Universidad ORT Uruguay

Facultad de Ingeniería

Programación de redes
Obligatorio 2

Jimena Sanguinetti(228322)
Joselen Cecilia(233552)

Entregado como requisito de la materia Programación de redes
Link al repositorio de GitHub
https://github.com/JoselenC/Obligatorio1-Programacion-De-Redes

21 de junio de 2021

Declaraciones de autoría

Nosotras, Joselen Cecilia y Jimena Sanguinetti, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Programación de redes ;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Índice general

1. Introducción

Se construye un sistema compuesto por; con un server administrativo encargado de procesar las request de grpc desde la web api y un server de logs encargado de procesar y guardar los logs del server principal el cual esta formado por; un server que responde al server administrativo y uno que responde a los client.

Cuando se inicia la aplicación cliente se presenta un menú con las distintas opciones que puede seleccionar el usuario del lado del cliente. Las opciones se dividen por Post, Tema, asociar Archivo y buscar post; dentro de post, se encuentran diferentes opciones como agregar, editar, eliminar y asociarles/desasociarles un tema. Al agregar un post, este se guarda en una lista en ManagerPostRepository junto al resto de los posts, al eliminarlo se borra de esta misma lista y al editarlo, se borra el actual y se vuelve a agregar con los datos actualizados. En caso de que el post esté en uso por otro cliente, al intentar eliminarlo o actualizarlo, no será posible. Por último, dado un post, el cliente puede tanto asociarle como desasociarle un tema existente en el sistema; en caso de que se desasocie un tema, se debe asociarle uno nuevo.

Dentro de los temas también es posible agregar, editar, eliminar con el mismo comportamiento que los posts pero en ManagerThemeRepository. Una vez agregado un tema, este puede ser luego asociado a un post.

Con respecto a los archivos, se debe elegir un post para asociarle a un archivo; el archivo hereda los temas del post y este se guarda en ManagerRepository.

Por último, en la opción de buscar un post, se selecciona uno de la lista y se muestran todos sus datos.

Por otro lado, cuando se inicia la aplicación servidor, también se presenta un menú pero con las opciones que se pueden seleccionar del lado del servidor. Estas se dividen por listar clientes, Post, Tema y Archivo; en la primera opción se muestran todos los clientes que están conectados al servidor, mostrando la ip, el puerto y fecha y hora en que se conectó. Con respecto a los posts, nuevamente se muestra un menú que presenta las opciones por las cual se desea filtrarlos; es posible listarlos ordenados según la fecha de creación, según un tema que se seleccione o por ambos filtros. Además, se puede seleccionar un post en particular y se muestran todos sus datos. También es posible seleccionar un post para el cual se muestran todos los datos del archivo asociado.

Dentro de los temas es posible mostrar un listado de todos los que existen dentro del sistema. Además se puede mostrar el tema que tiene mayor cantidad de posts asociados. En caso de existir más de uno con la mayor cantidad, se muestran todos

ellos.

Por último, también es posible mostrar una lista de todos los archivos; una opción es mostrar la lista completa pero también se puede filtrar por tema, por fecha de creación, por nombre o por tamaño.
También se levanta al correr esta aplicación el servidor grpc que se encarga de procesar las request de grpc.

Cuando se inicia el servidor de logs este queda esperando para recibir los logs que se van a e enviar desde el servidor principal, y muestra con un mensaje por consola en que cola se esta escuchando.

Por ultimo tenemos la Web API la cual luego de establecida la conexión nos permite mandar las request a grpc para agregar un nuevo post, modificarlo, borrarlo, agregar un nuevo theme, modificarlo borrarlo, asociar un theme a un post y obtener los logs.

2. Descripción de la arquitectura

Task:

Tanto en la aplicación del cliente como del servidor fue necesario el uso de Task, esto se hizo para poder mantener varios subprocesos a la vez de forma asincrónica. Para esto fue necesario modificar la firma de los métodos a .ªsync Task.º .ªsync Task¡T¿z a la hora de invocarlos se utilizó el .ªwait"; además debimos cambiar tanto los métodos de envío y recibo de data como los de conexión a tareas asincrónicas, por ejemplo, readAsync y acceptTcpClientAsync.

async/await:

Mediante el await se puede esperar el resultado de las async Tasks ya que con este se pausa la ejecución hasta que termine la tarea y se obtenga un resultado.

Stream:

Para poder enviar y recibir data, ya sean packet o archivos fue necesario el uso de la clase NetworkStream de modo que se pueda acceder a los métodos y propiedades de la clase Stream (clase padre) que se utilizan para facilitar las comunicaciones de red. Una vez que se obtiene una instancia de esta clase, se llama al método Write para enviar data al servidor, mientras que el servidor llama al método Read para poder recibir la data que llega. Ambos métodos se bloquean mientras se esta realizando la operación; la operación de lectura se desbloquea cuando los datos han llegado y están disponibles para ser leídos.

TcpClient y TcpListner:

Para establecer la conexión y luego enviar y recibir datos entre varios clientes y el servidor utilizamos la clase TcpClient la cual nos proporciona los métodos necesarios para esto.
Luego para aceptar las conexiones enviadas por el cliente de forma asincrónica utilizamos el método AcceptTcpClientAsync de la clase TcpListener.

Grpc

Para la comunicación entre el server Administrativo, el server GRP y la WebAPI usamos GRPC, por defecto GRPC usa HTTP/2, por lo que debemos indicar

esto en el program del server

RabbitMQ

Para el envío y recibo de los logs se uso RabbitMQ, como intermediario para la comunicación. RabbitMQ implementa el protocolo mensajería de capa de aplicación AMQP (Advanced Message Queueing Protocol), el cual está enfocado en la comunicación de mensajes asíncronos con garantía de entrega.

lock

Se usaron locks para establecer zonas de mutua exclusión para sincronizar el acceso a la modificación y borrado de post y threads

3. Descripción de la API

Nuestra API se basa en REST, por lo cual, hemos definido un conjunto de resources (Recursos) en los cuales se basan nuestros endpoints.

Entendemos que para los requerimientos del sistema, con estos Resources es suficiente.

En las siguientes secciones se detallan los resources que existen en el sistema y el endpoint base de cada uno, para ver un detalle de todos los endpoints ir al Anexo.

3.0.1. Descripcion endpoints

Post

Representa un post el cual es un nombre, una fecha de creacion y un theme

- **Endpoint base:** DOMAIN/api/posts
- **Verbos HTTP:** POST - DELETE - PUT
- **Headers:** No Aplica

Theme

Este recurso representa un theme elcual es un nombre y una descripcion

- **Endpoint base:** DOMAIN/api/themes
- **Verbos HTTP:**POST - DELETE - PUT
- **Headers:** No Aplica

ThemeToPost

Este recurso representa la relacion entre post y theme es un nombre de post y un nombre de theme

- **Endpoint base:** DOMAIN/api/posts/theme
- **Verbos HTTP:** POST - PUT
- **Headers:** No Aplica

Log

Representa a los logs, que son un mensjae y una fecha de creacion.

- **Endpoint base:** DOMAIN/api/logs
- **Verbos HTTP:** GET
- **Headers:** No Aplica

3.1. Codigos

Pasamos a detallar como se manejan las distintas respuestas HTTP en nuestra API. Esto se hizo siguiendo las buenas practicas detalladas en libro .ªPIgee Web API desing the missing link"

Handling 2XX

- **200**: expresa que "todo salio bien", normalmente un GET satifactorio es un ejemplo de esta respuesta
- **201**: expresa que .el objeto se creo correctamente", normalmente se usa durante en los metodos POST

Handling 4XX

- **404**: expresa que "no se encontró el objeto o endpoint", este error puede ocurrir en un GET, PUT, DELETE
- **409**: expresa que "no se pudo procesar debido a un conflicto", ejemplo agregar un objeto que ya existe
- **422**: expresa que "la entidad no se puede procesar", esto son errores como un string vació o una fecha con formato incorrecto.

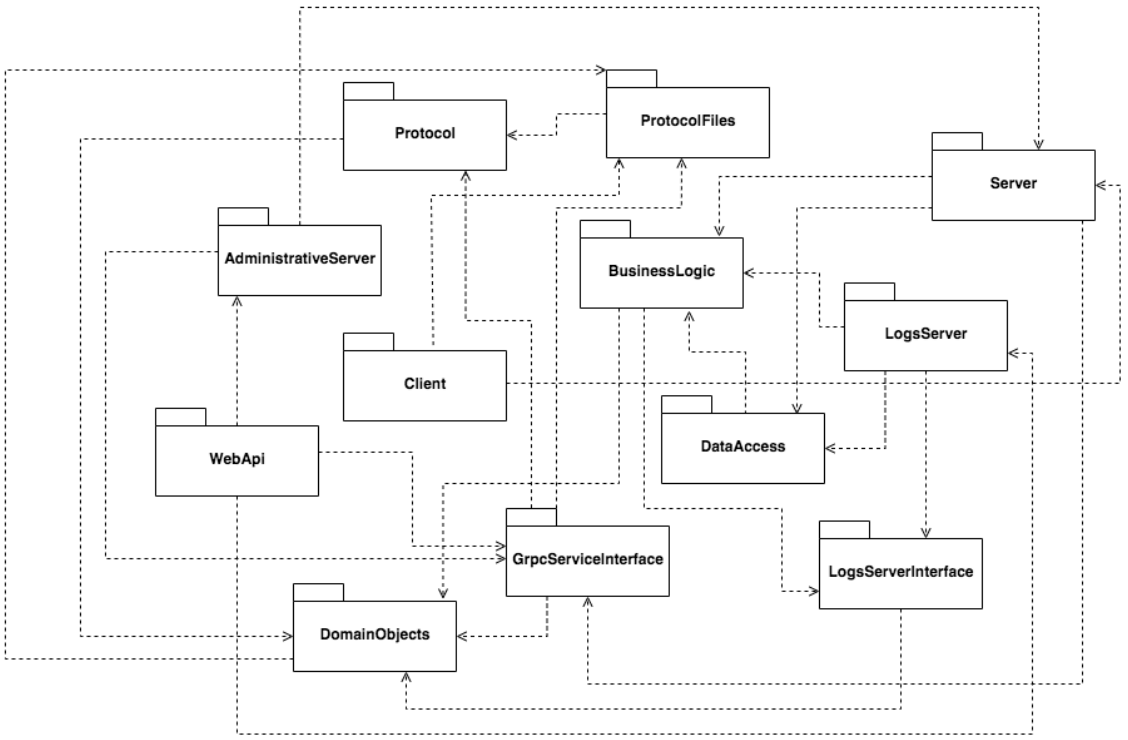
Handling 5XX

Se decidió controlar en un filtro todos las Excepciones y devolver siempre 500 y un mensaje genérico, de esta forma nos aseguramos no dejar posibles huecos de seguridad en los cuales se pueda obtener información sensible a través de una vulnerabilidad en el sistema.

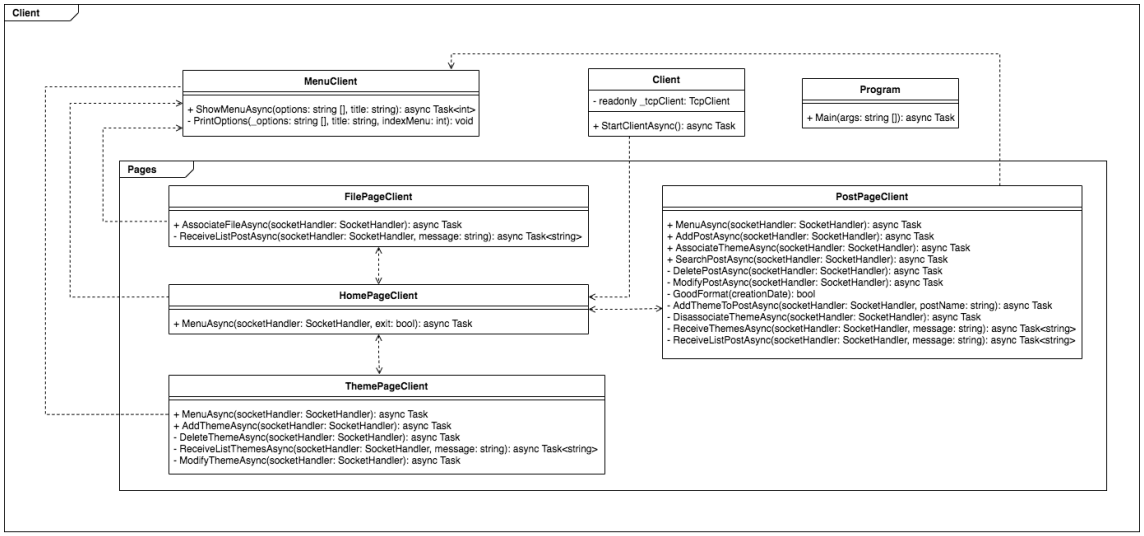
- **500**: Todo error no controlado por el sistema sera un 500.

4. Diseño detallado de cada uno de sus componentes

El diseño de nuestra aplicación se basa en varias capas las cuales tienen diferentes propósitos cada una, la división de las mismas fue definida justamente por la función que cumple dentro de la aplicación. Así mismo, estas capas también se subdividen, principalmente para aumentar la organización y facilitar el entendimiento. Estas son:



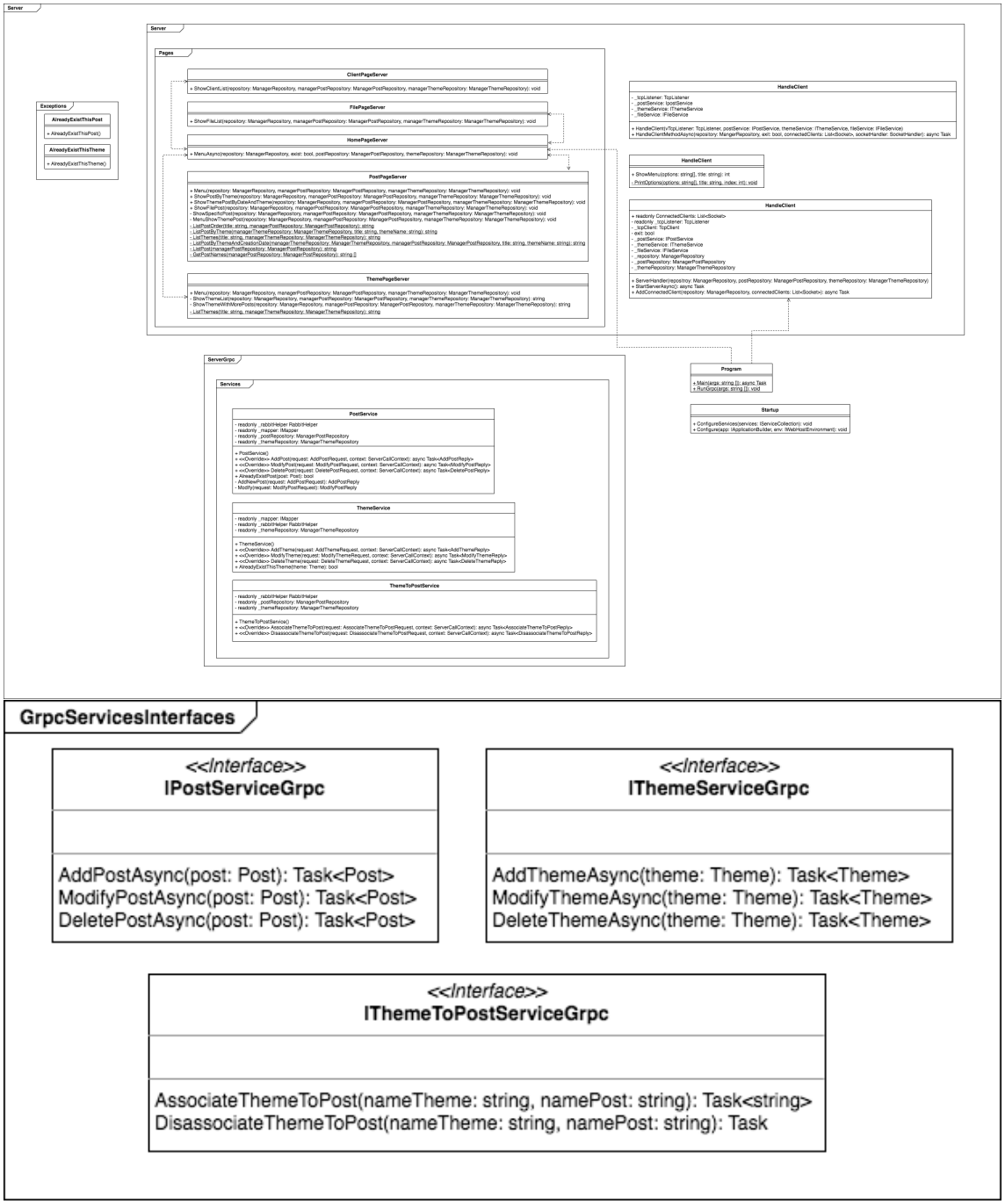
1. Client:



En esta capa podemos encontrar al cliente que se comunicará con el servidor, este es quien manda las diferentes requests correspondientes a las diferentes acciones que se pueden realizar. Es posible establecer comunicación entre el servidor y cuantos clientes se quieran, por lo tanto lo que se hace es, a cada cliente asignarle su socket y su endpoint correspondientes para luego poder identificarlo y así manejar las distintas respuestas que recibe de las distintas requests del lado del servidor.

Dentro del cliente podemos ver tres principales subdiviciones, contamos con la clase Program, en esta se llama a la clase connectionConfig donde se hace referencia a los puertos e IPs definidas en el appConfig para poder establecer la conexión; además se define el socket y su endpoint. Por otro lado tenemos un MenuClient donde se define como se va a mostrar el menú con las distintas opciones de las funciones que puede solicitar el cliente; decidimos que el menú se pueda controlar por el teclado de modo que sea más amigable para el usuario así como también para nosotras, dado que fue más sencillo de validar. Por último contamos con una carpeta llamada Pages, en esta podemos encontrar la lógica, dividida en cuatro clases diferentes: FilePageClient, HomePageClient, PostPageClient y ThemePageClient. Cada una de estas contiene todas las funcionalidades respecto a cada una de las entidades en cuestión (Post, File y Theme), además de HomePage la cual muestra el menú propio del cliente en el cual se delegan las diferentes tareas a las otras Pages; en estas se realizan las requests hacia el servidor.

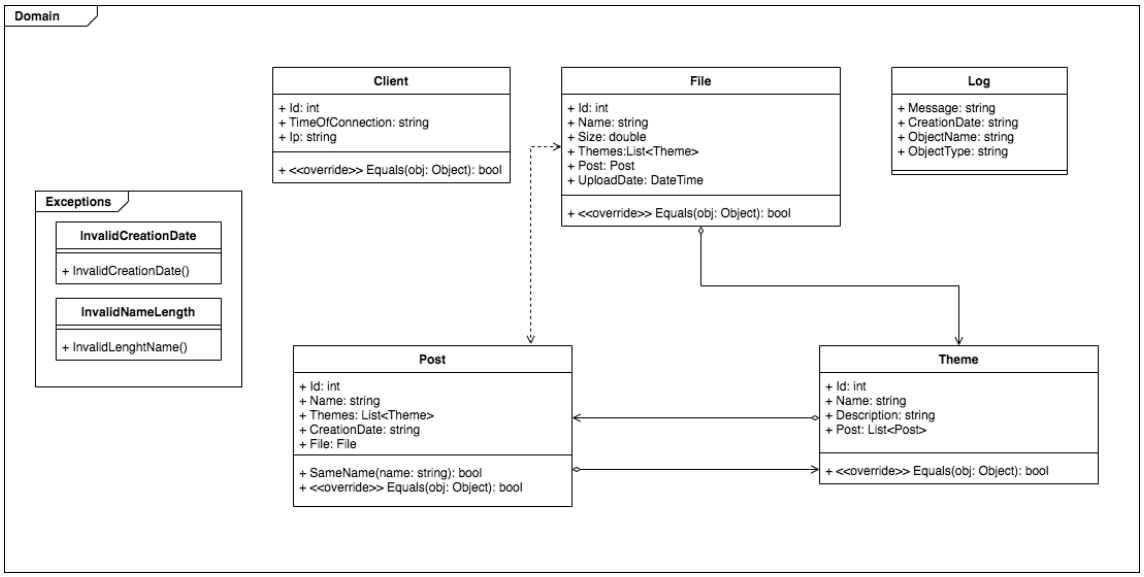
2. Server:



El diseño del lado del servidor es muy similar al del cliente; podemos encontrar la clase Program, MenuServer, la carpeta con las distintas pages, appConfig, connectionConfig, las cuales se encargan de la configuración de las conexiones. Por último tenemos una clase HandleClient. Esta clase es la que se encarga de manejar las request realizadas por cualquier cliente; podemos encontrar todas las diferentes funciones que el cliente puede solicitar y esta delega la implementación de la misma a los Service de cada objeto que se mencionan en el paquete del dominio.
La clase Program cumple la función de aceptar y manejar a los distintos clientes, además es quien se relaciona con el repositorio en memoria para guardar los clientes conectados con sus respectivos sockets. Este se encarga de lanzar un hilo donde se

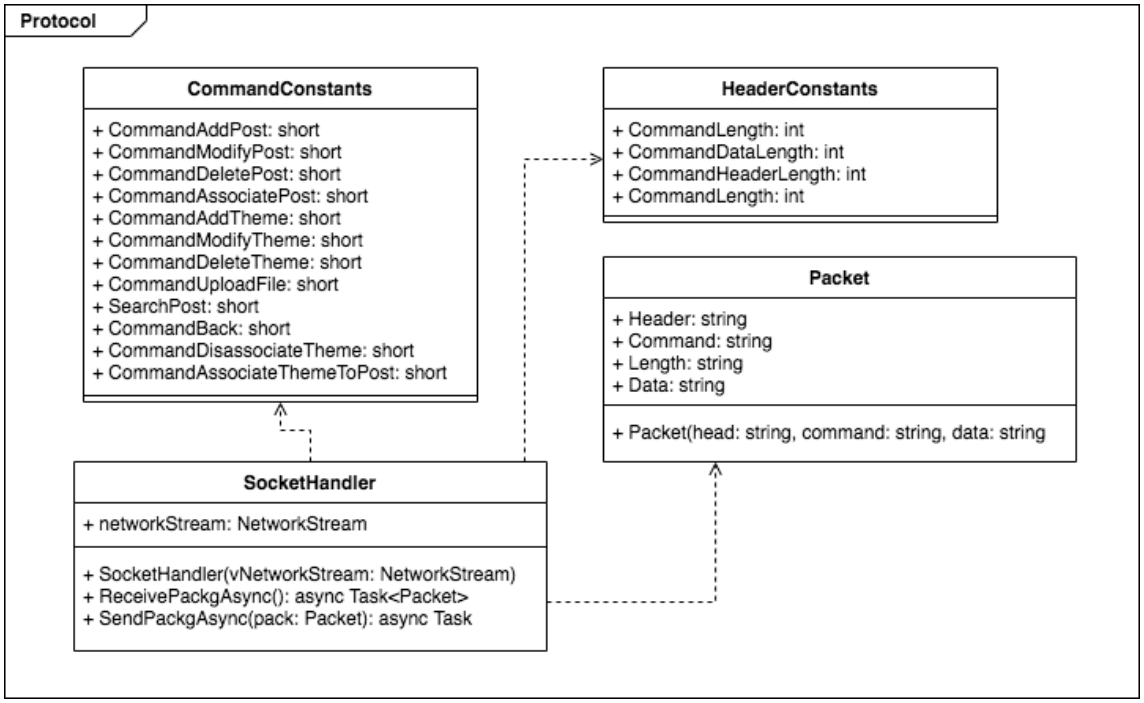
muestra el menú del servidor.

3. Domain:



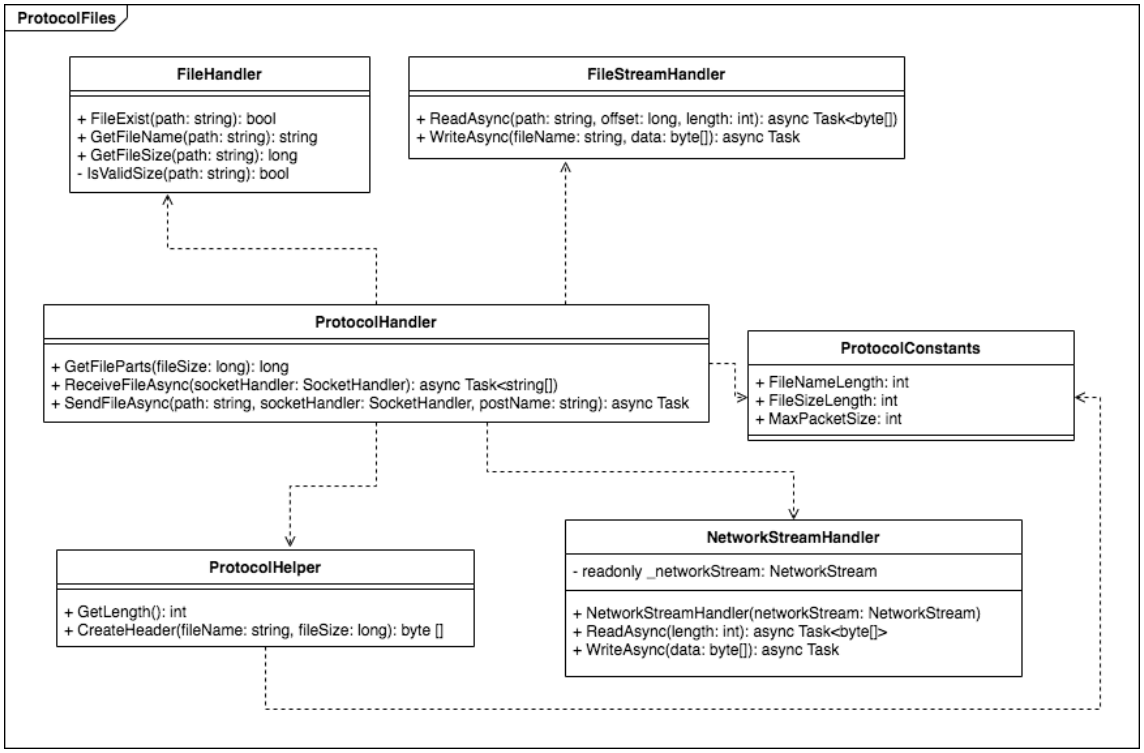
En el dominio podemos encontrar las distintas entidades en cuestión: File, Theme, Archive y Post; además contamos con la clase ClientConnection que contiene la ip, el puerto y la hora de conexión de cada cliente, la cual es utilizada para poder listar todos los clientes conectados al servidor, mostrando esos tres datos de cada uno. Por último tenemos el repositorio MemoryRepository que contiene las listas que deseamos guardar en memoria: Posts, Themes, Files y ClientConnections.
Además contamos con una carpeta Service que contiene un Service por cada entidad, en cada uno de estos encontramos una instancia del memoryRepository ya que estas clases se utilizan para el manejo de las listas en memoria.

4. Protocol:



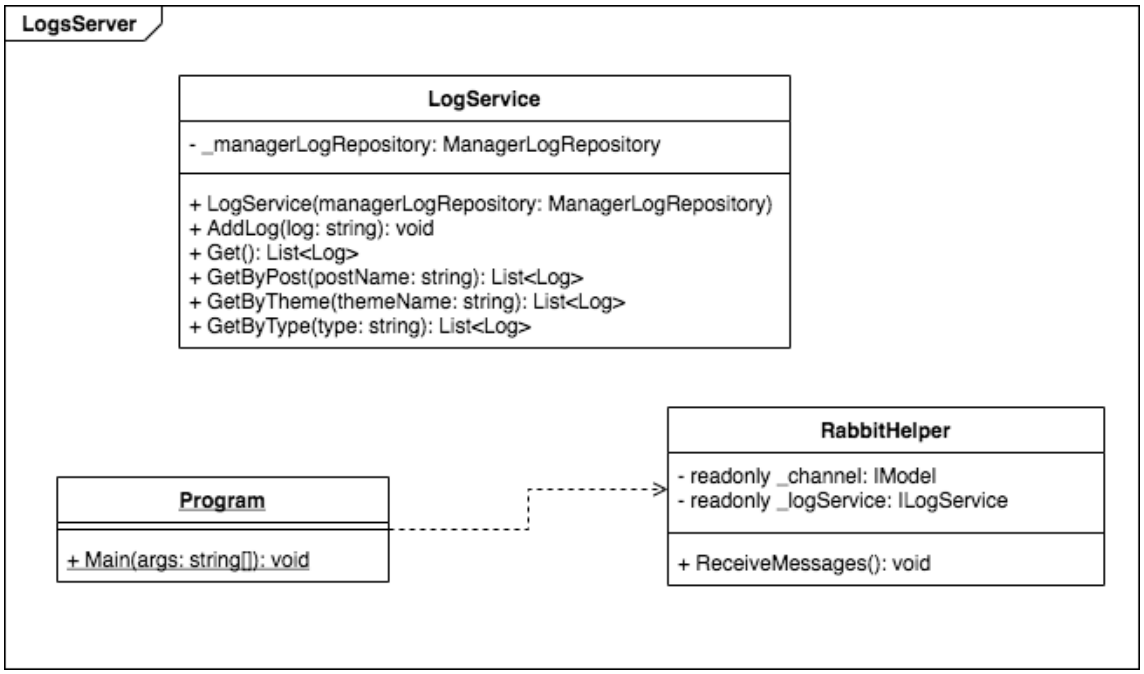
En esta capa se maneja el envío de datos mediante el protocolo definido por nosotras. Contamos con el recieve y send de los packages; estos son definidos dentro de ese paquete y están formados por un Header, un Command, un length y un Data, los cuales son explicados más adelante en la definición del protocolo. Por último se tienen las constantes que se utilizan para decidir el comportamiento frente a las requests.

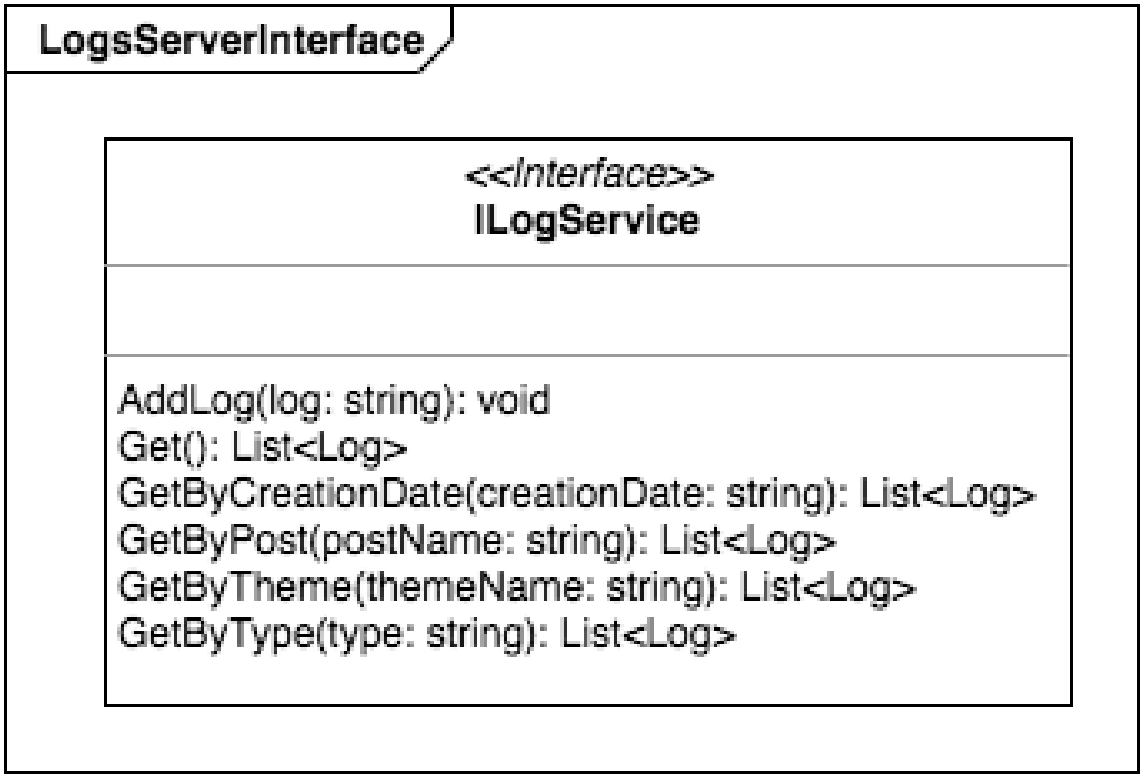
5. ProtocolFile:



En esta capa podemos encontrar todas las especificaciones del protocolo para el envío y la recepción de archivos; para esto fue necesario definir las clases: FileHandler, FileStreamHandler, ProtocolHandler, ProtocolHelper y ProtocolSpecification.

6. LogsServer:

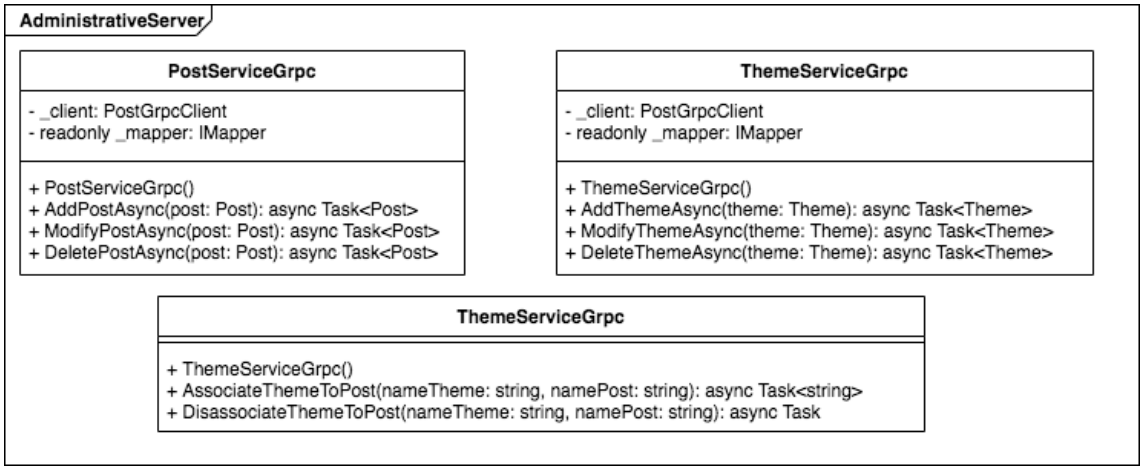




Este paquete es una REST API que expone un único endpoint que se utiliza para poder ver todos los logs en el sistema. En cada ocasión que un usuario realiza una cualquier acción, se genera un log de forma automática, con un mensaje. Esta API la consume el cliente administrativo. Además, se conecta con el repositorio de Logs, guardándose en base de datos en una tabla que los contiene a todos; esto decidimos ponerlo así de manera que estén todo juntos.

Además, contamos con otro paquete llamado LogsServerInterface que justamente lo que contiene es la interfaz que expone el contrato de las funciones que serán implementadas en LogsServer.

7. AdministrativeServer:

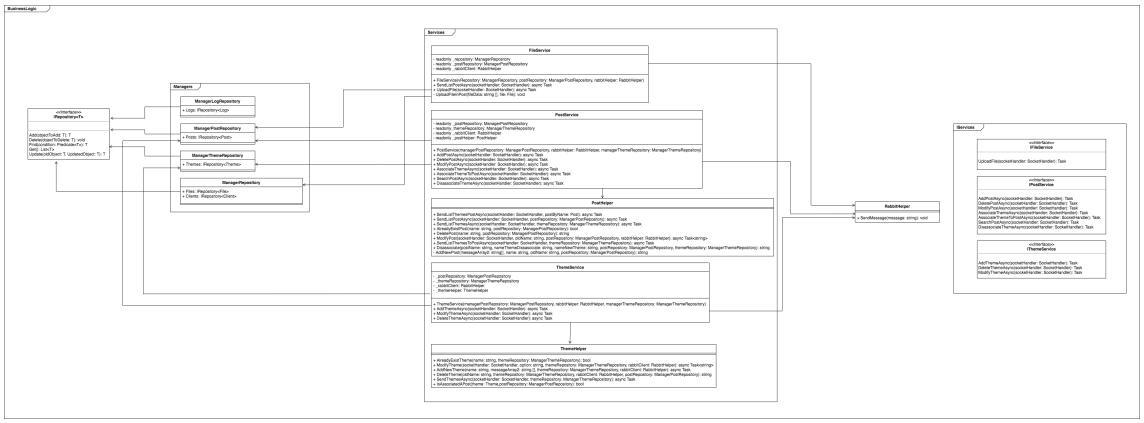


Este paquete utiliza la tecnología Google Remote Procedure Calls (Grpc), lo que esto nos permite es tener una aplicación cliente que pueda llamar a un metodo de la aplicación servidor directamente pero puede ser en una máquina diferente, como si fuese local. Esto nos facilita la creación de aplicaciones y servicios distribuidos.

La idea principal es definir un servicio, como son PostServiceGrpc y ThemeServiceGrpc en nuestro caso, los cuales especifican los metodos que se pueden llamar de forma remota con sus parametros y tipos de retorno. Del lado del servidor, este implementa dicha interfaz y luego ejecuta un servidor Grpc que maneja las distintas llamadas de los distintos clientes. Por otro lado, el cliente tiene un código auxiliar que proporciona los mismos metodos que el servidor.

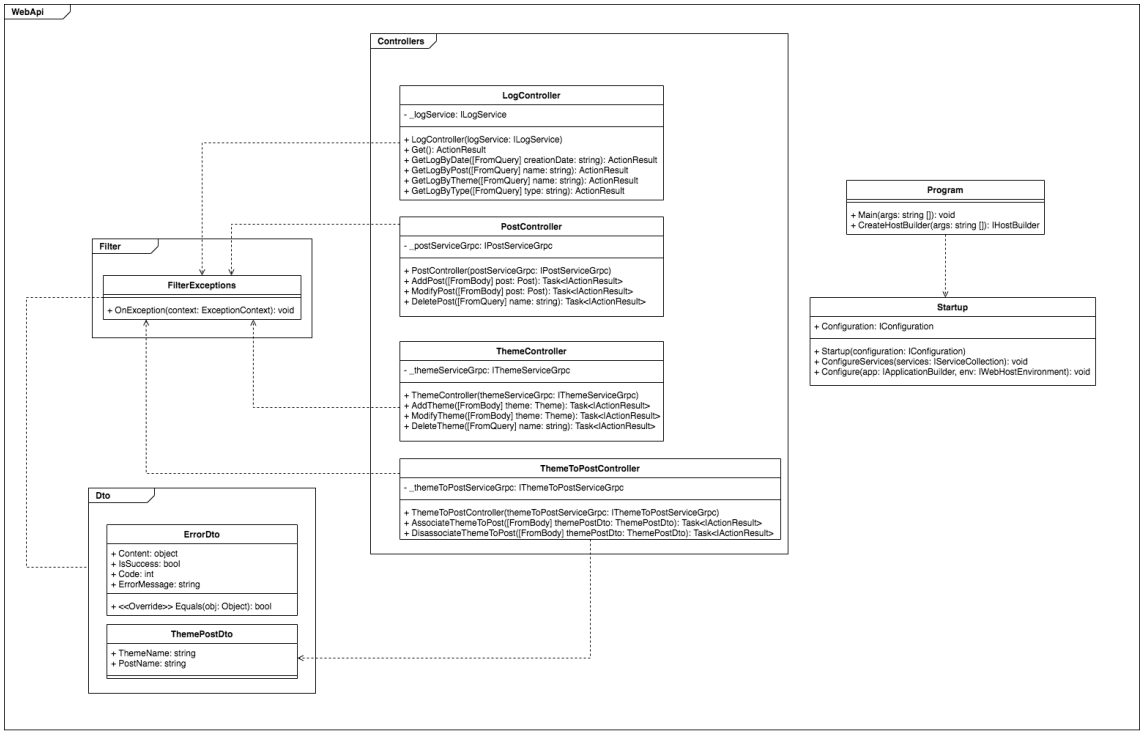
Este paquete también cuenta con un paquete de interfaces llamado GrpcServicesInterfaces que cuenta con dos interfaces, una correspondiente al servicio de Post y otra al de Theme, en las cuales nuevamente simplemente se definen los contratos de las funciones a implementar.

8. BusinessLogic:



En este paquete se define la lógica de las funciones del sistema. Se encarga de lanzar las distintas excepciones cuando algo no funciona correctamente. Valida que todos los datos que llegan sean correctos antes de realizar la persistencia. Contiene dos principales componentes, Managers y Services; el primero se encarga de manejar los distintos repositorios, mientras que el segundo contiene el RabbitHelper que implementa la parte de la recepción del mensaje.

9. WebApi:



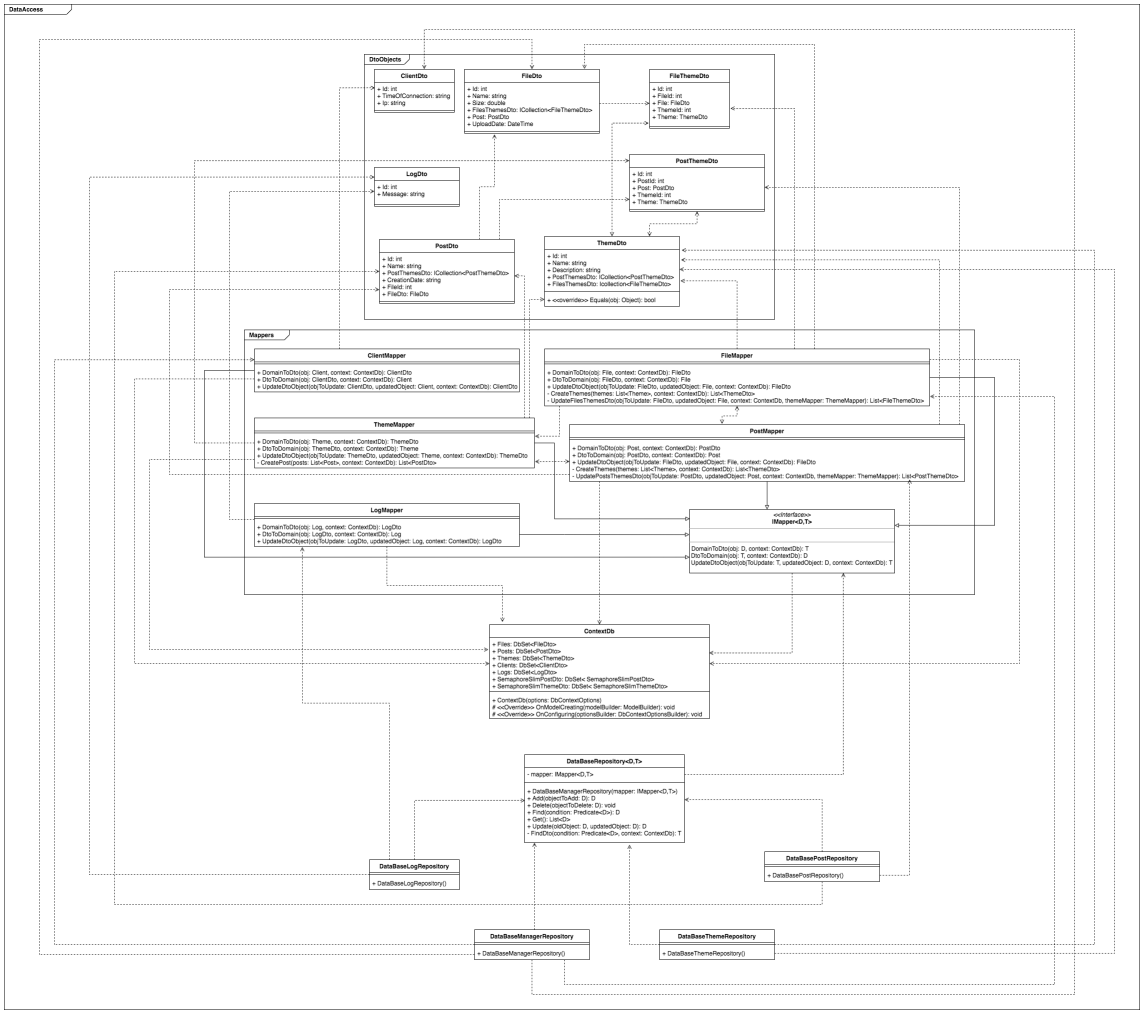
Este es el paquete ejecutable que expone endpoints del sistema al cliente. Se compone por los controladores, la clase program, la clase startup y la clase RabbitHelper. Los controladores son quienes se encargan de comunicarse con la lógica (los services) para poder realizar las diferentes funcionalidades; así mismo, capturan excepciones de todos los tipos, dependiendo de si se lleva a cabo lo deseado o no.

La clase RabbitHelper funciona como intermediario para poder llevar a cabo una comunicación eficiente entre . Esta comunicación es asincrónica y garantiza la recepción dado que recibe mensajes de confirmación. En este se definen colas que almacenan mensajes de quienes envían, hasta que quienes reciben obtengan y procesen dicho mensaje.

En la clase Program se inicializa dicho RabbitHelper con el ManagerLogRepository, para poder guardar en base de datos todos los mensajes y después si se inicia el programa.

Además para poder reducir el acoplamiento entre las capas, como dijimos la misma se realizó mediante el patrón de inyección de dependencias, haciendo que los componentes sean independientes unos de otros. Para configurar las inyección de dependencias se hace en la clase Startup de WebApi.

10. DataAccess:



En este paquete, como bien especifica su nombre, se maneja el acceso a datos en base de datos. Este cuenta con cinco principales componentes, el primero es la clase ContextDb la cual es la encargada de la interacción real entre nuestro sistema y la base de datos. Luego contenemos los distintos repositorios, uno por cada entidad que queremos persistir (Post, Theme, etc.); estos utilizan dos de los otros elementos para poder funcionar, DtoObjects y Mappers. DtoObjetcs define los atributos que tendrá cada una de las tablas en la base de datos, correspondientes a los atributos del objeto mismo. Las clases correspondientes a Mappers son las que efectivamente implementan las funciones que alteran la base de datos, ya sea para agregar, eliminar o actualizar datos. Por último, este paquete cuenta con las Migrations, este corresponde a código auto-generado por Entity Framework cuando se introduce un cambio en el modelo de datos, utilizamos esta herramienta para describir las actualizaciones necesarias a la base de datos.

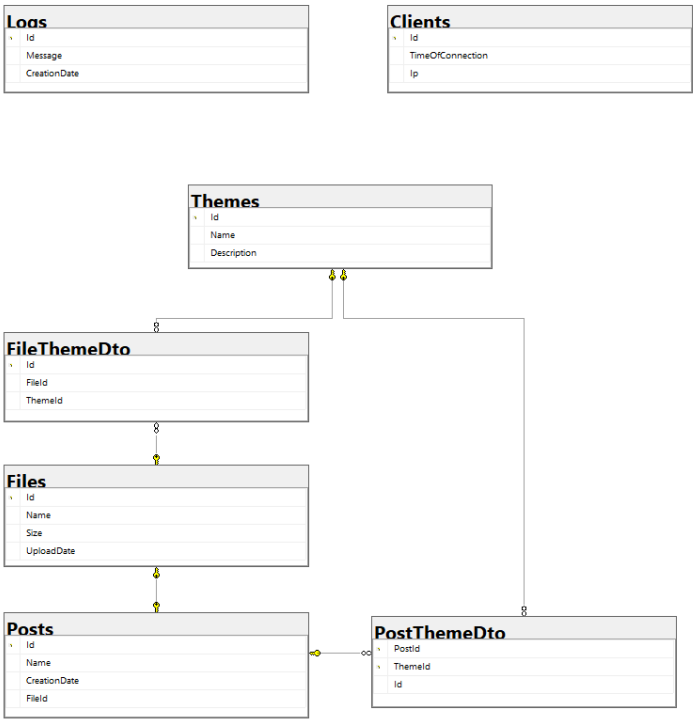
Nota: en caso de querer ver los UML con mejor calidad, se encuentran adjuntados en la carpeta del obligatorio dentro de UMLs.

5. Persistencia de datos

La persistencia de datos se realizo utilizando NET core, y como mencionamos anteriormente, para poder trabajar con la base de datos creamos objetos Dto, los cuales poseen la información que se deberá guardar en las tablas de la base de datos, tenemos una clase Dto por cada entidad de dominio, mas tres dos la relación n a n entre post con theme y file con theme.
Estos objetos se mapea, con los métodos definidos en las clases mappers, de Dto(Base de datos) a dominio y viceversa.

5.0.1. Modelado de la base de datos

Dentro de la base de datos se encuentran distintas tablas, correspondientes a los objetos del sistema, en el siguiente diagrama se muestra estas tablas y que columnas contienen, además se muestra la PK y FK de cada una.



6. Manejo de errores

6.1. WebAPI

Para el manejo de errores creamos nuestras excepciones las cuales fueron descriptas anteriormente en cada paquete.

Estas excepciones son lanzadas al encontrar un error, ya sea al settear una property de un objeto o usar alguna de las funciones del dataBaseRepository, y controladas en la lógica (services) con el uso de try catch tirando al momento de capturarlas la correspondiente excepción de grpc.
Luego estas excepciones son lanzadas a la API para ser controladas con la clase filterExceptions donde se capturan estos errores y se muestran, usando el objeto ErrorDto, el código de estado, si paso el error o no y el mensaje que corresponda para informar al usuario el error.

6.2. Console

Los errores de datos mal ingresados, o acciones que no se pueden realizar en determinado momento se encuentran validados del lado del servidor, en las clases services, estos indican el error con un mensaje que se muestra luego por consola en el lado del client.

Los errores de conexión perdida o de sockets se captura con try-catch en los lugares correspondientes y se lanzan mensajes que se muestran también por consola.
De la misma forma actúa la validación del path del archivo, donde en caso de ser una ruta incorrecta se lanza una excepción que es luego controlada en el filePageClient donde se informa al usuario que la ruta es incorrecta y se pide que se ingrese una nueva.

7. Documentación de mecanismos de comunicación de los componentes de su solución

7.1. Protocolo

Para que el Servidor y el Cliente puedan comunicarse de forma que el servidor reciba las requests y envíe las respuestas para que luego el cliente las interprete, fue necesario definir dos protocolos, uno para el envío de strings (Protocol) y otro para el envío de archivos (ProtocolFile).

Protocol

Para el envío y recibo de la información, se crearon dos métodos writePakgAsync y ReceivePakgAsync los cuales se encargan de enviar y recibir los paquetes con la información.

Estos paquetes están formados por un header, un command, un length y un data; el header nos indica si el mensaje se manda por una response o una request (RES/REQ), este tiene un largo fijo de3 bytes; el command es el número del requerimiento, que me indica cual funcionalidad debo llamar y tiene un largo fijo de 2 bytes, si el largo es de una cifra se rellena con ceros a la izquierda (ejemplo: 02).
El length me indica el largo de la data y se le suman cinco bytes(header+command) y tiene un largo fijo de 4, al igual que el command si la cifra del length es menor a 4 se rellena con ceros a la izquierda (ejemplo: 0012, indica que la data es de largo 12)
Por último, la data es un string que contiene toda la información que se esta enviando concatenada mediante #. Para completar el paquete.

Antes de mandar la información es necesario crear el Packet mientras que una vez que se recibe se debe hacer el proceso inverso para que se pueda interpretar la data del mismo. Este proceso inverso implica separar, según los #, la data del Packet, lo que nos permite acceder a la información en sí.
En el ReceivePakAsyn se leen los datos, con la función ReadAsync de networkStream, header, commmand y length almacenándolos en un array de largo 9 (3 para header, 2 para command y 4 para el length), que luego se convierte en string para

asignarlos al packet.command, packet.header y packet.length respectivamente.
Luego con ese length mas nueve (estos 9 son: 3 del header,2 del command y 4 del length) leemos la data, el mensaje que se envió tambien con la funcion ReadAsync de networkStream, almacenándola primero en un array de largo length y luego convirtiéndola en string para asignársela al packet.data.
En el SendPakgAsync se recibe un packet, se junta toda la información del packet en un string y se manda usando el WriteAsync de networkStream.

Nombre del campo	Header	Command	Length	Data
Valores	RES/REQ	1-12	0-9999	Data#data#...
Largo	3	2	4	Length

ProtocolFile

Para enviar y recibir archivos, se debe definir un FileNameLength, un FileSize y un MaxPacketSize; a partir de estos datos se define el paquete en el que se va a mandar. Este paquete se va a dividir en determinada cantidad de partes, dependiendo del MaxPacketSize ya que este es el tamaño máximo que puede tener.

Cuando el cliente envía el archivo utiliza el método WriteAsync de networkStream, invocado dentro del método SendFileAsync, escribe al mismo en partes.
Mientras que el servidor recibe el archivo llamando al método ReadAsync de networkStream invocado dentro del método ReceiveFileAsync, este también lee el archivo en partes
Dentro del envío y la recepción del archivo, también se incluye el envío de los datos del mismo (nombre, tamaño y Post al cual se va a asociar), como un paquete.

8. Justificación de las decisiones importantes de diseño e implementación.

Para diseñar nuestra aplicación, la primera decisión tomada fue que este iba a ser dividido en diferentes capas las cuales tienen distintas finalidades y distintos componentes.

Como mostramos en la sección 4, cada paquete cumple su función y corresponde a una de las capas del sistema, por un lado tenemos el punto de acceso al sistema que es la WebApi, esta se comunica con la lógica de negocio donde se encuentran las funciones respectivas a todas las funcionalidades de nuestra aplicación y luego esta se comunica con DataAcces que es quien maneja acceso a la base de datos.

Para favorecer a la mantenibilidad de nuestro sistema decidimos utilizar inyecciones de dependencia mediante las cuales se puede lograr que las clases de alto nivel no dependan de las de más bajo nivel. Lo utilizamos por ejemplo entre cada repositorio con su respectivo Manager, logrando que por ejemplo la WebApi no dependa de la implementación de la lógica de negocio ni del acceso a datos. Pensando a futuros cambios en el sistema, notamos que en caso de querer modificar el mecanismo de persistencia de datos actual, seria sencillo llevarlo a cabo dado que no dependen uno de otro, cada bloque es cambiable sin alterar al resto, se puede aislar del resto del sistema.

Para favorecer la independencia entre capas además utilizamos interfaces las cuales definen contratos, no la implementación en si; las inyecciones de dependencia son las que me brindan el beneficio de poder alterar el comportamiento con facilidad. Además, estas interfaces son pequeñas y específicas según su función, por esta razón las clases que luego implementaran lo descrito en estas solo quedan acopladas al contrato de las funciones que efectivamente usarán luego.

Otra decisión que tomamos con respecto a las clases fue crear ciertas clases auxiliares que brindan ayuda a las clases principales, denominadas Helpers. Esto lo podemos ver por ejemplo dentro de los Services de BusinessLogic, la clase PostService contiene una clase PostHelper la cual brinda funciones que el Service necesita para cumplir su propósito pero para reducir la extensión y la complejidad de la clase

decidimos dividirla.

9. Modificaciones realizadas al código

1- Como mencionamos anteriormente uno de los cambios que se hizo en el proyecto fue persistir todas las entidades, para lo que se agrego el paquete DataAccess con las entidades que se van a persistir en la base de datos y los mappers.Para esto también se agrego el repositorio genérico y los managers para manejar estos.

2- El proyecto server paso de ser una aplicación de consola sin tipo a ser una aplicación de consola GRPC, dentro de este se dejo la estructura que ya teníamos de server en un directorio server, y se agrego otro directorio serverGrpc donde se encuentra la estructura de este.

Se agrego la librería de clases AdministrativeServer que contiene los protos y services de grpc, y otra con las interfaces de estos services llamada GRPCServicesInterfaces.

3- Se agrego el proyecto de la WebAPI con los controllers, el filtro de excepciones y los dtos.

4- Se extrajo de los services las interfaces para poder hacer luego la inyección de dependencia.

5- Se agrego una nueva aplicación de consola para el server de logs, llamada logsServer y una librería de clases con las interfaces de los logs.

6- se hizo refactor sobre los services de BusinessLogic, extrayendo aquellos métodos que eran de control de datos o de envió de listas a clases helper para facilitar la lectura y el entendimiento y reducir la cantidad de lineas por clase.

7-Se cambiaron las estructuras creadas de semáforos por locks, en los casos de modificar o eliminar post y theme.

10. Manual de usuario

Luego de abierta la solución, se debe configurar en el App.config la IP correspondiente a la computadora del usuario.
Luego se debe configurar correctamente el connectionstring en el appseting.json para conectar a la bdd.

Después de esta acción se debe ejecutar la aplicación de consola Server, luego la de LogServer y luego la aplicación de consola Client y la WebAPI.

Una vez ejecutadas las dos aplicaciones

10.1. Aplicación cliente-servidor

Si el usuario usa la consola verá dos menús, en los cuales deberá seleccionar la opción que desee.

Para seleccionar la opción el usuario debe desplazarse por el menú con las teclas down y up del teclado y luego al llegar a la opción que desee deberá presionar enter. Según la opción que seleccione se le mostrará la página que corresponda.

En los casos que se muestren las opciones con índices, se debe ingresar el número indicado en cada opción para poder acceder a la funcionalidad.

10.2. Web API

Si el usuario desea usar la Web API puede abrir Postman y teniendo en cuenta los endpoins descriptos en la sección de web api podrá realizar las acciones que desee.