Proyecto final
Deployment de Api desarrollada en .Net Core y Sql Server utilizando Docker y Kubernetes

Armando Antonio Mora Reos
Centro Universitario de Ciencias Exactas e Ingenierías
Universidad de Guadalajara
Guadalajara, México
armando.mora5787@alumnos.udg.mx

***Abstract*—En este documento se describe el proceso que se siguió para desarrollar el proyecto final de la materia Computación paralela para aplicaciones orientadas a la red. Este proyecto consiste en aplicar todos los conocimientos adquiridos durante la clase.**

***Index Terms*—Docker, Kubernetes, Azure, .Net Core, Sql Server, Kubernetes Volumes**

I. Introducción

Para este proyecto se utilizo una API que se había desarrollado para una aplicación Android de un taller mecánico. Esta misma API se esta tomando como base para la elaboración del proyecto de titulación con sus debidas modificaciones.

Básicamente, en lo que consiste esta API es en un conjunto de endpoints desarrollados en .NET Core y con una base de datos SQL Server. Algunas de las funciones con las que cuenta se listan a continuación:

- Crear, Borrar, Consultar y Actualizar clientes
- Login de usuarios
- Crear, Borrar, Consultar y Actualizar Vehículos
- Crear, Borrar, Consultar y Actualizar Servicios

Esta API utiliza Swagger para poder verificar y probar que los endpoints estén funcionando correctamente.
Swagger es una herramienta que ayuda al desarrollo y depuración de APIs.

Fig. 1. Swagger.

En este proyecto, se presenta la UI de swagger para poder verificar el funcionamiento de los endpoints y la conectividad con la base de datos SQL server, en la imagen a continuación se observa los contenedores configurados con .NET Core y SQL server funcionando correctamente.

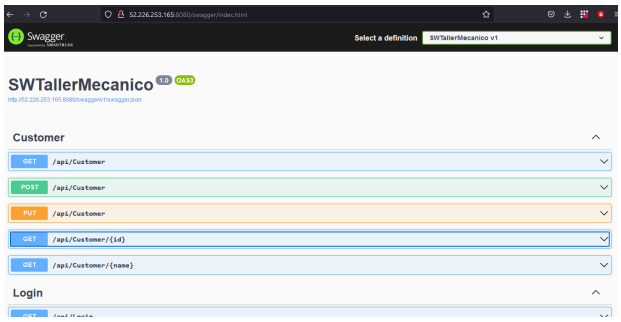

Fig. 2. Interfase de usuario de la herramienta de software Swagger.

II. Metodología

A. Herramientas

Para este proyecto se utilizo Docker para poder crear contenedores con las configuraciones necesarias, se crearon dos contenedores, uno con .Net Core y uno mas con las configuraciones para SQL Server.

Se decidió utilizar Azure debido a que ya se contaba con una cuenta previamente creada y se creo el deploy para poder ser accedido desde cualquier dispositivo mediante la dirección http://52.226.253.165:8080/swagger/index.html

B. Desarrollo

Para este proyecto se comenzó configurando los contenedores para .Net Core y SQL Server, se crearon un conjunto de Dockerfiles como los que se muestran a continuación:

```
RUN     wget https://packages.microsoft.com/config/ubuntu/21.04/packages-microsoft-prod.deb -O packages-microsoft-prod.
RUN     dpkg -i packages-microsoft-prod.deb
RUN     rm packages-microsoft-prod.deb
RUN     apt-get update; \
        apt-get install -y apt-transport-https && \
        apt-get update && \
        apt-get install -y dotnet-sdk-6.0
RUN     apt-get update; \
        apt-get install -y apt-transport-https && \
        apt-get update && \
        apt-get install -y aspnetcore-runtime-6.0
RUN     apt-get install -y dotnet-runtime-6.0
EXPOSE 5171
EXPOSE 1433
RUN mkdir /api
WORKDIR /api
COPY SWTallerMecanico /api/
#CMD dotnet run && tail -f /dev/null
CMD tail -f /dev/null
```

Fig. 3. Instalacion del stack de software necesario para compilar y ejecutar el API desarrollada en .Net Core.

Este contenedor fue subido a DockerHub para su posterior utilización con Kubernetes y Azure.

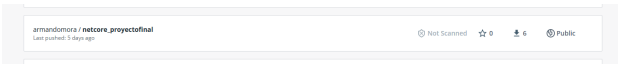

Fig. 4. Contenedor en DockerHub.

Una vez creada la imagen con el software necesario para ejecutar la aplicaciones .Net Core, se instalo el CLI de Azure para poder interactuar con Azure desde la terminal, utilizando el comando ”az aks install-cli” y así poder utilizar Kubernetes mediante kubectl usada en la practica anterior.

Una vez instalada las herramientas de Azure se creo un cluster con el comando az aks create –resource-group myResourceGroup –name myAKSCluster –node-count 1 –generate-ssh-keys –attach-acr acrName con un solo nodo como se puede apreciar a continuación:

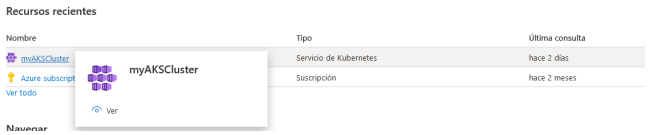

Fig. 5. Cluster de Kubernetes en Azure.

Despues de lo anterior, es necesario establecer una conexion con el cluster mediante elcomando az aks get-credentials –resource-group myResourceGroup –name myAKSCluster una vez hecho lo anterior, se puede utilizar kubectl para realiar el deploy e interactuar con el cluster de Kubernetes como se hizo en la practica 3, con la diferencia de que esta vez, se esta realizando en Azure para implementar el Api desarrollada en .Net Core.

C. Archivos de Deploy, Servicio y Volumenes

Para este proyecto se crearon una serie de archivos yaml, dos para el deploy del contenedor con .NetCore y Sql Server, dos mas para crear los servicios correspondientes y uno mas para crear un volumen persistente de almacenamiento.

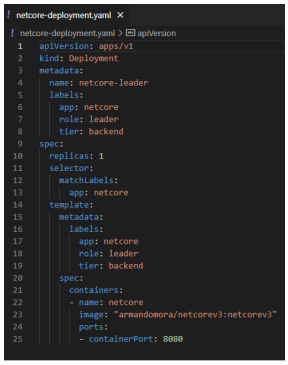

Fig. 6. Archivo para realizar el deployment del contenedor con .Net Core.

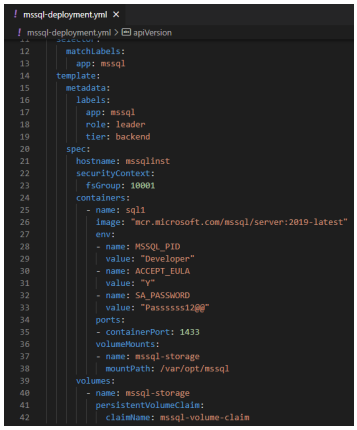

Fig. 7. Archivo para realizar el deployment del contenedor con Sql Server. En este caso, hay mejoras por aplicar, es el utilizar los Secrets para establecer el password de la base de datos.

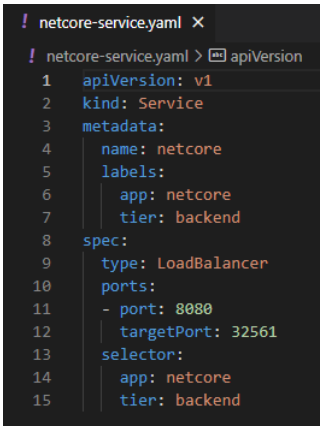

Fig. 8. Archivo para aplicar el servicio al contenedor con .Net Core.

Fig. 9. Archivo para aplicar el servicio al contenedor con Sql Server.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mssql-volume-claim
  labels:
    app: mssql
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

Fig. 10. Archivo para crear un volumen persistente y evitar la perdida de los datos cuando algún pod es destruido.

D. Azure Dashboard

Una vez aplicados los Deployments, Servicios y Volumenes, de lado de Azure podemos confirmar la creación de estos recursos, por ejemplo, en cargas de trabajo o pods tenemos netcore-leader y mssql-leader como se observa a continuación.

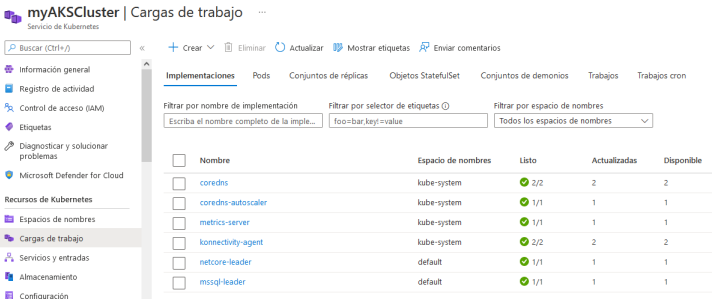

Fig. 11. Cargas de trabajo en Azure.

E. Prueba del API

Al aplicar todos los archivos con kubectl tenemos el Api ejecutándose y comunicándose correctamente con la base de datos. Como prueba es, que la interfase de Swagger se muestra mediante la url que proporciona Azure.

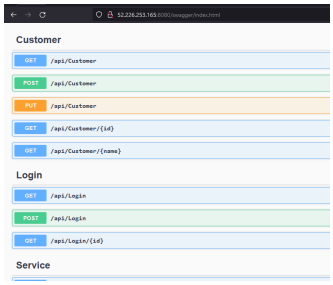

Fig. 12. Prueba de funcionamiento del Api al desplegar la interfase de Swagger.

Una vez que tenemos al Api ejecutando, se necesita de verificar que la comunicación y el contenedor de la base de datos se encuentra funcionando y accesible para la Api, este proceso se describe a continuación.

F. Prueba de la Base de datos

Para poder probar la correcta funcionalidad de la base de datos se utilizo un programa llamado TablePlus, este software nos permite establecer conexión con base de datos remotas y locales utilizando las credenciales adecuadas.

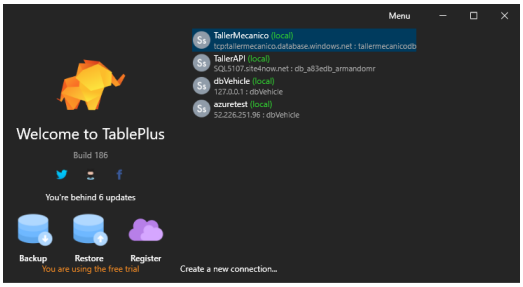

Fig. 13. Software TablePlus para probar la conexión con la base de datos remota.

Es necesario crear y agregar los accesos de conexión para poder establecer la comunicación con la base de datos, una vez establecida se pueden realizar cualquier modificación a la base de datos, ya sea, eliminar, editar registros, modificar la estructura de las tablas, crear tablas etc. en si, se puede realizar cualquier operación a la base de datos con este software.

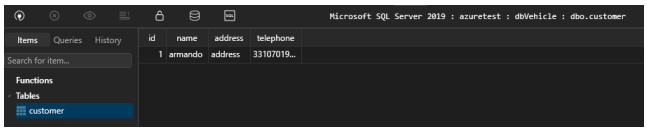

Fig. 14. Software TablePlus para probar las modificaciones a la base de datos.

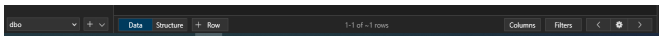

Fig. 15. Software TablePlus permite modificar la estructura y los datos de la base de datos.

Para guardar y enviar los cambios, solo se puede presionar ctrl + S y los cambios son enviados al servidor remoto o local modificando la base de datos desde la maquina local.

G. Prueba de la API y la Base de datos

Una vez que se probó el correcto funcionamiento de Swagger y la conexión con la base de datos remota, se realizaron pruebas para comprobar el correcto funcionamiento de la Api y las consultas a la base de datos, de momento en la base de datos solo esta implementada la tabla de customer, que fue la que se utilizó para verificar la comunicación entre el Api y la base de datos.

Para verificar el funcionamiento, Swagger provee un botón de try,

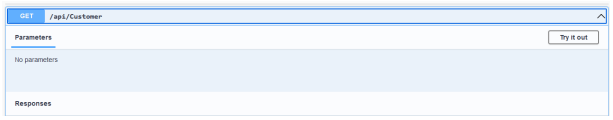

Fig. 16. Botón para ejecutar el request.

al presionar este botón aparece uno mas que al presionarlo se ejecuta el request, y se extrae la información de la base de datos, en este caso, de la tabla de customer.

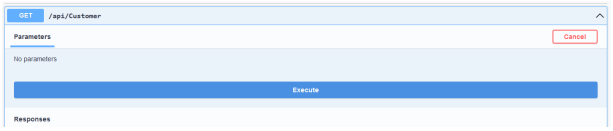

Fig. 17. Botón para ejecutar el request.

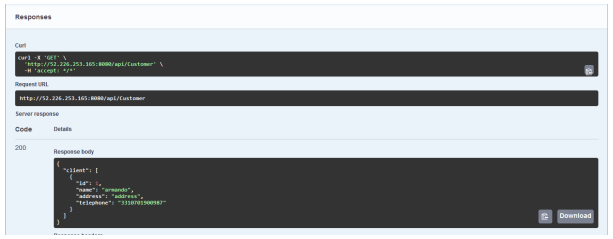

Fig. 18. Respuesta después de consultar la base de datos y ejecutar el request.

Para poder validar que la información regresada por el Api es la correcta, se utilizo de nuevo el software TablePlus para modificar los datos de la tabla customer y ejecutar el request desde Swagger para verificar que la información coincide con la introducida desde este software.

Por ejemplo, se agrego un nuevo registro con id =2, y con nombre Profe Leon desde TablePlus, seguido de esto se ejecuta

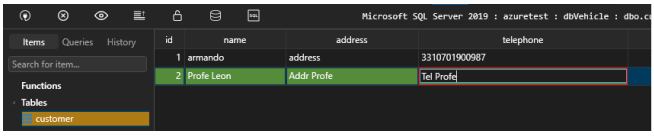

Fig. 19. Ejemplo de modificación de los datos de la base de datos remota.

y coteja la información regresada por el Api en Swagger,

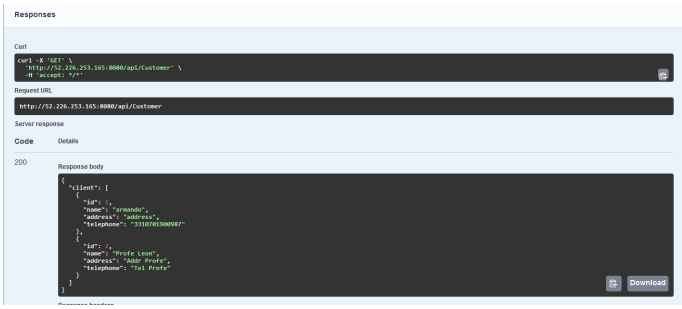

Fig. 20. Prueba de la Api después de modificar la información en la base de datos.

III. Resultados

Los resultados obtenidos fueron una api desarrollada en .Net Core funcionando y conectada a una base de datos Sql Server. Esta Api fue desplegada en la plataforma Azure utilizando Docker para crear el contenedor con .Net Core y Sql Server, además se utilizo Kubernetes para orquestar estos contenedores y manejar la comunicación entre ellos.
A continuación agrego los links solicitados:

- GIT: https://github.com/armandoantoniomorareos/CloudComputingPr
- Link de acceso: http://52.226.253.165:8080/swagger/index.html
- Acceso a Azure: Envié una invitación al profesor ve la imagen a continuación.

Fig. 21. Invitación a la instancia de Azure.

Conclusiones

El uso de contenedores y herramientas de orquestación tales como Docker y Kubernetes facilitan el deployment y generación de entornos de programación así como la replicabilidad de ambientes. Además, permite la actualización y escalamiento según las necesidades de la aplicación y la demanda de recursos que necesite con archivos de configuración y sin necesidad de preocuparse por modificar completamente el entorno.