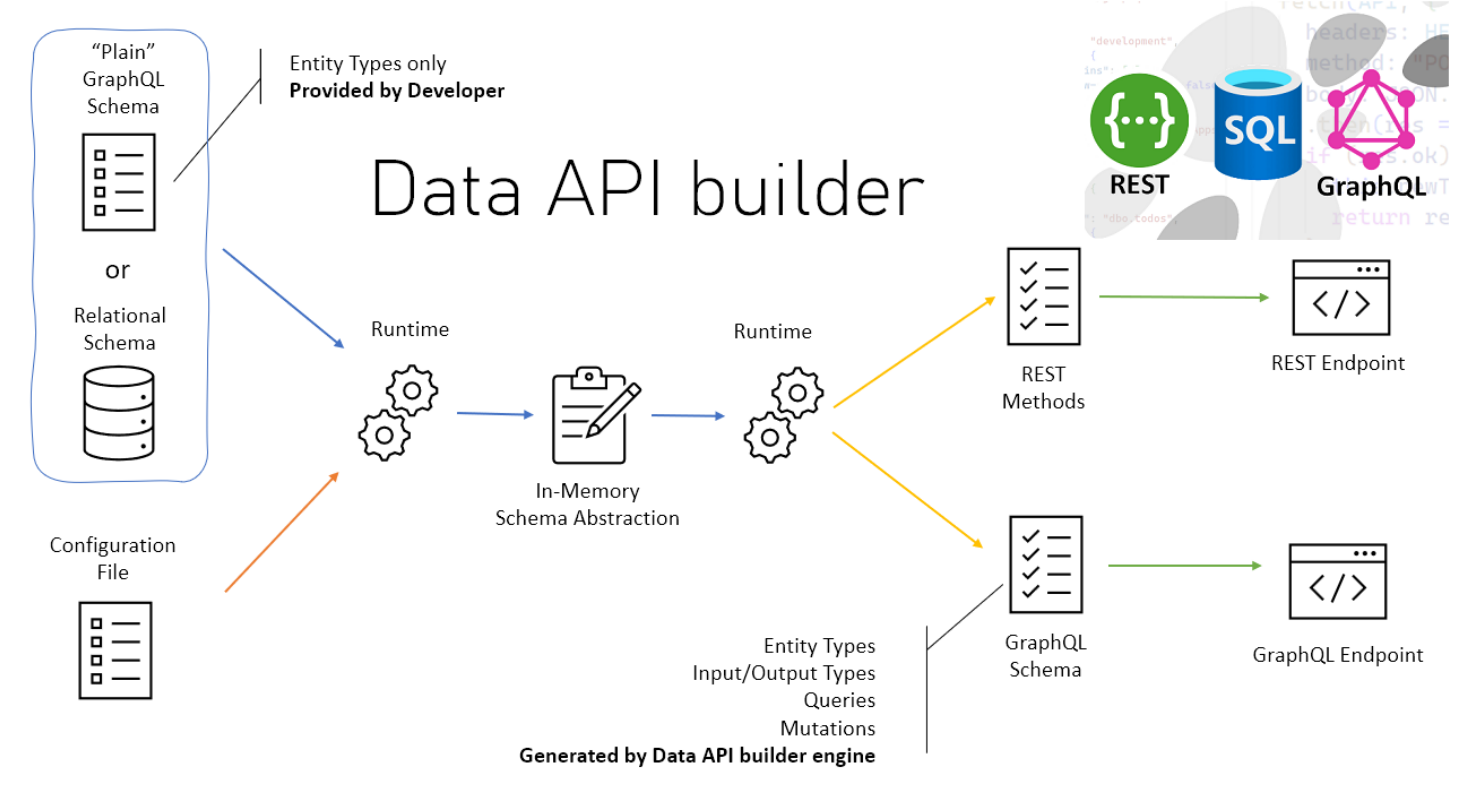
Data API Builder

Convierte de manera instantánea y segura, una base de datos SQL en Azure en una API REST y GraphQL moderna



CONTENIDO

1. INTRODUCCIÓN

El 15 de marzo de 2023, Microsoft anunció una preview pública del Data API Builder. Este Data API Builder te permite convertir de manera instantánea y segura, una base de datos SQL en Azure en una API REST y GraphQL moderna **sin la necesidad de escribir código,** cabe destacar que es un proyecto open source.

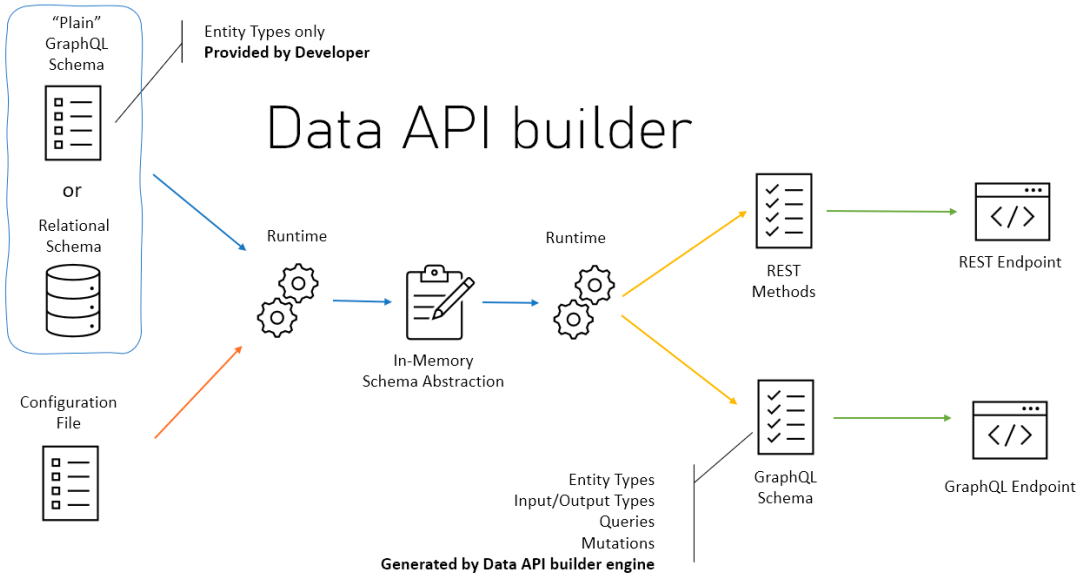


Ilustración 1: Funcionamiento de Data API Builder

Referencias

Documentación

creating-and-securing-a-codeless-rest-api-on-azure-using-data-api-builder

github:data-api-builder

azure-dev-corner

https://learn.microsoft.com/en-us/azure/data-api-builder/rest

2. CARACTERÍSTICAS Y ASPECTOS PARA DESTACAR

General

- Soporta Azure SQL, Cosmos DB, Postgres, y MySQL.
- Las APIs pueden estar hosteadas en AKS, ACI, Static Web Apps y App Services.
- Soporta collections, tablas, vistas, y procedimientos almacenados.
- Desarrollo via CLI dedicado.

Autenticación y Autorización

- Soporta EasyAuth si se ejecuta en Azure.
- Soporta Autenticación via OAuth2/JWT.
- Autorización basada en roles.
- Seguridad de ítems via policy expressions.

REST API

- Soporta operaciones CRUD via POST, GET, PUT, PATCH y DELETE.
- Filtrado, ordenación y paginado.

GraphQL API

- Soporta queries y mutaciones.
- Filtrado, ordenación y paginado.
- Relationship navigation.

Básicamente, el Data API runtime se trata de una una aplicación ASP.NET Core que define un controlador de API REST genérico. Sus rutas, fuentes de datos y entidades se definen externamente a través de un archivo de configuración.

Además, contiene métodos estáticos para generar tipos de objetos GraphQL a partir de definiciones de tablas SQL y los expone mediante el uso de la plataforma ChilliCream GraphQL.

3. CASO PRÁCTICO

Usaremos el Data API Builder para crear una API REST sin código para la famosa base de datos de demo Adventure Works LT.

Crearemos un endpoint para cada una de las tablas junto con 3 roles en Azure AD, cada uno teniendo permisos diferentes.

Posteriormente, hostearemos todo empleando Azure Containers.

4. BASE DE DATOS SQL EN AZURE

En primer lugar, necesitaremos un Azure SQL server para hostear nuestra base de datos de pruebas. Para ello podremos crear directamente desde Azure una SQL Database o mediante Azure CLI. Hay que tener en cuenta que inicialmente será poblada con los datos de la base de datos demo AdventureWorks.

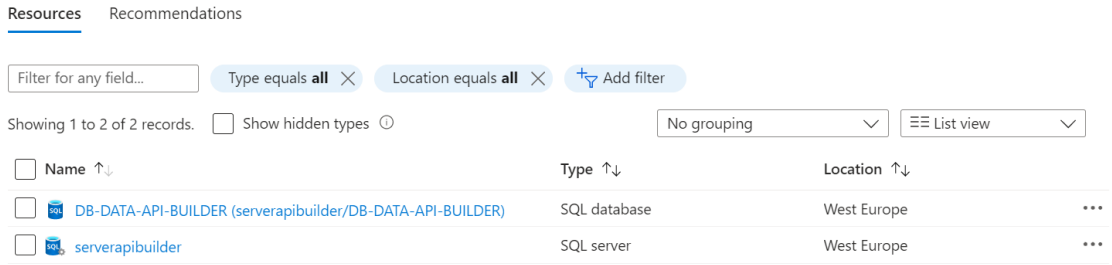


Ilustración 2: Creación de base de datos en Azure

5. CUENTA DE ALMACENAMIENTO EN AZURE Y FILE SHARE

Crearemos también una cuenta de almacenamiento en Azure la cual hosteará nuestro fichero de configuración.

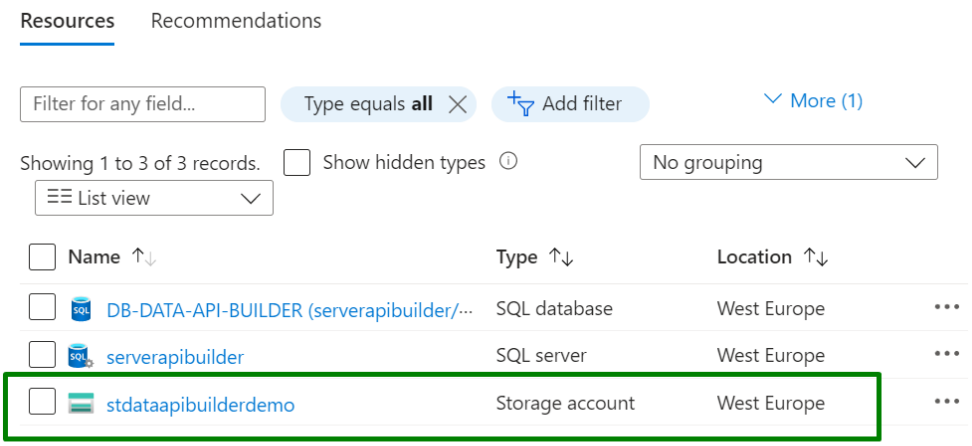


Ilustración 3: Cuenta de almacenamiento de azure

Crearemos también un file share via azure CLI:

```
# Create file share
az storage share create `
  --name dab-config `
  --account-name stdataapibuilderdemo
```

Ilustración 4: Creación de un file share via azure CLI

6. REGISTRO DE LA APP EN AZURE ACTIVE DIRECTORY

El siguiente paso será registrar la app en el Active Directory de Azure para posteriormente poder exponer la API. Para ello iremos al Active Directory → app registrations

Le pondremos un nombre y la registraremos. Posteriormente añadiremos un Application ID URI

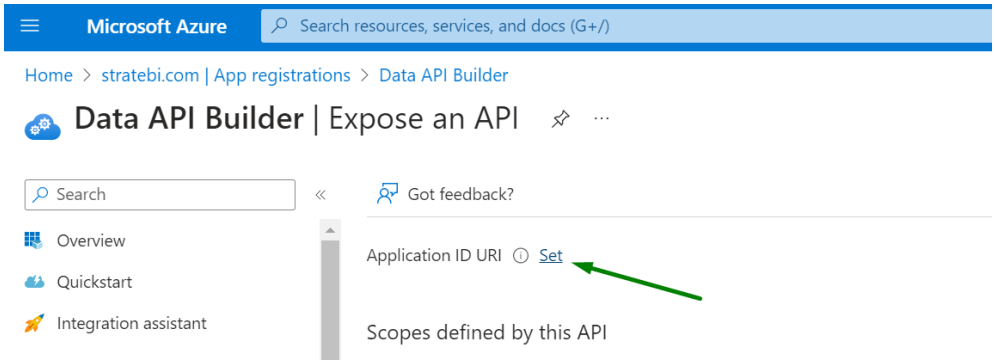


Ilustración 5: Añadir application ID URI

Añadiremos un scope y definiremos roles.

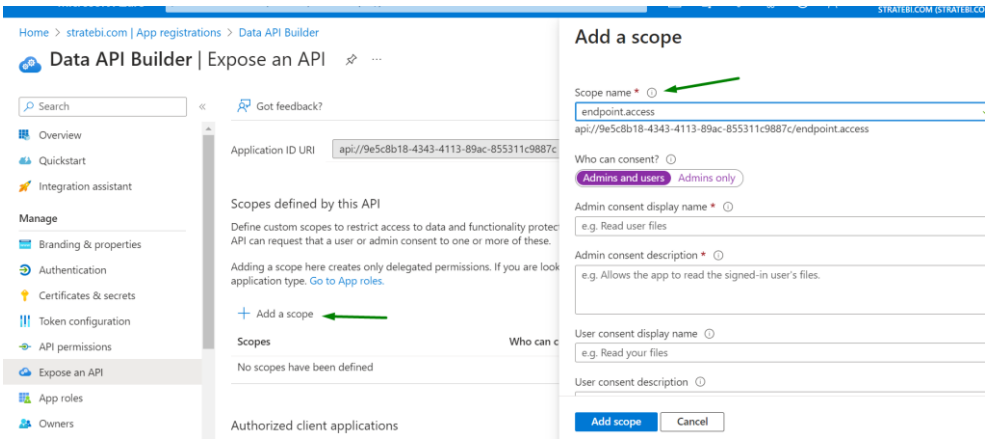


Ilustración 6: Añadiendo scope

Añadiremos también roles de usuario

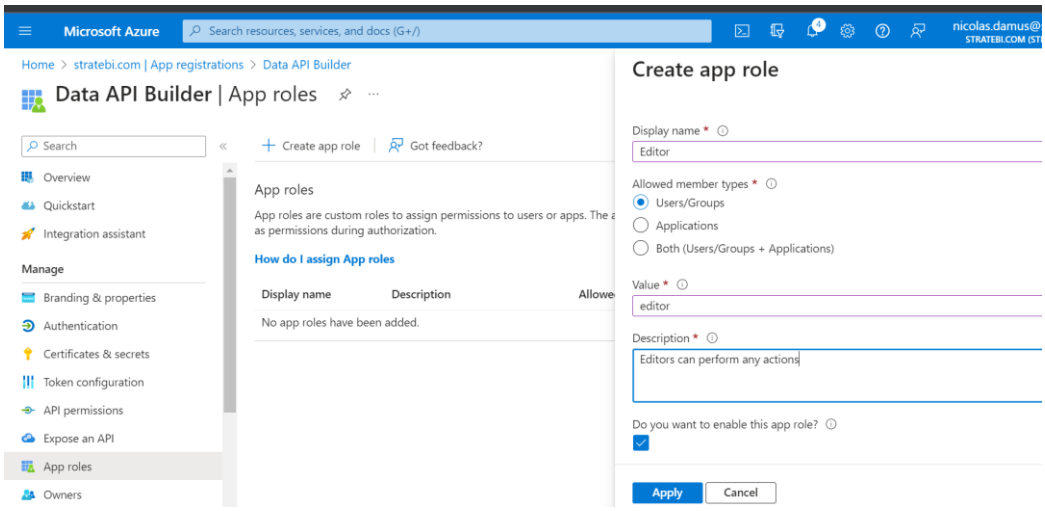


Ilustración 7: Añadiendo roles

Por último, iremos a Manifest y pondremos la propiedad accessTokenAcceptedVersion a 2

Ilustración 8: accessTokenAcceptedVersion

7. FICHERO DE CONFIGURACIÓN

Para el fichero de configuración emplearemos el siguiente [template] sustituyendo los valores audience por nuestro client id e issue por nuestro tenant id.

Por último, subiremos nuestro fichero a nuestro file share.

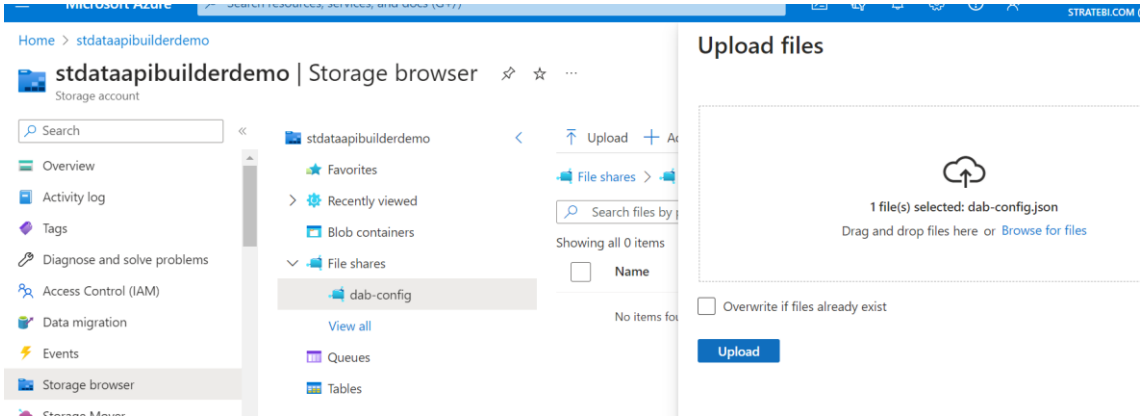


Ilustración 9: Subir fichero al file share de nuestro storage

8. CONTENEDOR DE AZURE

Ahora que ya tenemos todo listo, crearemos nuestra instancia de Azure Container. Nos hará falta el string de conexión a la base de datos creada previamente para poder poner valor a la variable de entorno. Para ello:

```
PS C:\WINDOWS\system32> az sql db show-connection-string `
>>    --client ado.net `
>>    --name DB-DATA-API-BUILDER `
>>    --server serverapibuilder `
>>    --output tsv
Server=tcp:serverapibuilder.database.windows.net,1433;Initial Catalog=DB-DATA-API-BUILDER;Persist Security Info=False;User ID=<usernam
e>;Password=<password>;MultipleActiveResultSets=False;Encrypt=true;TrustServerCertificate=False;Connection Timeout=30;
PS C:\WINDOWS\system32>
```

Ilustración 10: Obtener string de conexión a la base de datos

Una vez obtenido el string, sustituiremos username y password por los valores correspondientes y crearemos una instancia de un container de azure de la siguiente manera:

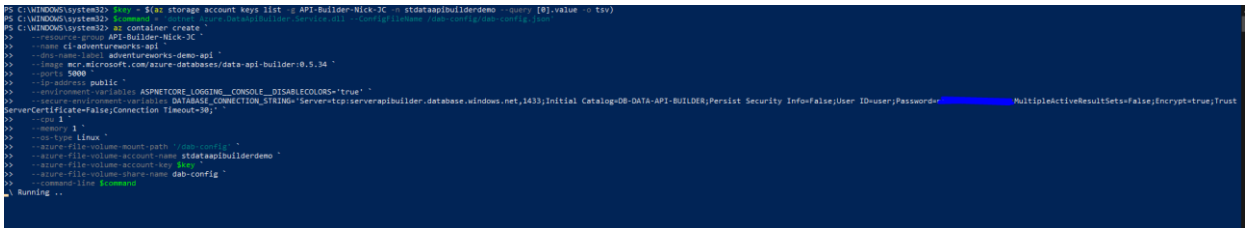


Ilustración 11: Ponemos en marcha el contenedor

Comprobamos en el portal de azure que el container ha sido creado con éxito.

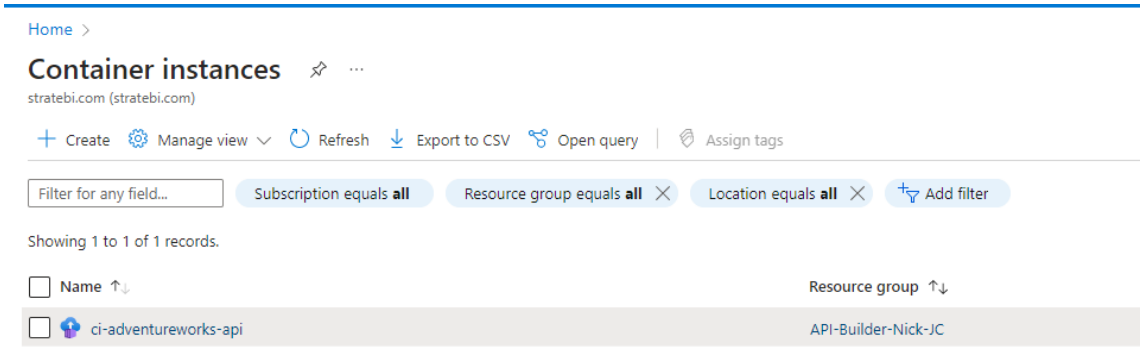


Ilustración 12: Comprobación creación del container

9. PRUEBAS

Ahora que lo tenemos todo funcionando, vamos a hacer algunas pruebas para ver el funcionamiento del API.

Como se ha comentado previamente, hemos cargado la base de datos de AdventureWorks para tener unos datos de prueba.

En primer lugar, creamos una variable para almacenar la url base. Dicha URL consta del valor asignado en el campo –dns-name-label junto con un valor fijo. Siendo la URL genérica http://dns-name-label.location.azurecontainer.io:port

$baseUrl = 'http://adventureworks-demo-api.westeurope.azurecontainer.io:5000/api'

SELECTS

Ya con esto Podemos hacer llamadas GET por terminal como las siguientes:

Invoke-RestMethod -Method GET -Uri $baseUrl/product → nos devuelve todos los productos.

Invoke-RestMethod -Method GET -Uri $baseUrl/product/ProductID/680 → nos devuelve el product cuyo id es 680.

Invoke-RestMethod -Method GET "$baseUrl/product?`$filter=Color eq 'Black' and ListPrice gt 34" → nos devuelve productos de color negro cuyo list price es mayor que 34.

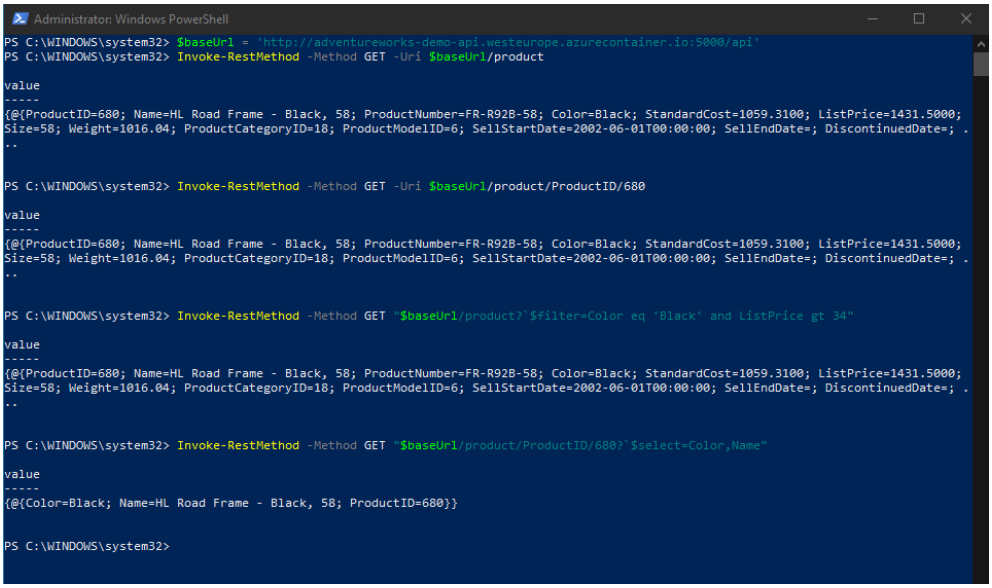


Ilustración 13: Resultados obtenidos

DELETE

Para borrar filas en nuestro endpoint podemos usar el método DELETE

```
DELETE -Uri $baseUrl/product/ProductID/680
```

Borrará el elemento con id 680.

UPDATE

Para actualizar valores de alguna tabla podemos usar los métodos PUT o PATCH. Se recomienda usar el método PATCH. Usar PUT para crear/actualizar un registro cuando está usando una primary key con identidad administrada generará un error.

Por ejemplo:

```
PATCH $baseUrl/product/ProductID/680
```

```
{
    "Name": "Nuevo_Nombre",
}
```

Se actualiza el nombre del producto al valor especificado en el body de la petición HTTP.

CREATE/INSERT

Por último, en cuanto a las operaciones CRUD, podemos añadir filas a una tabla utilizando el método POST.

Por ejemplo:

```
POST $baseUrl/product

{
    "Name": "Producto_Creado",
    "ProductNumber": "PR-C-1",
    "Color": "White"
...
}
```

Se creará un nuevo registro en la tabla producto con los valores especificados en el body de la petición HTTP.

GRAPHQL ENDPOINT

Hasta ahora, hemos estado viendo todo lo relacionado con el REST endpoint. También cabe destacar que tenemos disponible un endpoint para graphql disponible en $baseUrl/grapghql.

Sin embargo, para poder usar este endpoint no nos vale con la terminal como previamente hemos hecho, debemos utilizar un cliente rest como por ejemplo postman. Si queremos realizar una sencilla query en este formato sobre la base de datos de libros

```
{
  books(first: 5, orderBy: { title: DESC }) {
    items {
      id
      title
    }
  }
}
```

Ilustración 14: Query usando el endpoint de graphql

Esta query nos devuelve los 5 primeros libros ordenados por título de manera descendente.

Tal y como pasa con los anteriores ejemplos, disponemos de formas para realizar operaciones create, update y delete empleando mutaciones.

Por ejemplo si tenemos la entidad “book” cargada de alguna de las bases de datos de ejemplo podremos realizar dichas operaciones siguiendo este patrón:

- createbook: Crea un nuevo libro
- updatebook: Actualiza un libro
- deletebook: Borra un libro

CREATE CON GRAPHQL

```
mutation {
  createbook(item: {
    id: 2000,
    title: "Leviathan Wakes"
  }) {
    id
    title
  }
}
```

Ilustración 15: Ejemplo de creado con graphql

Aquí estamos creando un nuevo libro cuyo campo id vale 2000 y título es “Levitation Wakes”

UPDATE CON GRAPHQL

```
mutation {
  updatebook(id: 2000, item: {
    year: 2011,
    pages: 577
  }) {
    id
    title
    year
    pages
  }
}
```

Ilustración 16: Ejemplo de actualizar con graphql

Para actualizar se necesitan dos parámetros, el valor de la clave primaria del item a actualizar y los campos que se van a actualizar junto con sus valores. En este caso estamos actualizando los campos year y pages del libro con id 2000

DELETE CON GRAPHQL

```
mutation {
  deletebook(id: 1234)
  {
    id
    title
  }
}
```

Ilustración 17: Ejemplo de borrado con graphql

Para el borrado de elementos necesitaremos solamente como parámetro la clave primaria del item a borrar. En este caso estamos eliminando el libro con id 1234.

10. CONCLUSIONES

Como hemos visto, hemos podido crear un API REST de una base de datos en azure de manera bastante sencilla pudiendo realizar queries sobre la misma sin necesidad de añadir código SQL.

Vemos que se trata de una herramienta potente cuyo uso tanto para desarrollo web como para en el mundo de los datos es más que interesante, pudiendo *apificar* una base de datos para realizar queries sobre ella con peticiones HTTP al endpoint correspondiente.

Además de todo, Data API Builder es open source y funciona en cualquier plataforma, por lo que puede ser usado desde Azure como hemos visto, on premise o con un container.

stratebi
Analytics and Big Data Consulting