Operación Fuego Quasar

Ing. Daniel M. Serkin

Documentación Técnica

Introducción

Este documento proporciona una descripción detallada de la API Operación Fuego Quasar, incluyendo su arquitectura, funcionalidades, implementación, pruebas y requisitos. La API Operación Fuego Quasar es una aplicación diseñada para resolver el desafío técnico de determinar la ubicación y el mensaje de una nave espacial en peligro a partir de la información recibida de varios satélites. A lo largo de este documento, exploraremos los componentes clave de la API, su diseño, las tecnologías utilizadas y los pasos necesarios para ejecutar, probar y desplegar la aplicación en diferentes entornos. Este documento servirá como una guía completa para comprender y trabajar con la API Operación Fuego Quasar.

Tecnología Utilizada

- Tecnología utilizada para la API: La API está construida con .NET Core, un framework de desarrollo de aplicaciones de código abierto desarrollado por Microsoft. En particular, se utilizó .NET Core 7.0 para desarrollar la aplicación.

- Dockerfile: El Dockerfile es un archivo de configuración utilizado para construir una imagen de contenedor Docker para la aplicación. En este caso, el Dockerfile especifica los pasos necesarios para construir la imagen de la aplicación, incluyendo la configuración del entorno de trabajo, la copia de archivos de código fuente, la restauración de dependencias, la compilación del código y la publicación de la aplicación.

- Versiones de .NET Core: Se utilizaron las versiones específicas de .NET Core SDK y .NET Core ASP.NET para el desarrollo y la ejecución de la aplicación. En este caso, se usó .NET Core SDK 7.0 y ASP.NET Core 7.0.

- Utilización de contenedores Docker: La aplicación se ejecuta dentro de un contenedor Docker, lo que proporciona un entorno de ejecución aislado y portátil para la aplicación. Esto facilita el despliegue y la gestión de la aplicación en diferentes entornos de implementación, como localmente, en la nube o en entornos de producción.

- Despliegue en Kubernetes: La imagen de contenedor Docker construida a partir del Dockerfile se desplegó en un clúster de Kubernetes en Google Cloud Platform. Kubernetes se encarga de la orquestación y gestión de los contenedores de la aplicación, garantizando su disponibilidad, escalabilidad y fiabilidad.

- Acceso a la API: Una vez desplegada en Kubernetes, la API se hace accesible a través de una dirección URL única proporcionada por Kubernetes. Esto permite a los usuarios interactuar con la API y consumir sus servicios desde cualquier ubicación con acceso a Internet.

Arquitectura de la API

La arquitectura de la API Operación Fuego Quasar está basada principalmente en microservicios, aunque también se pueden identificar principios de Diseño Dirigido por Dominio (DDD). Podemos explicarlo de la siguiente manera:

Enfoque de la Arquitectura:

La arquitectura de la API Operación Fuego Quasar se fundamenta en una combinación de microservicios y Diseño Dirigido por Dominio (DDD), aprovechando las ventajas de ambos enfoques para construir un sistema robusto y flexible.

Microservicios:

Los microservicios son unidades independientes de desarrollo y despliegue, cada uno con su propia lógica de negocio y responsabilidades bien definidas. En el contexto de esta API, los microservicios se encargan de tareas específicas, como la recepción de datos de los satélites, el cálculo de la ubicación y el mensaje de la nave espacial en peligro, y la gestión de la persistencia de datos.

La arquitectura basada en microservicios permite escalar y actualizar componentes de forma independiente, lo que facilita la gestión del sistema y la adaptación a los cambios en los requisitos y la demanda de usuarios.

Diseño Dirigido por Dominio (DDD):

El Diseño Dirigido por Dominio (DDD) es una metodología que se centra en comprender y modelar el dominio de negocio para desarrollar software que refleje fielmente las reglas y conceptos de dicho dominio. En el contexto de esta API, se pueden identificar conceptos del dominio como satélites, mensajes de radio y ubicaciones espaciales, que se reflejan en el diseño de los microservicios y la estructura de la aplicación.

La aplicación de principios de DDD contribuye a la claridad y coherencia del diseño de la API, asegurando que los diferentes componentes estén alineados con los conceptos y reglas del negocio, lo que facilita su comprensión y mantenimiento a lo largo del tiempo.

Esta combinación de microservicios y DDD proporciona una arquitectura flexible y adaptable, que permite desarrollar una API robusta y orientada al negocio.

Capas de la Arquitectura:

La arquitectura de la API Operación Fuego Quasar se organiza en varias capas, cada una con responsabilidades específicas que contribuyen al funcionamiento global del sistema. Estas capas incluyen:

1. Capa de Presentación:

- Esta capa se encarga de interactuar con los clientes externos, como aplicaciones front-end o servicios consumidores de la API. Aquí se manejan las solicitudes HTTP, la validación de datos de entrada y la generación de respuestas adecuadas.

2. Capa de Aplicación:

- La capa de aplicación contiene la lógica de negocio principal de la API. Aquí se procesan y coordinan las operaciones recibidas desde la capa de presentación. Se pueden implementar patrones como el patrón de diseño Command para gestionar las solicitudes entrantes y orquestar las acciones necesarias en las capas inferiores.

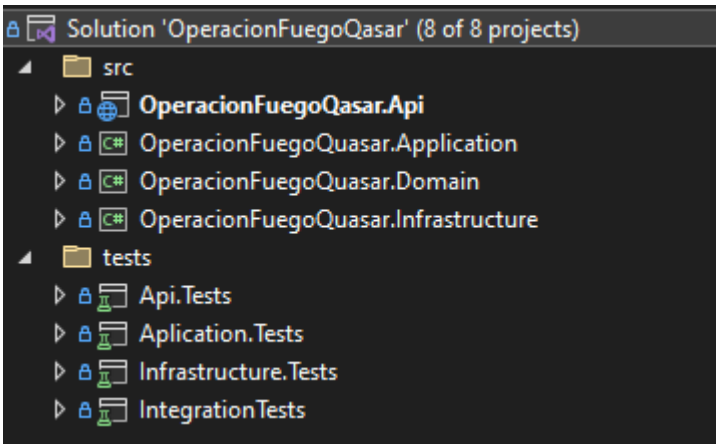
3. Capa de Dominio:

- Esta capa representa los conceptos fundamentales del negocio y define las reglas y comportamientos que rigen el sistema. Aquí se encuentran las entidades, los objetos de valor y los servicios de dominio que encapsulan la lógica de negocio. Se aplican principios de DDD para modelar el dominio de manera precisa y coherente.

4. Capa de Infraestructura:

- La capa de infraestructura proporciona los servicios técnicos necesarios para que la aplicación funcione correctamente. Esto incluye el acceso a bases de datos, servicios externos, manejo de errores, registro y monitoreo, entre otros. En esta capa se implementan adaptadores de infraestructura que permiten la comunicación con recursos externos.

Estructura:



Detalle de cada capa de la API

Api

La capa de la API consta de dos controladores principales: SatelliteController y TopSecretController.

SatelliteController

El SatelliteController maneja las solicitudes HTTP relacionadas con la obtención de la información de los satélites. Contiene un endpoint GET /satelite/topsecret_split que recupera los datos de los satélites divididos y los decodifica para obtener la información relevante.

TopSecretController

El TopSecretController maneja las solicitudes HTTP relacionadas con la información "top secret". Contiene dos endpoints: uno para recibir la información completa de los satélites (POST /topsecret) y otro para recibir la información de un satélite individual (POST /topsecret_split/{satelliteName}). Ambos endpoints procesan la información recibida y devuelven la respuesta correspondiente.

Program

El archivo Program.cs configura y construye la aplicación web. Aquí se agregan los servicios necesarios al contenedor de dependencias, se configura Swagger para la documentación de la API y se manejan las excepciones. También se realiza la migración de la base de datos al iniciar la aplicación.
Este archivo establece toda la configuración necesaria para iniciar la aplicación web y gestionar las solicitudes HTTP entrantes.

Application

ShipService

Este es el servicio principal de la aplicación, ShipService, que se encarga de decodificar la información recibida desde los satélites para determinar la ubicación y el mensaje enviado por la nave.

El método DecodeTopSecretInfoAsync es el punto de entrada del servicio y recibe un objeto TopSecret que contiene la información de los satélites. Primero, se valida la solicitud para asegurarse de que la información recibida sea válida. Luego, se eliminan todos los datos anteriores de los satélites y se agregan los nuevos datos recibidos. Después de esto, se procede a calcular la ubicación y el mensaje utilizando los métodos GetLocation y GetMessage, respectivamente. Finalmente, se devuelve un objeto TopSecretDecoded con la ubicación y el mensaje decodificado.

El método GetLocation calcula la posición de la nave utilizando la fórmula de trilateración, que utiliza las distancias medidas desde los satélites para determinar las coordenadas X e Y de la nave.

El método GetMessage extrae el mensaje enviado por la nave a partir de las diferentes transmisiones recibidas de los satélites. Primero, se eliminan las palabras duplicadas y se ordenan según su posición en el mensaje original. Luego, se unen todas las palabras únicas para formar el mensaje completo.

Este servicio encapsula toda la lógica necesaria para decodificar la información recibida de los satélites y proporciona una interfaz coherente para el resto de la aplicación.

Exceptions

La carpeta Exceptions contiene clases que representan excepciones específicas que pueden ocurrir en la aplicación. Estas excepciones están diseñadas para capturar errores o condiciones inesperadas que pueden surgir durante la ejecución del código y proporcionar información clara sobre qué salió mal.

En general, estas excepciones están diseñadas para ser capturadas en lugares donde se pueda manejar el error de manera apropiada, como en un controlador de excepciones global o en un bloque try-catch. Proporcionan una forma estructurada de manejar y comunicar errores en la aplicación, lo que facilita la depuración y el mantenimiento del código.

Domain

Entities

La carpeta Entities en el dominio (Domain) generalmente contiene las clases que representan entidades principales o conceptos clave dentro del dominio del problema que la aplicación aborda. Estas entidades suelen reflejar los objetos principales que son fundamentales para el negocio o la lógica de la aplicación.

Estas entidades pueden tener relaciones con otras entidades en el dominio y pueden participar en la lógica de negocio de la aplicación. Por lo general, se utilizan en conjunto con los repositorios para acceder y manipular los datos relacionados con esas entidades en el almacenamiento persistente.

Infrastructure

SatelliteDataRepository

La clase SatelliteDataRepository en la capa de infraestructura es responsable de interactuar con la capa de persistencia de datos para realizar operaciones relacionadas con los datos de los satélites. Aquí tienes una explicación de su funcionamiento:

Constructor:

El constructor recibe una instancia de ApplicationDbContext, que es el contexto de la base de datos utilizado para realizar operaciones de acceso a datos.

Métodos:

AddAsync:

Este método agrega un nuevo objeto SatelliteData al contexto y guarda los cambios en la base de datos.

DeleteAllDataFromTablAsync:

Elimina todos los datos de la tabla SatelliteData en la base de datos.

GetAllSatelliteDataAsync:

Obtiene todos los datos de satélite de la base de datos y los devuelve como una lista.
Manejo de Excepciones:

Todos los métodos capturan cualquier excepción que pueda ocurrir durante las operaciones de acceso a datos y lanzan una excepción personalizada DbOperationException en caso de error. Esto ayuda a centralizar el manejo de errores y simplifica el código en otras capas de la aplicación.

ApplicationDbContext

La clase ApplicationDbContext en la carpeta Data de la infraestructura es una subclase de DbContext, que es parte de Entity Framework Core y se utiliza para configurar y acceder a la base de datos.

Migrations

La carpeta Migrations es una parte esencial de Entity Framework Core y se utiliza para almacenar todas las migraciones de la base de datos. Las migraciones son clases que representan cambios en el esquema de la base de datos a lo largo del tiempo. Cuando se realizan cambios en el modelo de datos, como agregar nuevas tablas, cambiar el tipo de datos de una columna, etc., se crean migraciones para aplicar esos cambios en la base de datos.

Cada archivo de migración contiene un código que describe las operaciones necesarias para aplicar o revertir un cambio en el esquema de la base de datos. Estas operaciones pueden incluir la creación o eliminación de tablas, la adición o eliminación de columnas, la modificación de restricciones, entre otros.

Cuando se ejecuta el comando dotnet ef migrations add, Entity Framework Core genera un nuevo archivo de migración en la carpeta Migrations que refleja los cambios realizados en el modelo de datos desde la última migración. Luego, cuando se ejecuta el comando dotnet ef database update, Entity Framework aplica esas migraciones pendientes a la base de datos, asegurando que el esquema de la base de datos esté sincronizado con el modelo de datos actualizado.

En resumen, la carpeta Migrations es donde se almacenan todas las migraciones de la base de datos, lo que permite realizar un seguimiento de los cambios en el esquema de la base de datos a lo largo del tiempo y aplicar esos cambios de manera controlada y automatizada.

Api.Tests

En Api.Tests, se encuentran las pruebas unitarias para los controladores de la API. Estas pruebas se implementaron utilizando la biblioteca nUnit y Moq para simular el comportamiento de las dependencias.

Application.Tests

Dentro Application.Tests, se encuentran las pruebas unitarias para la lógica de la capa de aplicación. Estas pruebas garantizan que las diferentes funciones y servicios en esta capa funcionen como se espera.

ShipServiceTests

Este archivo contiene pruebas para el servicio ShipService, que maneja la lógica relacionada con la decodificación de información de los satélites y el cálculo de la ubicación y el mensaje de la nave.

Infrastructure.Tests

Infrastructure.Tests contiene pruebas unitarias destinadas a validar el funcionamiento de las clases y componentes específicos de la capa de infraestructura. Estas pruebas se centran en garantizar que la infraestructura de la aplicación, como la persistencia de datos y el manejo de excepciones, se comporte según lo esperado.

Integration Tests

Los Integration Tests se utilizan para probar la interacción entre los diferentes componentes de la aplicación en un entorno similar al de producción. Estas pruebas garantizan que los diferentes módulos de la aplicación funcionen correctamente juntos y se comuniquen adecuadamente.

TopSecretControllerIntegrationTests

Este archivo contiene pruebas de integración para el controlador TopSecretController, que maneja las solicitudes relacionadas con la decodificación de mensajes y la obtención de información de los satélites.

PostAsync_WithValidData_ReturnsDecodedInfo:

Esta prueba verifica que la solicitud POST al endpoint /topsecret con datos válidos devuelva la información decodificada correctamente. Se envía una solicitud con datos simulados de satélites y se espera que la respuesta contenga la información decodificada.

SplitAsync_WithValidData_ReturnsDecodedInfo:

Esta prueba valida que la solicitud POST al endpoint /satelite/topsecret_split/{satelliteName} con datos válidos devuelva la información decodificada correctamente. Se envía una solicitud con datos simulados de satélites y se espera que la respuesta contenga la información decodificada.

SplitAsync_WithInvalidData_ReturnsError:

Esta prueba comprueba que la solicitud POST al endpoint /satelite/topsecret_split/{satelliteName} con datos inválidos devuelva un error. Se envía una solicitud con datos de satélite inválidos y se espera que la respuesta indique un estado de error.

GetSatelliteInfoAsync_ReturnsSatelliteInfo:

Esta prueba verifica que la solicitud GET al endpoint /satelite/{satelliteName} devuelva la información del satélite correctamente. Se envía una solicitud para obtener información de un satélite específico y se espera que la respuesta contenga la información del satélite.

Estas pruebas de integración se centran en validar el comportamiento del controlador TopSecretController en situaciones de uso real, asegurando que las solicitudes HTTP se manejen correctamente y que se devuelva la información esperada.

Base de Datos

SQLite se utiliza como el motor de base de datos en la aplicación "OperacionFuegoQuasar" para almacenar y gestionar la información relacionada con los datos de los satélites. A continuación, se describen los aspectos principales sobre cómo se integra SQLite en la arquitectura de la aplicación:

Configuración de la Base de Datos

La configuración de SQLite se realiza a través de la clase ApplicationDbContext, que actúa como el contexto de la base de datos. En esta clase, se especifica la conexión a la base de datos mediante un archivo SQLite (operacion-fuego-qasar.db). La configuración se realiza en el método OnConfiguring, donde se establece la conexión con la base de datos.

Modelo de Datos

El modelo de datos se define utilizando entidades como SatelliteData, que representan los datos de los satélites en la base de datos. Estas entidades se mapean directamente a tablas en la base de datos SQLite. Por ejemplo, la entidad SatelliteData define propiedades como Name, Distance y Message, que se corresponden con las columnas de la tabla en la base de datos.

Operaciones de Base de Datos

Las operaciones de base de datos se realizan a través del contexto de la base de datos (ApplicationDbContext). En las clases de repositorio, como SatelliteDataRepository, se implementan métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos. Estos métodos permiten agregar, eliminar y recuperar datos de la tabla SatelliteData de la base de datos SQLite.

Migraciones

El sistema de migraciones de Entity Framework Core se utiliza para administrar la estructura de la base de datos. Durante el inicio de la aplicación, las migraciones se aplican automáticamente para garantizar que la base de datos esté sincronizada con el modelo de datos. Esto permite mantener la integridad y consistencia de la base de datos a medida que evoluciona la aplicación.

En resumen, SQLite se integra de manera efectiva en la aplicación "OperacionFuegoQuasar" como un motor de base de datos ligero y autónomo, proporcionando una solución eficiente para el almacenamiento y gestión de datos relacionados con los satélites. Su configuración, modelo de datos, operaciones de base de datos y migraciones se gestionan de manera coherente dentro de la arquitectura de la aplicación.

Documentacion de la API

Ruta: /satelite/topsecret_split

Método: GET

Descripción: Obtiene la información decodificada de los mensajes de los satélites.

Respuestas:

200 OK: Devuelve la información decodificada en formato JSON.
Schema: TopSecretDecoded
404 Not Found: No se encontraron datos para decodificar.
Schema: ProblemDetails

Ruta: /topsecret

Método: POST

Descripción: Envía la información de los mensajes de los satélites para su decodificación.
Parámetros de cuerpo de solicitud:
TopSecret: Información de los mensajes de los satélites.

Respuestas:

200 OK: Devuelve la información decodificada en formato JSON.
Schema: TopSecretDecoded
400 Bad Request: La solicitud es incorrecta o incompleta.
Schema: ProblemDetails

```
{
  "satellites": [
    {
```

```
      "name": "kenobi",
      "distance": 100.0,
      "message": ["este","","","mensaje",""]
    },
     {
      "name": "skywalker",
      "distance": 105.5,
      "message": ["","es","","","secreto"]
    },
     {
      "name": "sato",
      "distance": 142.7,
      "message": ["este","","un","",""]
    }
  ]
}
```


Ruta: /topsecret_split/{satelliteName}

Método: POST

Descripción: Almacena la información de un mensaje de un satélite específico para su decodificación.
Parámetros:
satelliteName: Nombre del satélite del cual se recibe el mensaje.
Parámetros de cuerpo de solicitud:
TopSecretSplit: Información del mensaje del satélite.

Respuestas:

200 OK: El mensaje del satélite se almacenó correctamente.
400 Bad Request: La solicitud es incorrecta o incompleta.
Schema: ProblemDetails

topsecret_split/kenobi


```
{
  "distance": 100.0,
  "message": ["este","","","mensaje",""]
}
```


topsecret_split/kenobi


```
{
```

```
  "distance": 142.7,
  "message": ["este","","un","",""]
}
```


topsecret_split/skywalker


```
{
  "distance": 115.5,
  "message": ["","es","","","secreto"]
}
```


Ejecución local

Para probar la API utilizando Visual Studio y Swagger, puedes seguir estos pasos:

Iniciar la aplicación desde Visual Studio: Abre tu solución en Visual Studio y asegúrate de que el proyecto de la API esté configurado como el proyecto de inicio. Luego, simplemente presiona F5 para iniciar la aplicación. Esto compila y ejecuta tu API localmente.

Acceder a Swagger: Una vez que la aplicación se haya iniciado correctamente, puedes acceder a Swagger visitando la URL base de tu API en un navegador web y agregando /swagger al final de la URL. Por ejemplo, si tu API se ejecuta en http://localhost:5000, la URL de Swagger sería http://localhost:5000/swagger.

Explorar la documentación de la API: Una vez en Swagger, verás una interfaz interactiva que muestra todos los endpoints de tu API, así como los modelos de datos y las operaciones disponibles. Puedes probar cada endpoint directamente desde Swagger haciendo clic en ellos y utilizando los controles proporcionados para enviar solicitudes y ver las respuestas.

Enviar solicitudes de prueba: Utilizando la interfaz de Swagger, puedes enviar solicitudes de prueba a tus endpoints utilizando diferentes parámetros y valores de carga útil para asegurarte de que la API responde correctamente.

Revisar la documentación generada automáticamente: Además de permitirte probar la API, Swagger también genera automáticamente documentación detallada para cada endpoint, incluyendo descripciones de los parámetros, códigos de respuesta y modelos de datos. Puedes revisar esta documentación para comprender mejor cómo usar la API y qué esperar de cada endpoint.

Dockerizacion

Base de Imagen ASP.NET: La sección base utiliza la imagen base de ASP.NET para crear un entorno de ejecución para la aplicación.

Compilación y Publicación: La sección build se encarga de compilar la aplicación utilizando la imagen de SDK de .NET y restaura las dependencias del proyecto. Luego, publica la aplicación en un directorio específico.
Imagen Final: La sección final utiliza la imagen base definida anteriormente y copia los archivos publicados de la sección publish a la imagen final. Define el punto de entrada de la aplicación.
Para usar este Dockerfile localmente:

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
COPY ["src/OperacionFuegoQasar.Api/OperacionFuegoQasar.Api.csproj", "src/OperacionFuegoQasar.Api/"]
COPY ["src/OperacionFuegoQuasar.Application/OperacionFuegoQuasar.Application.csproj", "src/OperacionFuegoQuasar.Application/"]
COPY ["src/OperacionFuegoQuasar.Domain/OperacionFuegoQuasar.Domain.csproj", "src/OperacionFuegoQuasar.Domain/"]
COPY ["src/OperacionFuegoQuasar.Infrastructure/OperacionFuegoQuasar.Infrastructure.csproj", "src/OperacionFuegoQuasar.Infrastructure/"]
COPY ["src/OperacionFuegoQasar.Api/operacion-fuego-qasar.db", "src/OperacionFuegoQasar.Api/"]

RUN dotnet restore "src/OperacionFuegoQasar.Api/OperacionFuegoQasar.Api.csproj"
COPY . .
WORKDIR "/src/src/OperacionFuegoQasar.Api"
RUN dotnet build "OperacionFuegoQasar.Api.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "OperacionFuegoQasar.Api.csproj" -c Release -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "OperacionFuegoQasar.Api.dll"]
```

- Asegúrate de tener Docker instalado en tu máquina.
- Coloca el Dockerfile en el directorio raíz de tu proyecto ASP.NET.
- Abre una terminal en el directorio donde se encuentra el Dockerfile.
- Ejecuta el siguiente comando para construir la imagen Docker:

```
docker build -t nombre-de-tu-imagen .
```

Este comando ejecutará el contenedor en segundo plano (-d), mapear el puerto 8080 de tu máquina host al puerto 80 del contenedor (-p 8080:80) y utilizará la imagen que acabas de construir (nombre-de-tu-imagen).

¡Y eso es todo! Ahora deberías poder acceder a tu aplicación ASP.NET ejecutándose dentro de un contenedor Docker localmente en http://localhost:8080.

Publicación en la nube

- Subida de la imagen a Docker Hub: La imagen de la aplicación se cargó en Docker Hub, una plataforma de almacenamiento en la nube para imágenes de contenedores. Esto permite acceder y compartir fácilmente la imagen desde cualquier lugar.

- Configuración de Kubernetes en Google Cloud: Se estableció un entorno de Kubernetes en Google Cloud Platform, una plataforma de computación en la nube que permite ejecutar aplicaciones en contenedores de manera escalable y eficiente.

- Despliegue de la API en Kubernetes: La aplicación se desplegó en Kubernetes, donde Kubernetes se encargó de gestionar la infraestructura subyacente y distribuir los contenedores de la aplicación en los nodos disponibles para garantizar la disponibilidad y la escalabilidad.

- Acceso a la API: Una vez desplegada, la API se hizo accesible a través de una dirección web única proporcionada por Kubernetes. Esto permite a los usuarios interactuar con la API y consumir sus servicios desde cualquier ubicación con acceso a Internet.

URL DockerHub:

https://hub.docker.com/repository/docker/danielserkin/operacion-fuego-qasar

URL API Pública para pruebas en la nube:

http://34.133.116.5/index.html