Universidad ORT Uruguay

Facultad de Ingeniería

Proyecto: Obligatorio Programación de Redes

Entregado como requisito para la aprobación del Obligatorio 3 de Programación de Redes

Autores:

Francisco Martinez (233126)

Federico Caimi (176902)

Tutores:

Luis Barrague

Roberto Assandri

2021

Índice

Obligatorio 1

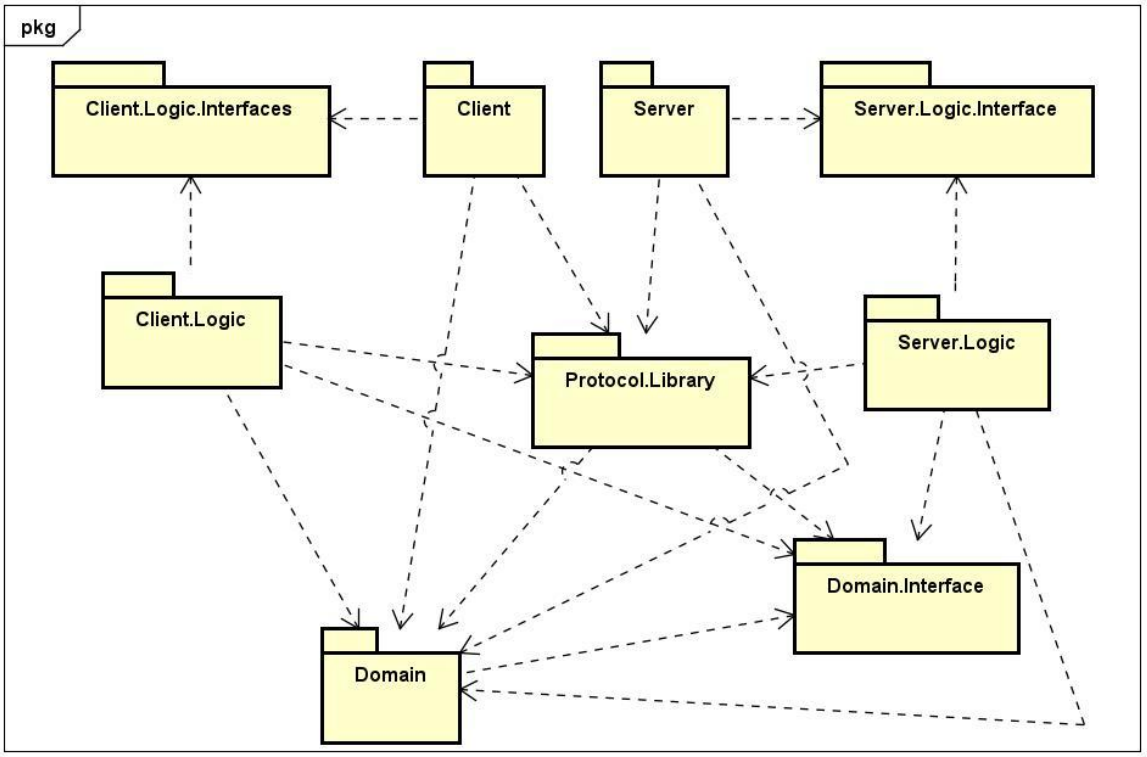
Alcance de la aplicación

El objetivo consta de construir un sistema que cuente con dos aplicaciones, un servidor encargado de guardar datos de publicaciones de usuarios con su respectiva descripción, temas y a lo sumo un archivo relacionado y una aplicación de consola cliente que se encarga de interactuar con el servidor enviando solicitudes para el manejo de las publicaciones, temas y archivos.

Diseño y arquitectura del sistema

La arquitectura es del tipo cliente-servidor, consiste básicamente en un cliente que realiza peticiones (requests) a otro programa (en este caso el servidor) que le da una respuesta (response), respetando un protocolo de red definido por nosotros para enviar y recibir datos, el cual será especificado en la sección “Diseño de protocolo”.

Diagrama de paquetes



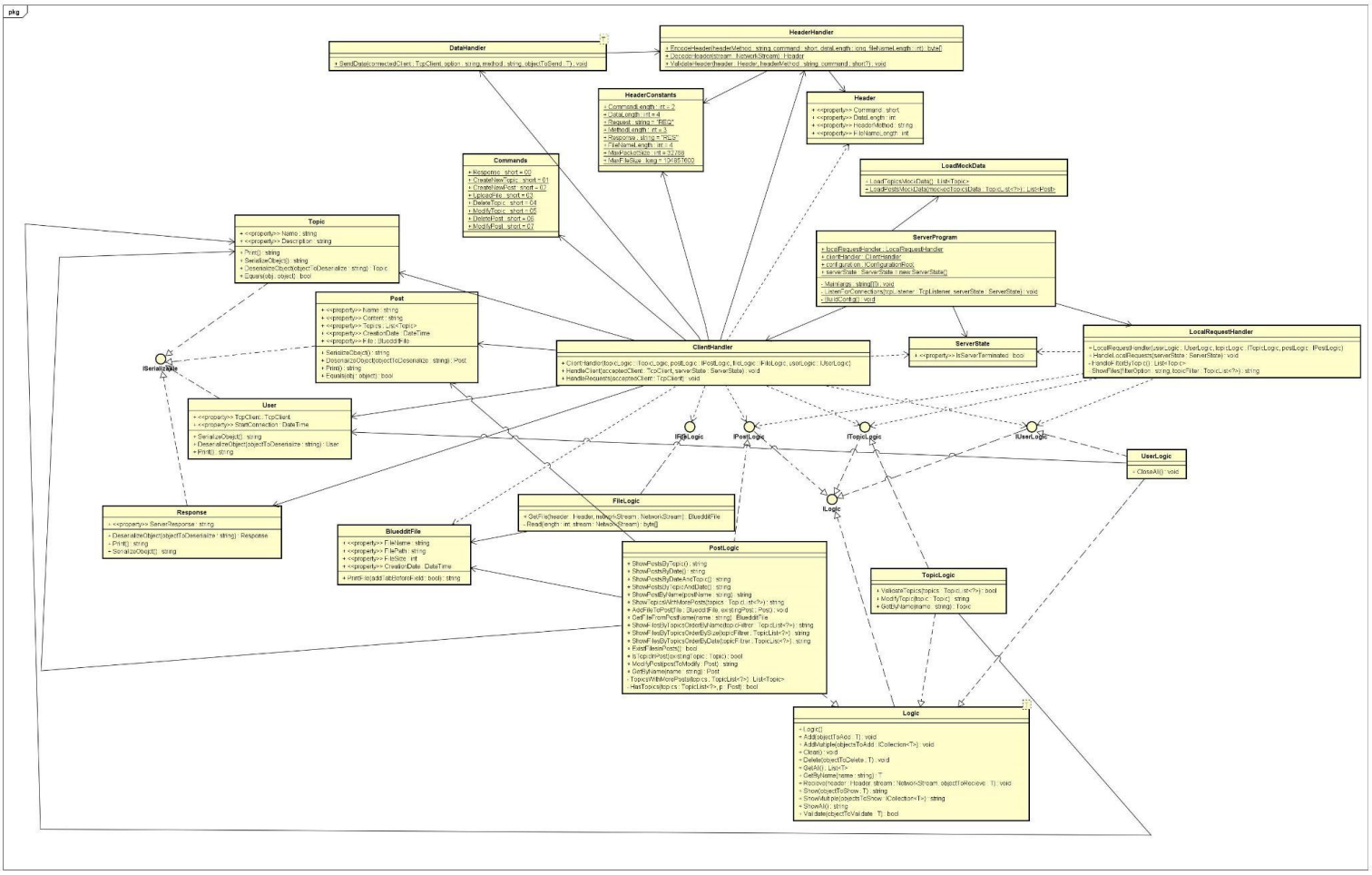
En este diagrama se muestra las dependencias entre los paquetes que conforman tanto el cliente como el servidor. Mostrando de forma explícita que el cliente no tiene dependencias del servidor y viceversa, sin embargo ambos dependen del dominio (**Domain**), la interfaz de dominio y las librerías de protocolo (**Protocol.Library**), que son comunes a ambos para evitar repetir código. A continuación detallaremos en las secciones servidor y cliente, la arquitectura de cada uno de ellos.

Servidor

Para implementar el servidor se realizó, de acuerdo a lo pedido en la letra, una aplicación de consola capaz de soportar múltiples solicitudes de clientes mediante el uso de threads, así como para soportar peticiones internas desde el mismo servidor.
Con respecto a la persistencia de datos, según a lo acordado con los docentes, se decidió utilizar persistencia en memoria en vez de una base de datos o algún otro medio de persistencia.

El servidor está compuesto por tres paquetes propios (**Server**, **Server.Logic** y **Server.Logic.Interface**) y tres compartidos con el cliente (**Protocol.Library**, **Domain** y **Domain.Interface**). Como se puede apreciar en el diagrama, la aplicación de consola no depende del paquete de la lógica, sino del paquete de interfaces de la lógica. Esto se debe a la utilización de inyección de dependencias mediante el inyector provisto por .net core. Esto permite cumplir con el principio SOLID de inversión de dependencias, ya que un módulo de alto nivel como es la aplicación de consola no depende de la librería de la lógica, sino que depende de interfaces bien definidas y estables. A partir del diagrama de clases del servidor detallaremos más en profundidad los paquetes que componen al servidor.

Diagrama de clases del servidor

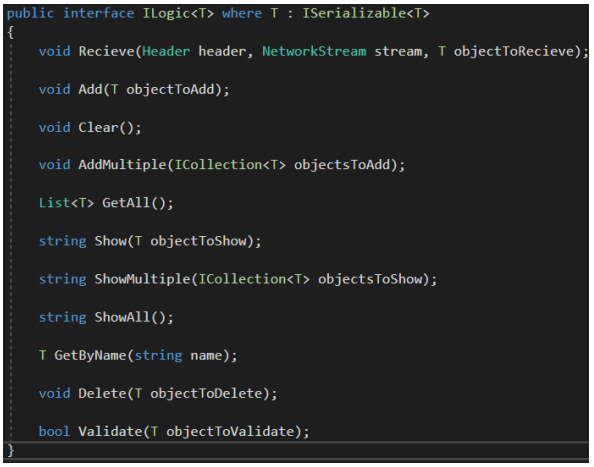


El paquete **Server** se compone de cuatro clases: **ServerProgram, LocalRequestHandler, ClientHandler y LoadMockData.** Como se puede ver en el diagrama, el **ServerProgram** es la aplicación principal de la consola (donde se encuentra el método main). Aquí se maneja la inicialización del inyector de dependencias y del archivo de configuración.

Es a su vez el encargado de manejar la creación de threads para atender las peticiones que se realizan desde la consola del servidor utilizando la clase **LocalRequestHandler** y las peticiones externas de los clientes mediante la clase **ClientHandler**. Estos serían controladores según los criterios GRASP, ya que tienen la responsabilidad de manejar los eventos del sistema y de utilizar la lógica que corresponda de acuerdo al evento entrante. También colaboran a la alta cohesión y el bajo acoplamiento del sistema, ya que le saca responsabilidad al **ServerProgram** y disminuye las dependencias entre módulos.

La clase **LoadMockData** se utiliza para cargar datos de prueba en el servidor, permitiendo sobreescribir la memoria del mismo una vez que se vuelven a cargar.

El paquete **Server.Logic.Interfaces** está compuesto por varias interfaces que implementan a la interfaz ILogic<T>. Esta interfaz es una interfaz genérica que se utiliza para definir los métodos comunes a cada lógica, lo que permite reducir considerablemente la repetición de código.



```
public interface ILogic<T> where T : ISerializable<T>
{
    void Recieve(Header header, NetworkStream stream, T objectToRecieve);

    void Add(T objectToAdd);

    void Clear();

    void AddMultiple(ICollection<T> objectsToAdd);

    List<T> GetAll();

    string Show(T objectToShow);

    string ShowMultiple(ICollection<T> objectsToShow);

    string ShowAll();

    T GetByName(string name);

    void Delete(T objectToDelete);

    bool Validate(T objectToValidate);
}
```

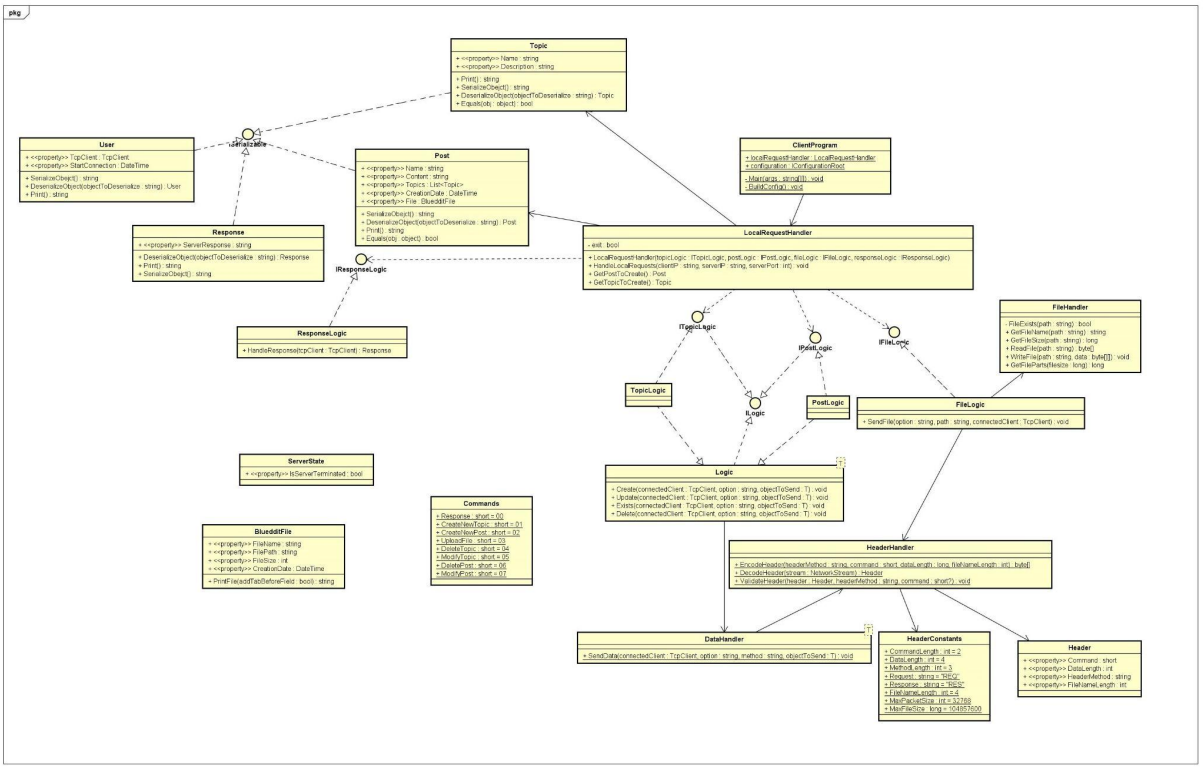
Las clases del paquete **Server.Logic** implementan a todas las interfaces definidas en el paquete anterior, y además heredan (salvo FileLogic) a la clase genérica Logic. En esta clase (que implementa a ILogic) se implementan los métodos en común a todas las lógicas. Esta estructura es muy similar al patrón repository, pero aplicada a la lógica de la aplicación.

Cliente

Para implementar el cliente se realizó una aplicación de consola para realizar solicitudes al servidor y recibir respuestas del mismo. A diferencia del servidor, no es una aplicación multi thread, ya que se envían y reciben peticiones dentro del mismo thread.

El cliente está compuesto por tres paquetes propios (**Client**, **Client.Logic** y **Client.Logic.Interface**) y tres compartidos con el server (**Protocol.Library**, **Domain** y **Domain.Interface**). También se utiliza la técnica de inyección de dependencias en el cliente para lograr cumplir con el principio de inversión de dependencias.

Diagrama de clases del cliente



Como se puede observar, el diagrama de clases del cliente es muy parecido al diagrama de clases del servidor, salvo que se tiene un manejador menos. La clase **ClientProgram** es el punto de entrada a la aplicación. Ambos utilizan arquitecturas análogas.

Paquetes compartidos

Para reutilizar código entre aplicaciones, las aplicaciones de cliente y servidor comparten los paquetes de **Domain**, **Domain.Interfaces** y **Protocol.Library.**
En el paquete **Domain** se encuentran las clases que componen el dominio de la aplicación, las cuales son utilizadas por ambas aplicaciones. Los objetos Topic, Post y BlueddItFile se identifican por su nombre, mientras que el resto de las clases del dominio no tienen identificador.
Estas clases implementan la interfaz **ISerializable** que se encuentra en **Domain.Interfaces.** La implementación de los métodos definidos por esta interfaz permite serializar y deserializar objetos a un formato similar a json para su posterior envío mediante las clases **DataHandler**, **HeaderHandler** y **FileHandler** que se encuentran dentro del **Protocol.Library**.

DataHandler es una clase genérica y estática que se encarga del envío de datos (salvo archivos) incluyendo su header el cual es enviado antes que el dato definiendo su largo.
HeaderHandler es una clase estática utilizada para codificar y decodificar el header a enviar o recibir, y a su vez valida la correctitud del header recibido.
FileHandler es una clase que se encarga de las transformaciones y validaciones de archivos.

Dentro del paquete **Protocol.Library** también se encuentran las clases **HeaderConstants**, la cual se utiliza para definir constantes relacionadas al protocolo implementado y la clase Commands, que son los comandos definidos para enviar solicitudes al servidor.

Decisiones

Diseño de protocolo

Se diseñó un protocolo muy similar al dado en la letra, con algunas modificaciones. Es un protocolo orientado a caracteres, implementado sobre TCP/IP, el cual tiene 4 campos: METHOD, COMMAND, DATALENGTH y FILENAMELENGTH. Nótese que no se envían datos en la trama, esto es porque se optó por realizar un protocolo más sencillo donde los datos se envían luego de validado el header. Solo se recibirán los datos transferidos si el header es validado correctamente tanto en el cliente o en el servidor (para esto se utiliza la clase **HeaderHandler** detallada anteriormente).

A continuación se detalla el formato general de la trama:

Nombre del campo	METHOD	COMMAND	DATALENGTH	FILENAMELENGTH
Valores	REQ/RES	0-99	Entero	Entero
Largo	3	2	4	4

- El campo METHOD indica que tipo de operación se está haciendo, si una solicitud o una respuesta. Al recibir un header se valida que el método recibido sea el esperado.
- El campo COMMAND se utiliza para indicar al servidor qué operación se debe realizar.
- El campo DATALENGTH se utiliza para indicar el largo de los datos a ser enviados luego del header.
- El campo FILENAMELENGTH se utiliza solamente a la hora de enviar archivos, para obtener la longitud del nombre del archivo, en el resto de los casos tiene valor 0.

Manejo de errores

El manejo de excepciones de la aplicación es un manejo básico. Los try-catch están posicionados fuera del while loop principal tanto del cliente como del servidor, que permiten transmitir los mensajes de excepciones y cerrar el socket si es el caso. Las excepciones propias se lanzan cuando hay un error decodificando el header, cuando el archivo a enviar es mayor a 100MB o cuando el path del archivo no existe.

Con respecto a errores a la hora de realizar una acción (no se puede dar de alta un tema o borrar un archivo, como ejemplos), se envían mensajes de respuesta desde el servidor al cliente para especificar que tal o cual acción no pudo ser realizada.

Manejo de Threads y Concurrencia

El manejo de threads y concurrencia se da en el servidor, ya que el cliente es un programa que corre en un solo thread y su objetivo es enviar solicitudes y mostrar respuestas del servidor. Con respecto al servidor, se crea un thread para escuchar pedidos de conexión de los distintos clientes, y los comandos ingresados en la consola del server se atienden en el thread principal:

```
var threadServer = new Thread(()=> ListenForConnections(tcpListener, serverState));
threadServer.Start();

while (!serverState.IsServerTerminated)
{
    localRequestHandler.HandleLocalRequests(serverState);
}
tcpListener.Stop();
```

En el método ListenForConnections, se esperan nuevas conexiones por parte de los clientes. No se crean threads innecesarios ya que se usa la operación bloqueante AcceptTcpClient antes de crear nuevos threads. Esto permite que se cree un nuevo thread solo cuando se conecta un cliente nuevo al servidor.

```
private static void ListenForConnections(TcpListener tcpListener, ServerState serverState)
{
    while (!serverState.IsServerTerminated)
    {
        try
        {
            var acceptedClient = tcpListener.AcceptTcpClient();
            var threadClient = new Thread(() => clientHandler.HandleClient(acceptedClient, serverState));
            threadClient.Start();
        }
        catch (SocketException se)
        {
            Console.WriteLine("El servidor está cerrándose...");
            serverState.IsServerTerminated = true;
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Nótese que tanto el thread principal como los threads creados para atender a los clientes utilizan el objeto estático **ServerState**. Este objeto tiene como objetivo el terminar la ejecución de todos los threads existentes cuando desde el thread principal se ingrese el comando para cerrar el server.

El manejo de concurrencia se realizó mediante la utilización de locks sobre el objeto del dominio (en este caso post o tema) a modificar o borrar. Esto permite que otras operaciones

sobre objetos no lockeados se realicen de forma normal, mientras que operaciones sobre el objeto lockeado deban esperar a que termine la operación anterior. En el caso de que se borre el objeto, se pregunta si el objeto existe dentro de la zona de mutua exclusión.

Un ejemplo de esto es el de modificar un tema:

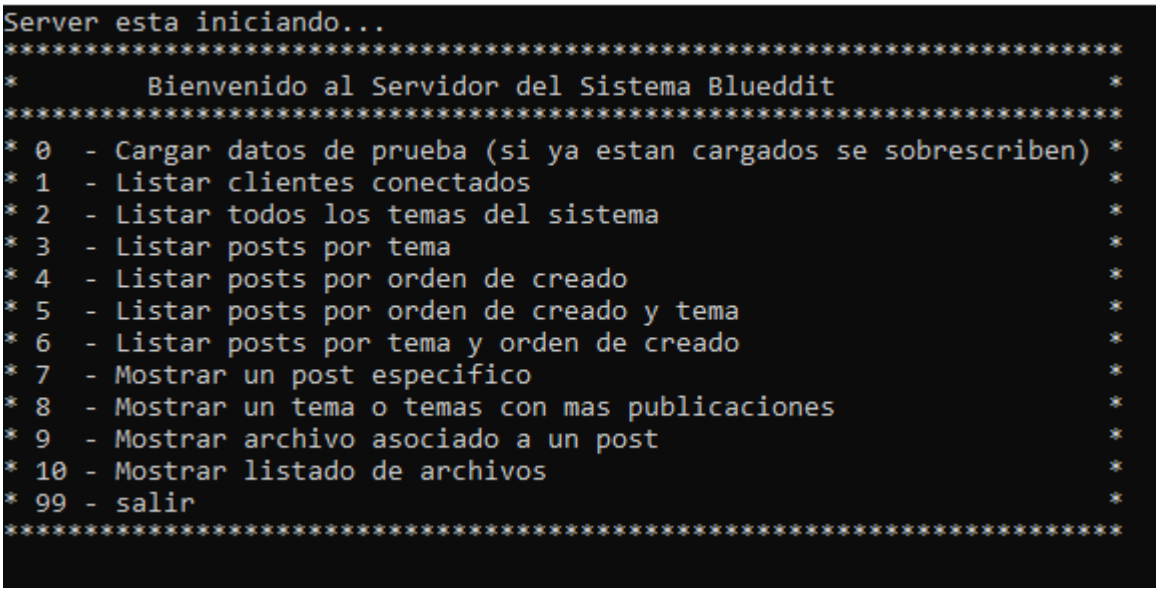
```
var topicToModify = new Topic();
_topicLogic.Recieve(header, networkStream, topicToModify);
var topicToModifyLock = _topicLogic.GetByName(topicToModify.Name);
if(topicToModifyLock != null)
{
    lock (topicToModifyLock)
    {
        var topicResponse = _topicLogic.ModifyTopic(topicToModify);
        var topicModifiedResponse = new Response { ServerResponse = topicResponse };
        DataHandler<Response>.SendData(acceptedClient, Commands.Response.ToString(), HeaderConstants.Response, topicModifiedResponse);
    }
}
else
{
    var topicModifiedErrorResponse = new Response { ServerResponse = "El tema no existe" };
    DataHandler<Response>.SendData(acceptedClient, Commands.Response.ToString(), HeaderConstants.Response, topicModifiedErrorResponse);
}
```

```
public string ModifyTopic(Topic topic)
{
    var existingTopic = GetByName(topic.Name);
    if(existingTopic != null)
    {
        var topicIndex = _elements.FindIndex(t => t.Equals(topic));
        _elements[topicIndex] = topic;
        return "El tema se modifico con exito";
    }
    else
    {
        return "El tema no existe";
    }
}
```

Aquí se observa que se pregunta nuevamente dentro de la operación ModifyTopic (dentro de la zona de mutua exclusión) si el objeto existe.

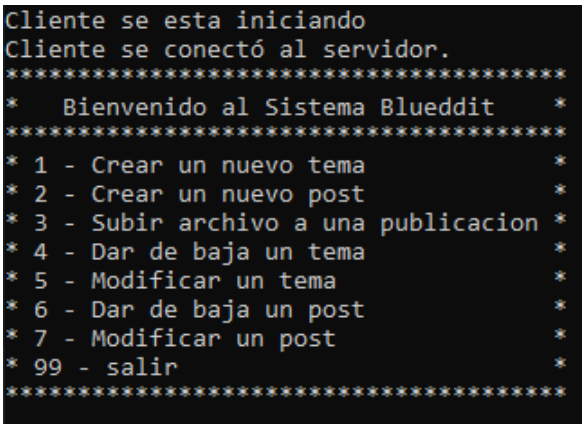
Funcionamiento de la aplicación

Servidor



- La opción 0 permite generar datos de prueba básicos.
- La opción 1 permite mostrar los clientes conectados, como se especifica en el SRF2. Con esto se da por entendido que el servidor es capaz de aceptar pedidos de conexión de múltiples clientes (SRF1).
- La opción 2 permite listar todos los temas del sistema, como se especifica en la letra. (SRF3)
- Las opciones 4 a 6 permiten listar los posts de acuerdo a temas, por orden de creado o por una combinación de ambos filtros. A la hora de listar los posts por temas, se listan todos los posts ordenados alfabéticamente de acuerdo a sus temas, apareciendo primero el menor por orden alfabético (si hay dos posts con dos temas a y b, se muestra primero el post con tema a). Si el post tiene más de un tema, se considera para ordenarlo al tema que aparezca primero por orden alfabético.
  Cuando se listan por orden de creado se muestran ordenados de forma descendente de acuerdo a su fecha de creación.
  La combinación de ambos filtros se usa para “desempatar” en caso de que haya posts con el mismo orden en el primer filtro. Con esto se cumple con el requerimiento funcional SRF4.
- La opción 7 permite ver un post específico, como se indica en el requerimiento funcional SRF5.
- La opción 8 permite ver el tema (o temas si tienen la misma cantidad de publicaciones). Cumple con el requerimiento SRF7.
- La opción 9 permite ver la información de un archivo asociado a un post (SRF6).
- La opción 10 permite listar los archivos de acuerdo a lo definido en el requerimiento funcional SRF8.

Cliente

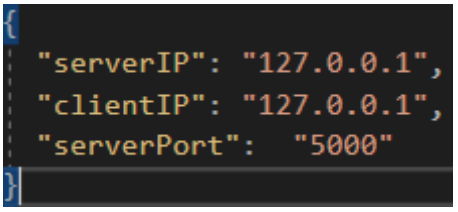


- La opción 1 permite dar de alta un tema, con nombre y descripción (CRF2).
- La opción 2 permite dar de alta una nueva publicación que contiene nombre, contenido, y temas. Solo se dará de alta una publicación si los temas existen previamente y si la publicación es única. (CRF4).
- La opción 3 permite subir un archivo a una publicación. Si la publicación existe se envía el path del archivo, en caso contrario se recibe un mensaje indicando que no existe la publicación (CRF7).
- La opción 4 permite dar de baja un tema siempre y cuando exista y no esté asociado a un post. Se controla que otro cliente no esté borrando/modificando el tema al mismo tiempo. (CRF3)
- La opción 5 permite modificar un tema siempre y cuando exista. Se controla que otro cliente no esté borrando/modificando el tema al mismo tiempo. (CRF3)
- La opción 6 permite dar de baja un post siempre y cuando exista. Se controla que otro cliente no esté borrando/modificando el post al mismo tiempo. (CRF5)
- La opción 7 permite modificar un post siempre y cuando exista. También permite asociar nuevos temas al mismo. Se controla que otro cliente no esté borrando/modificando el post al mismo tiempo. (CRF5 y CRF6)

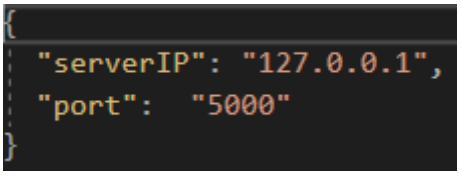
Archivos de configuración

Tanto el cliente como el servidor contienen archivos de configuración, los cuales permiten configurar para el servidor su ip y puerto, y para el cliente su ip, y la ip y el puerto del servidor. Esto permite que el cliente y el servidor puedan estar en terminales diferentes dentro de la misma red.

Archivo de configuración del cliente



Archivo de configuración del servidor



Obligatorio 3

Cambios en el alcance de la aplicación

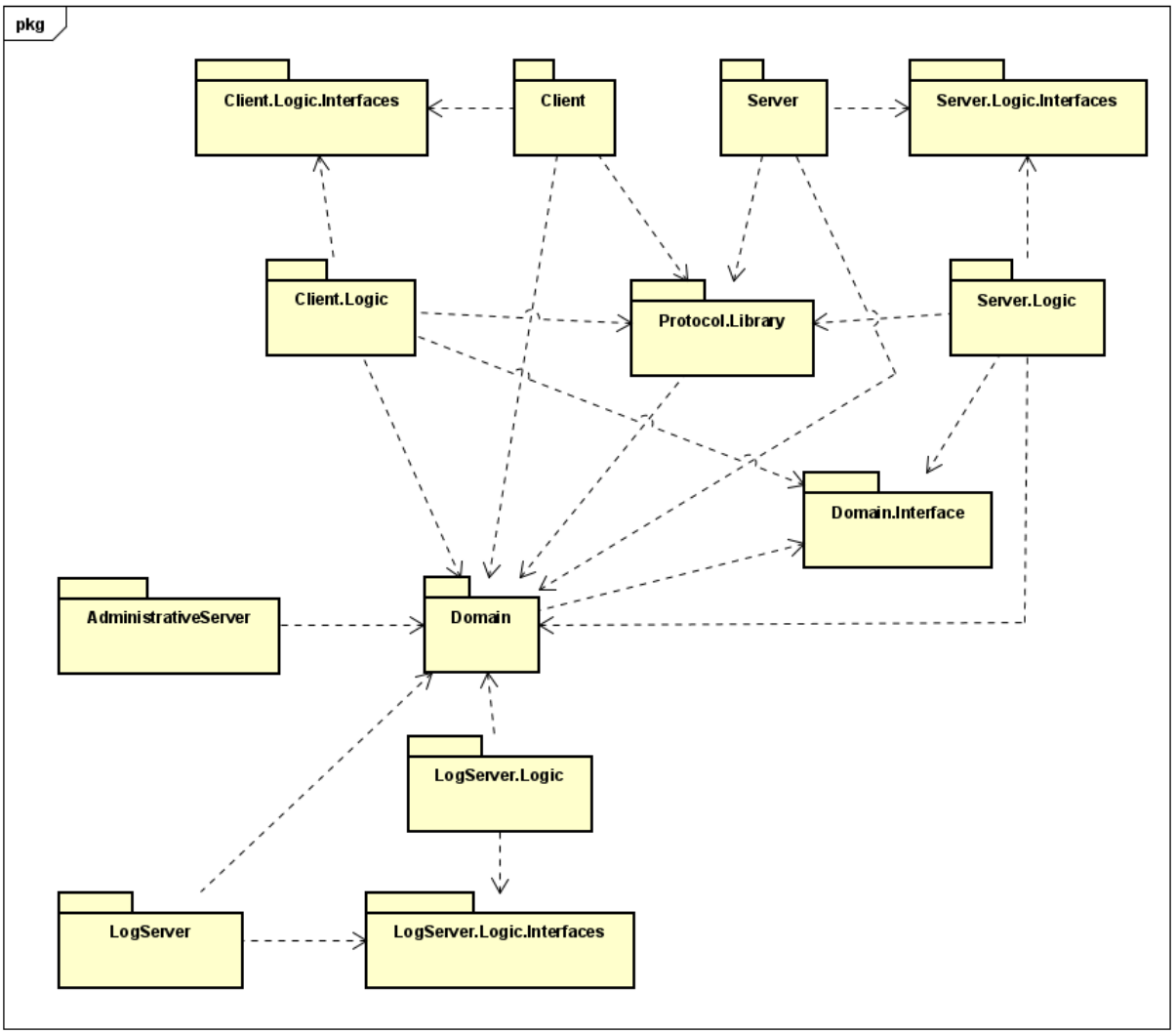
A lo realizado en el obligatorio 1 se le agrega un RESTful web service (conocido como Web API en .NET Core) encargado de exponer operaciones para obtención, alta, baja y modificación de publicaciones y temas, y una Web API encargada de mostrar un registro de las actividades anteriormente mencionadas (logs).

Cambios en el diseño y arquitectura del sistema

La arquitectura sigue siendo de cliente-servidor, extendiéndose para agregar un nuevo cliente (una Web API) y para loguear distintos eventos, agregarlos a una cola de mensajes, y consumir estos mensajes desde otra Web API. Para soportar los cambios mencionados anteriormente, se extiende la arquitectura mencionada en el obligatorio 1. No hay cambios en el cliente ni en los protocolos utilizados en el anterior obligatorio, ya que siguen funcionando de la misma forma y son compatibles con el nuevo cliente agregado en este obligatorio.

Con respecto al servidor, se realizan cambios en su arquitectura para poder agregar los logs a una cola de mensajes (en este caso RabbitMQ) y se lo extiende para que pueda soportar peticiones utilizando gRPC, con lo que puede soportar al nuevo cliente, que también utiliza gRPC para comunicarse con el servidor. Este se encarga de enviar peticiones al servidor para la obtención, alta, baja y modificación de temas y publicaciones del sistema, siendo una Web API que actúa como intermediario entre alguien que accede a ella y el servidor del anterior obligatorio. Por último, se agrega una última Web API que consume los mensajes agregados a la cola de mensajes de RabbitMQ, y los muestra pudiendolos filtrar por tema, publicación o fecha de creación del mensaje.

Nuevo diagrama de paquetes

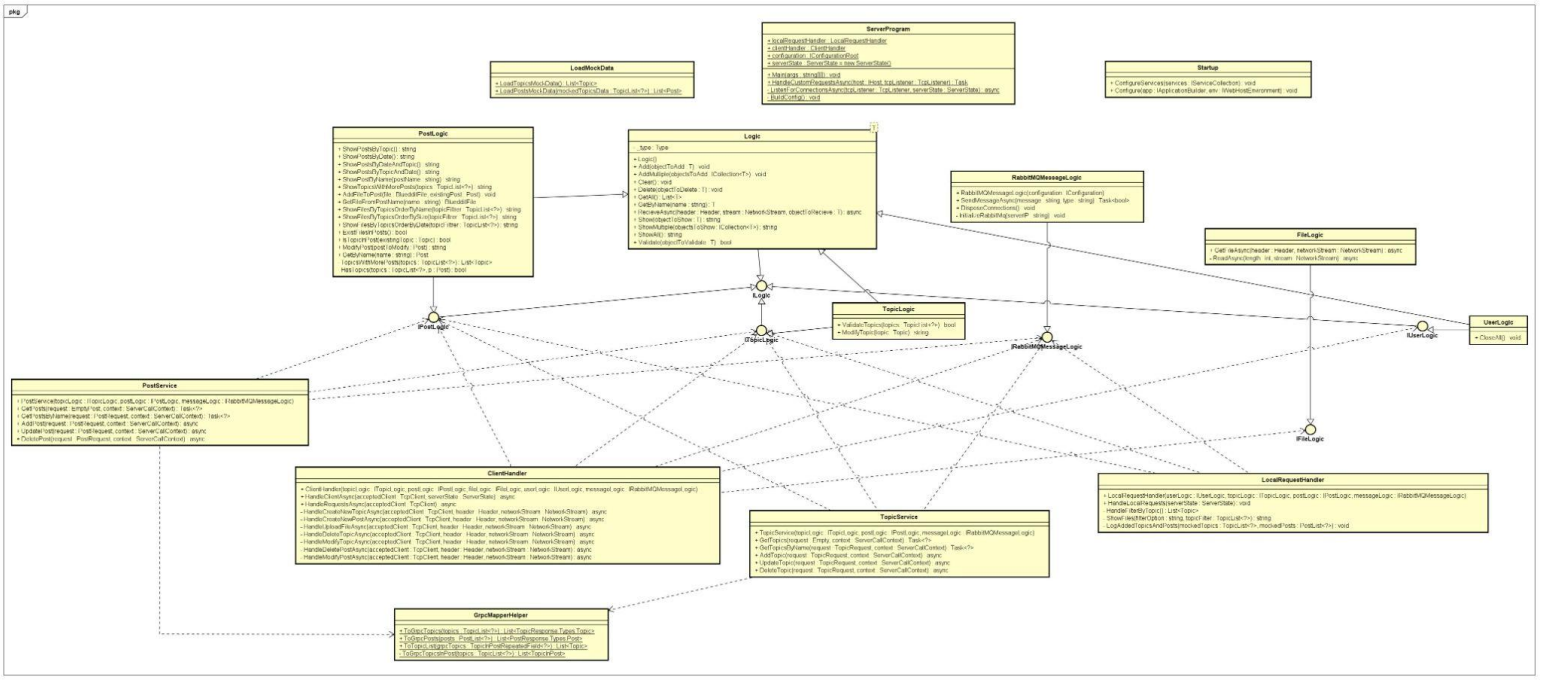


En este diagrama se muestra que la parte correspondiente al anterior obligatorio permanece incambiada, y las Web APIs creadas (**AdministrativeServer** y **LogServer**) solo comparten una dependencia con el dominio. A continuación detallaremos en las secciones **servidor**, **AdministrativeServer** y **LogServer**, la arquitectura de cada uno de ellos.

Servidor

El servidor sigue estando compuesto por tres paquetes propios y tres paquetes compartidos con el cliente. Los cambios realizados fueron dentro del paquete **Server**. Sigue soportando múltiples solicitudes del cliente del anterior obligatorio, a lo que se le agrega soporte de solicitudes gRPC. Gracias a la implementación anterior, que manejaba el almacenamiento de datos de la aplicación en memoria en la lógica y utilizaba inyección de dependencias para cumplir con el principio de inversión de dependencias, se pudo fácilmente inyectar estas nuevas dependencias a los nuevos servicios de gRPC para que la memoria pudiera ser compartida entre el servicio de este obligatorio y los nuevos servicios de gRPC. A partir del diagrama de clases del servidor detallaremos más en profundidad lo anteriormente mencionado, así como el manejo de cola de mensajes.

Nuevo diagrama de clases del servidor



Como se puede ver en este diagrama, los servicios gRPC **PostService** y **TopicService**, reciben por parámetro a las interfaces utilizadas para el anterior obligatorio, lo que permite que compartan la memoria dinámica entre ellos y con los tasks encargados de atender a los clientes del obligatorio 1. Esto se logra haciendo que la lógica sea inyectada como singleton por el inyector de dependencias, para que haya solo una instancia de estas y por ende la memoria se comparta (recordemos que la memoria es manejada dentro de las clases de lógica). A continuación se muestra una imágen del código para ver cómo se implementó:

```
public class ServerProgram
{
    public static LocalRequestHandler localRequestHandler;
    public static ClientHandler clientHandler;
    public static IConfigurationRoot configuration;
    public static ServerState serverState = new ServerState();
    public static void Main(string[] args)
    {
        Console.WriteLine("Server esta iniciando...");

        BuildConfig();

        var serverIP = configuration.GetSection("serverIP").Value;
        var port = Convert.ToInt32(configuration.GetSection("port").Value);

        var tcpListener = new TcpListener(IPAddress.Parse(serverIP), port);
        tcpListener.Start(10);

        var host = Host.CreateDefaultBuilder(args).ConfigureServices((context, services) =>
            {
                services.AddSingleton<IRabbitMQMessageLogic, RabbitMQMessageLogic>();
                services.AddSingleton<IUserLogic, UserLogic>();
                services.AddSingleton<ITopicLogic, TopicLogic>();
                services.AddSingleton<IPostLogic, PostLogic>();
                services.AddSingleton<IFileLogic, FileLogic>();
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            })
            .Build();

        _ = HandleCustomRequestsAsync(host, tcpListener);

        host.Run();

    }
```

Aquí la configuración de gRPC se realiza en la clase Startup, y antes de comenzar a ejecutar los servicios gRPC, se crea un nuevo task para atender a los clientes del anterior obligatorio.

En el diagrama también se puede ver que se agregó una nueva interfaz y clase de lógica respectivamente: **IRabbitMQMessageLogic** y **RabbitMQMessageLogic.** Esta clase es la encargada de inicializar RabbitMQ y crear la cola de mensajes utilizada para enviarlos (llamada **log_queue**) y loguear los distintos eventos relevantes del sistema, a la hora de crear, actualizar o borrar publicaciones o temas. Una instancia de esta clase es inyectada tanto a los tasks que atienden a los clientes del primer obligatorio como a los servicios gRPC para que puedan loguear lo que sea necesario. A continuación se muestran los aspectos más relevantes de esta implementación:

```
private void InitializeRabbitMq(string serverIP)
{
    var factory = new ConnectionFactory()
    {
        HostName = serverIP
    };
    _connection = factory.CreateConnection();
    _channel = _connection.CreateModel();
    _channel.QueueDeclare(queue: "log_queue",
        durable: false,
        exclusive: false,
        autoDelete: false,
        arguments: null);
}
```

La función encargada del envío de mensajes asíncronos es la siguiente:

```
public Task<bool> SendMessageAsync(string message, string type)
{
    bool returnVal;
    var log = new Log
    {
        LogType = type,
        Message = message,
        CreationDate = DateTime.Now
    };
    try
    {
        var logJson = JsonConvert.SerializeObject(log);
        var body = Encoding.UTF8.GetBytes(logJson);
        _channel.BasicPublish(exchange: "",
            routingKey: "log_queue",
            basicProperties: null,
            body: body);
        returnVal = true;
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
        returnVal = false;
    }

    return Task.FromResult(returnVal);
}
```

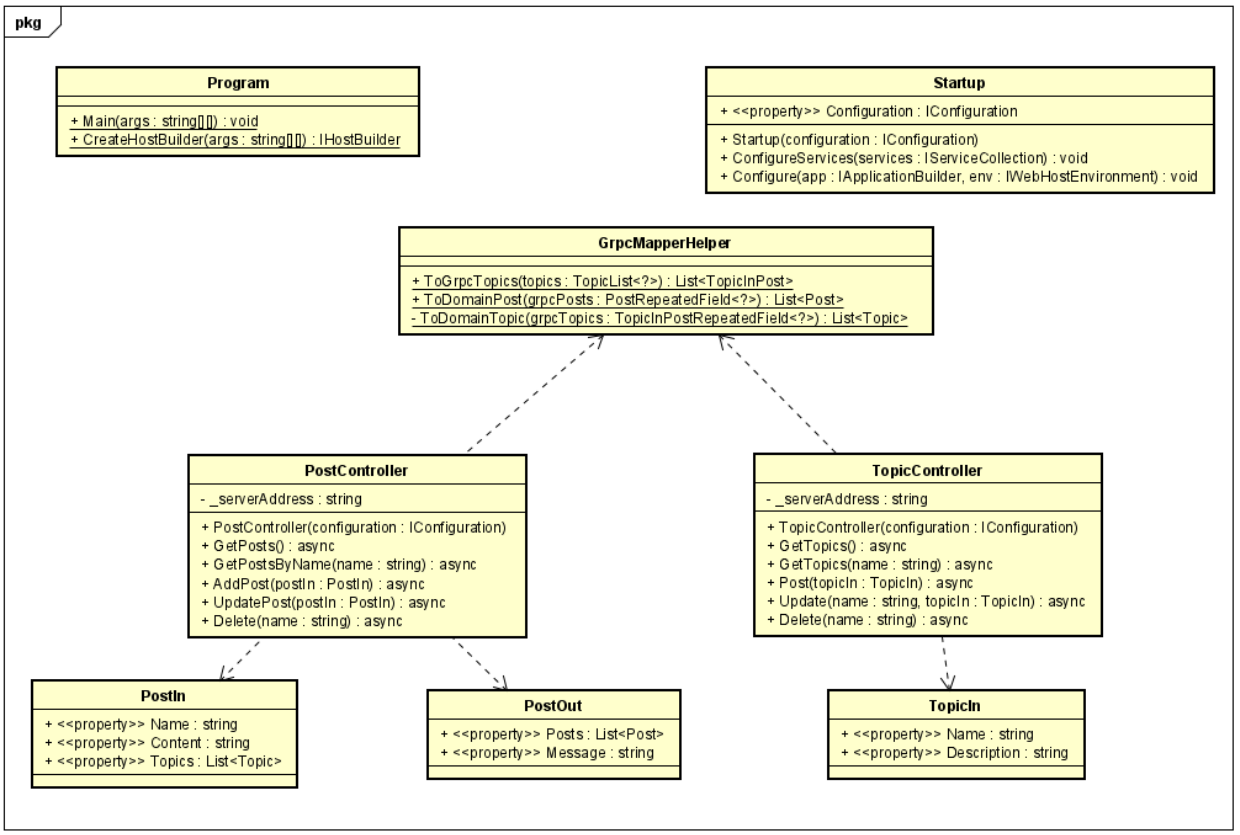
Vale destacar que estos cambios siguen los principios SOLID y de inversión de dependencias mencionados en el informe del anterior obligatorio, y que algunas clases presentes en el diagrama de clases del servidor del obligatorio pasado (las pertenecientes

al paquete **Protocol.Helpers**, fueron omitidas del diagrama por no haber sufrido cambios relevantes).

Administrative Server

Esta Web Api es la encargada de comunicarse con el server detallado en la parte anterior mediante requests gRPC, y también de exponer mediante la utilización de REST, endpoints que permiten el acceso al server.

Diagrama de clases del administrative server

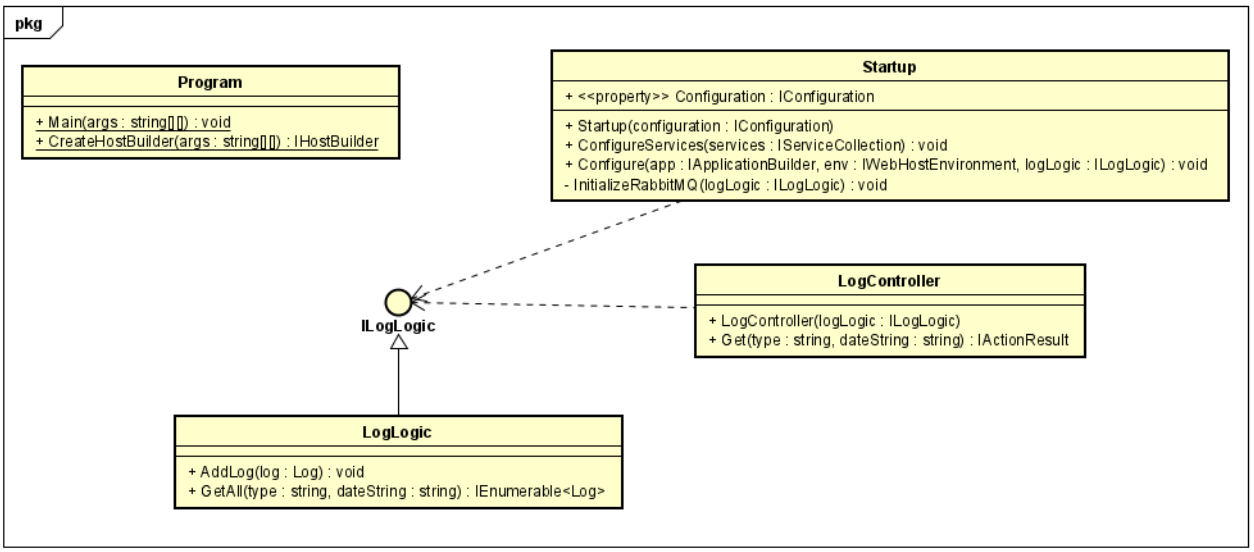


Como se puede ver aquí, el administrative server está compuesto por dos controllers, **TopicController** y **PostController,** que se encargan de las operaciones de retornar, dar de alta, de baja y modificar publicaciones y temas. Estos al recibir la solicitud de un cliente mediante el protocolo REST, utilizan gRPC para comunicarse con el server y así devolverle una respuesta al cliente. Las clases **PostIn**, **PostOut** y **TopicIn** son modelos de transferencia de datos que se utilizan para enviar o recibir data con un cliente REST. También ambos utilizan una clase helper llamada **GrpcMapperHelper**.

Log Server

Esta Web API se encarga de conseguir los mensajes de la cola de mensajes de RabbitMQ y mostrarlos a los distintos clientes mediante filtros por tema, publicación y fecha.

Diagrama de clases del log server



El log server está compuesto por un solo controller (**LogController**), cuyo único endpoint es el encargado de devolver los logs de acuerdo a los filtros seleccionados. Los logs son obtenidos de la cola de mensajes en el método **InitializeRabbitMQ**, ejecutado en la clase **Startup**, la cual corre al iniciar la aplicación.

```
private void InitializeRabbitMQ(ILogLogic logLogic)
{
    var rabbitIP = Configuration.GetValue<string>("rabbitIP");
    var factory = new ConnectionFactory() { HostName = rabbitIP };
    var connection = factory.CreateConnection();
    var channel = connection.CreateModel();

    channel.QueueDeclare("log_queue",
        false,
        false,
        false,
        null);

    var consumer = new EventingBasicConsumer(channel);
    consumer.Received += (sender, eventArgs) =>
    {
        var body = eventArgs.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        var log = JsonSerializer.Deserialize<Log>(message);
        logLogic.AddLog(log);
    };

    channel.BasicConsume("log_queue", true, consumer);
}
```

Al **LogController** se le inyecta una instancia de la clase **LogLogic**, la cual guarda en memoria los distintos logs y permite filtrarlos para devolverlos mediante el controller. Esta clase se inyecta mediante inyección de dependencias al controller siendo un singleton, ya que cuando se recibe un nuevo log, se guarda en la lógica y el controller debe de poder acceder a él.

Decisiones

Cambios en el manejo de concurrencia

A diferencia de lo realizado en el primer obligatorio, en donde se utilizaron threads para atender múltiples conexiones, en este obligatorio se pasó al modelo asincrónico. Esto se hizo por razones de performance, ya que generar nuevos threads es mucho más costoso en recursos de la CPU que utilizar el modelo asincrónico. También se hace difícil de debuggear, porque los threads están corriendo en paralelo, lo cual hace que el debugger se comporte de forma extraña. También el modelo asincrónico provee ciertas ventajas, como la escalabilidad, ya que el tener threads disponibles para atender más requests le da la posibilidad a las aplicaciones de escalar mejor. Otra ventaja es que es mucho más fácil saber cuándo se va a cambiar de un task a otro que lograr mecanismos de sincronización exitosos para que los threads funcionen correctamente.

A la hora de manejar desde el servidor las consultas gRPC y las consultas locales, se utilizó el modelo asincrónico para crear una nueva task para tomar las solicitudes internas del servidor.

Estándares seguidos para crear las REST APIs

Los estándares seguidos para crear las REST APIs fueron los siguientes:

- Se organizó la API de acuerdo a los recursos existentes en el sistema. Por ejemplo, los controllers creados en el **AdministrativeServer** y en el **LogServer** se basaron en recursos existentes en el sistema, como la clase de dominio **Post**, **Topic** o **Log**.
- Las URLs se basan en sustantivos y no en verbos. Por ejemplo: *{{AdministrativeServerURL}}/post/{{nombre del post}}*
- Se utilizaron métodos comunes a la mayoría de las APIs REST, como ser GET, POST, PUT y DELETE.
- Se utilizaron códigos de status HTTP para devolver respuestas al usuario.

Códigos utilizados

Los códigos utilizados a la hora de la devolución de respuestas fueron los siguientes:

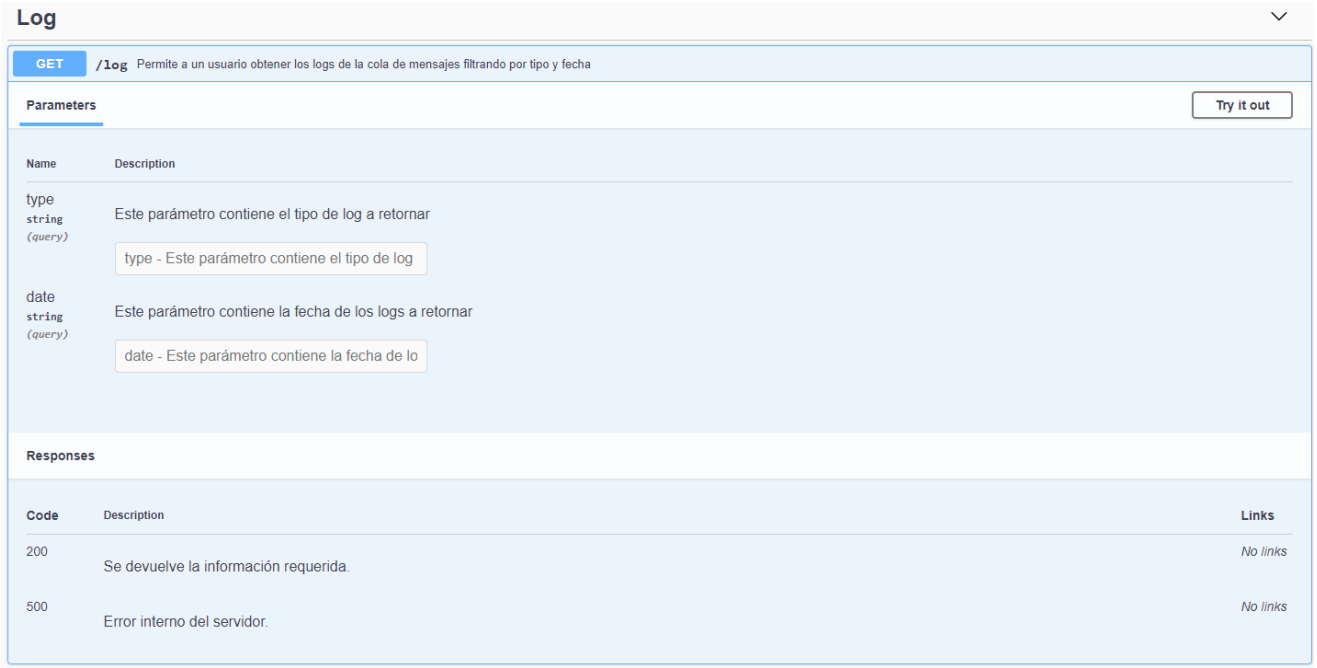
- Status code OK: 200. La acción fue ejecutada correctamente.
- Bad Request: 400. El servidor no puede o no procesa el request debido a un error del cliente.
- Internal Server Error: 500. Indica un error en el servidor al ejecutar cierta acción.

Funcionamiento de la aplicación

Con respecto a lo realizado en el obligatorio 1, no hubieron cambios en el funcionamiento de la aplicación. Las dos Web APIs agregadas y detalladas en secciones anteriores en este obligatorio, exponen endpoints que serán detallados a continuación.

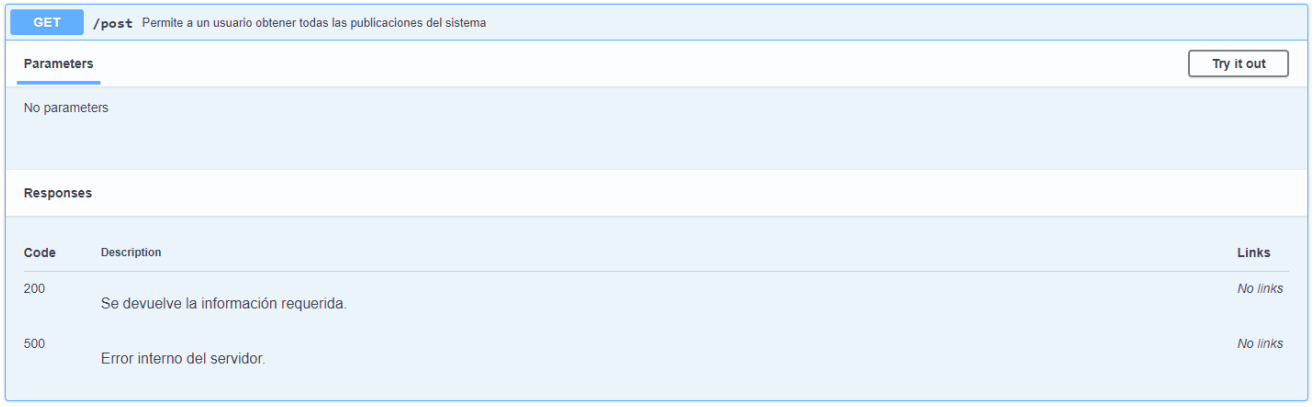
Especificación de endpoints

Log Server



Administrative Server

- PostController



POST `/post` Permite a un usuario agregar una nueva publicación al sistema

Parameters | Try it out

No parameters

Request body | application/json

Este parámetro contiene la información de la publicación a agregar

Example Value | Schema

```
{
  "name": "string",
  "content": "string",
  "topics": [
    {
      "name": "string",
      "description": "string"
    }
  ]
}
```

Responses

Code	Description	Links
200	Se devuelve la información requerida.	*No links*
400	Error de solicitud del cliente Media type: text/plain Example Value | Schema	*No links*
500	Error interno del servidor.	*No links*

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

PUT `/post` Permite a un usuario actualizar una publicación del sistema

Parameters | Try it out

No parameters

Request body | application/json

Este parámetro contiene la información de la publicación a actualizar

Example Value | Schema

```
{
  "name": "string",
  "content": "string",
  "topics": [
    {
      "name": "string",
      "description": "string"
    }
  ]
}
```

Responses

Code	Description	Links
200	Se devuelve la información requerida.	*No links*
400	Error de solicitud del cliente Media type: text/plain Example Value | Schema	*No links*
500	Error interno del servidor.	*No links*

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

GET `/post/{name}` Permite a un usuario obtener una publicación con el nombre que se pase por parámetro

Parameters | Try it out

Name	Description
name * required string *(path)*	Este parámetro contiene el nombre del post a retornar name - Este parámetro contiene el nombre d

Responses

Code	Description	Links
200	Se devuelve la información requerida.	*No links*
400	Error de solicitud del cliente Media type: text/plain Example Value | Schema	*No links*
500	Error interno del servidor.	*No links*

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

DELETE `/post/{name}` Permite a un usuario borrar una publicación del sistema

Parameters | Try it out

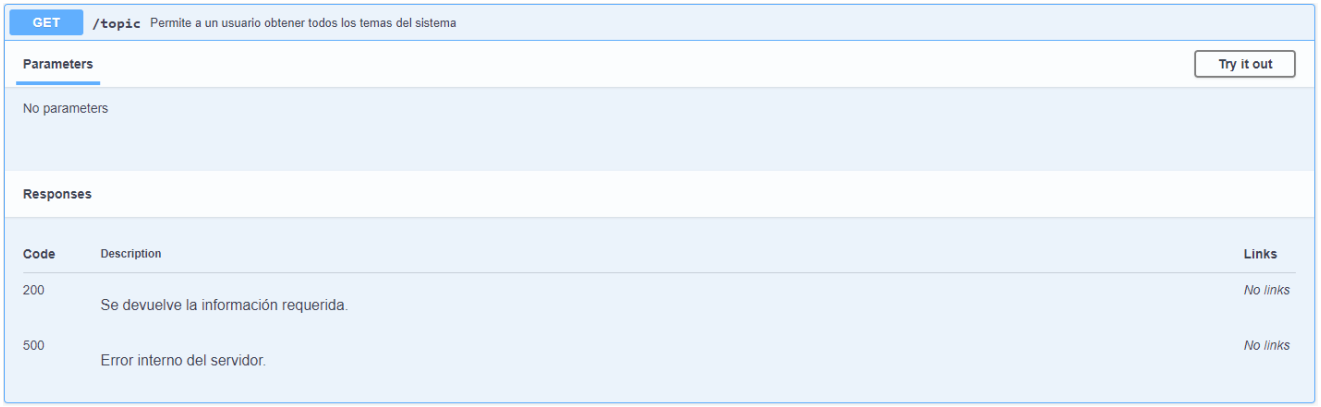
Name	Description
name * required string *(path)*	Este parámetro contiene el nombre de la publicación a borrar name - Este parámetro contiene el nombre d

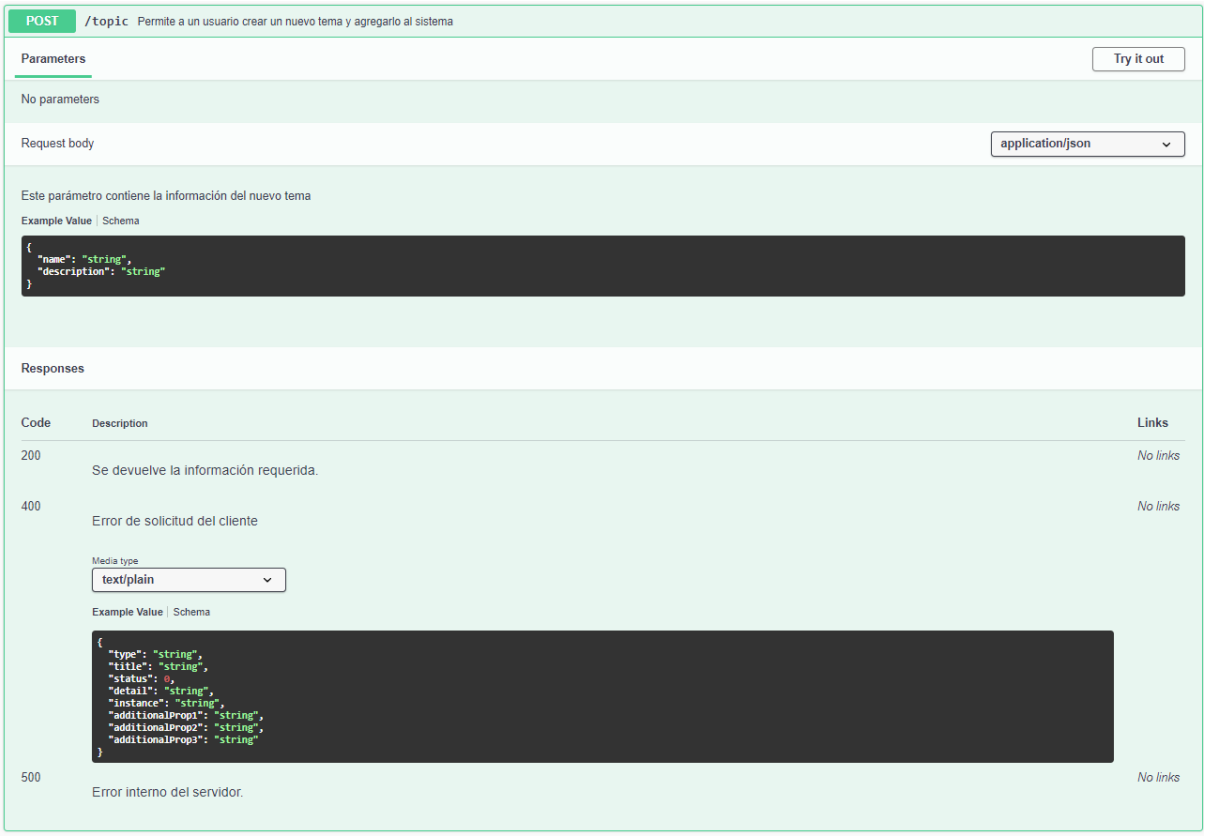
Responses

Code	Description	Links
200	Se devuelve la información requerida.	*No links*
400	Error de solicitud del cliente Media type: text/plain Example Value | Schema	*No links*
500	Error interno del servidor.	*No links*

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

- TopicController





GET `/topic/{name}` Permite a un usuario obtener un tema del sistema mediante su nombre ingresado por parámetro

Parameters | Try it out

Name	Description
name * required string *(path)*	Este parámetro contiene el nombre del tema a obtener name - Este parámetro contiene el nombre d

Responses

Code	Description	Links
200	Se devuelve la información requerida.	*No links*
400	Error de solicitud del cliente Media type: text/plain Example Value | Schema	*No links*
500	Error interno del servidor.	*No links*

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

PUT `/topic/{name}` Permite a un usuario actualizar un tema del sistema

Parameters | Try it out

Name	Description
name * required string *(path)*	Este parámetro contiene el nombre del tema a actualizar name - Este parámetro contiene el nombre d

Request body: application/json

Este parámetro contiene la información del tema a actualizar

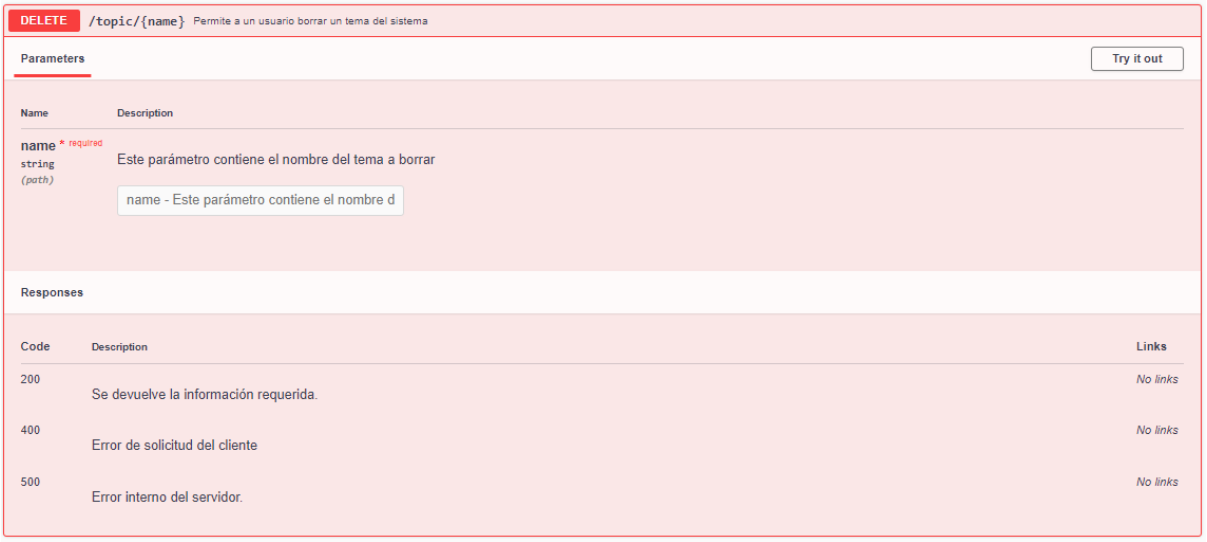
Example Value | Schema

```
{
  "name": "string",
  "description": "string"
}
```

Responses

Code	Description	Links
200	Se devuelve la información requerida.	*No links*
400	Error de solicitud del cliente Media type: text/plain Example Value | Schema	*No links*
500	Error interno del servidor.	*No links*

```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

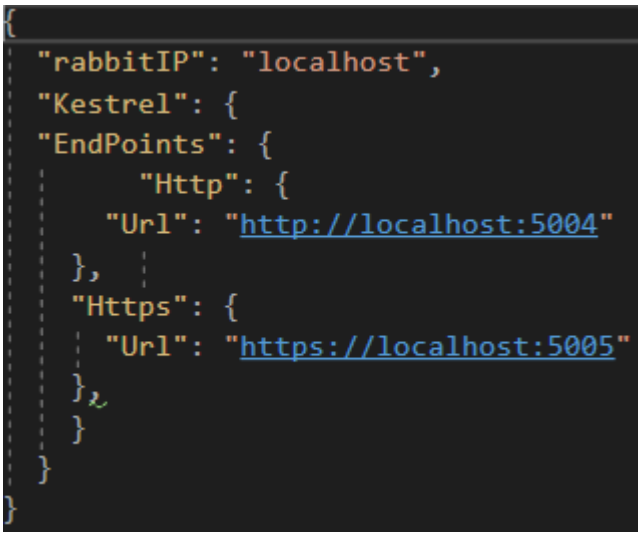

DELETE
/topic/{name}
Permite a un usuario borrar un tema del sistema
Parameters
Try it out
Name
Description
name * required
string
(path)
Este parámetro contiene el nombre del tema a borrar
name - Este parámetro contiene el nombre d
Responses
Code
Description
Links
200
Se devuelve la información requerida.
No links
400
Error de solicitud del cliente
No links
500
Error interno del servidor.
No links

Archivos de configuración

A los archivos de configuración definidos en el obligatorio 1 (del cual se especificará solo el del server que fue el único que sufrió modificaciones) se agregan los siguientes archivos:

Archivo de configuración del server

Archivo de configuración del log server



Archivo de configuración del administrative server

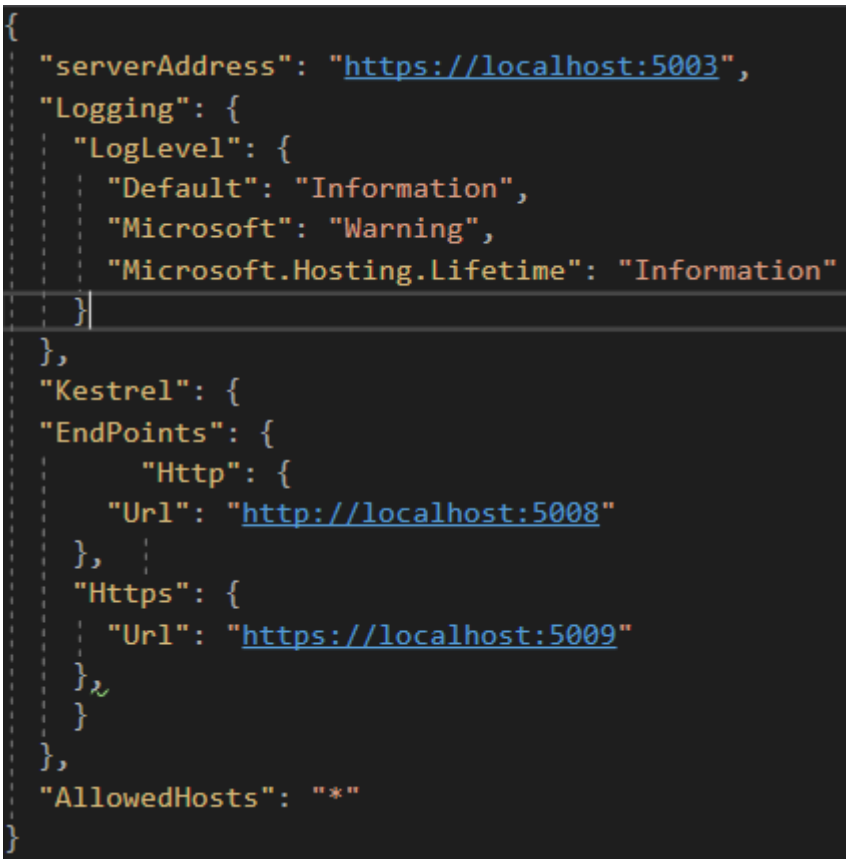


Diagrama de despliegue