Universidad ORT Uruguay

Obligatorio 3

Programación de Redes

Agustina Disiot 221025

Iván Monjardin 239850

https://github.com/IvanMonjardin/ProgramacionRedesOB1

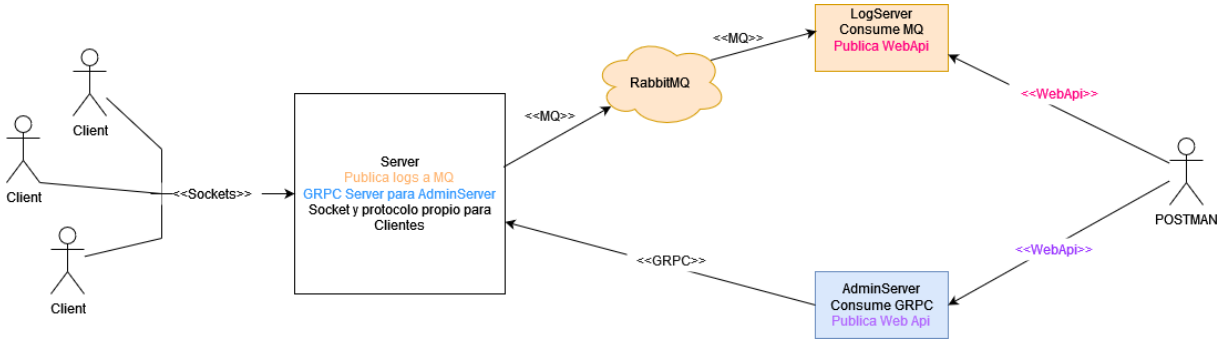
2021

Índice:

Introducción

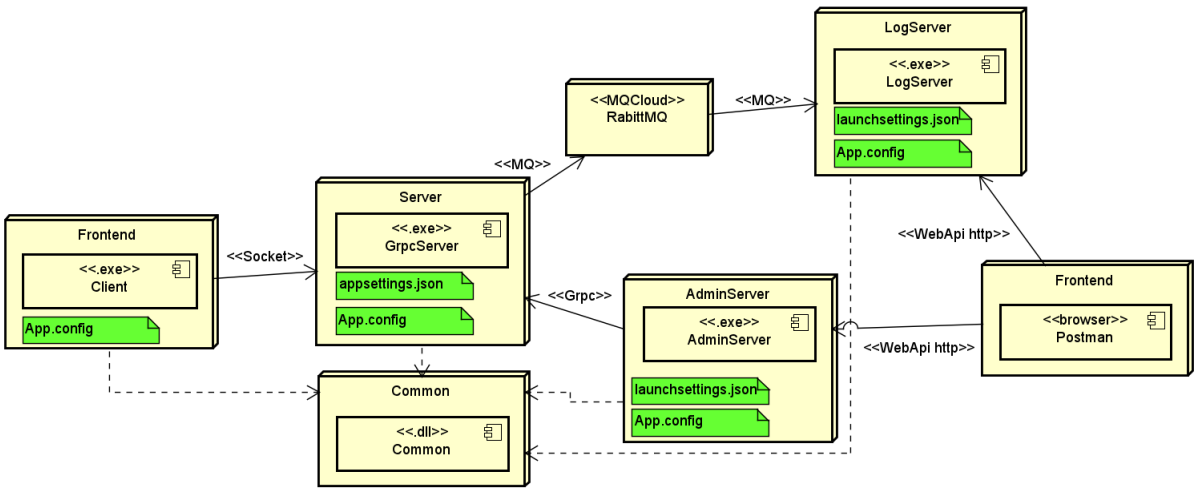
Para esta entrega se pidió que se agregue una interfaz de monitoreo (logs) y de gestión administrativa (AdminServer).
Además se pidió que utilizáramos las tecnologías MOM, GRPC y Web Api.

Tomando en cuenta estos requerimientos se diseñó la siguiente solución:



A la izquierda vemos los servicios ya existentes que permiten la conexión entre el cliente y el servidor utilizando sockets y nuestro propio protocolo de comunicación.
En naranja se puede ver como el servidor principal pública los logs de los eventos hacia una cola de mensajes. Dicha MQ es utilizada por un servidor de Logs el cual recibe los logs enviados y los publica utilizando una WebApi.
En azul podemos ver que el servidor principal contiene un servicio de GRPC el cual es utilizado por el servidor administrativo para hacer pedidos. Dichos pedidos pueden ser iniciados utilizando una Web Api publicada por el AdminServer.

En este diagrama de despliegue se ve con un poco más de detalle la solución encontrada.



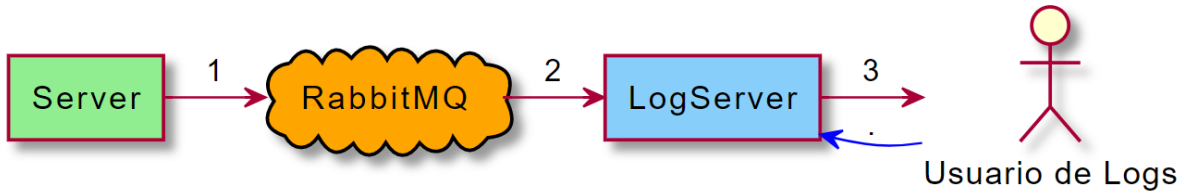
En este caso se decidió utilizar un solo paquete Common con las funcionalidades compartidas, también se podría tener distintos Common dependiendo de qué parte se comparte.

Todos los diagramas de este documento se encuentran en la carpeta Documentacion/Diagramas

A continuación vamos a explicar más en profundidad cada una de las partes y la motivación de la elección de cada una para cumplir sus respectivos requerimientos.

Sistema de Logs

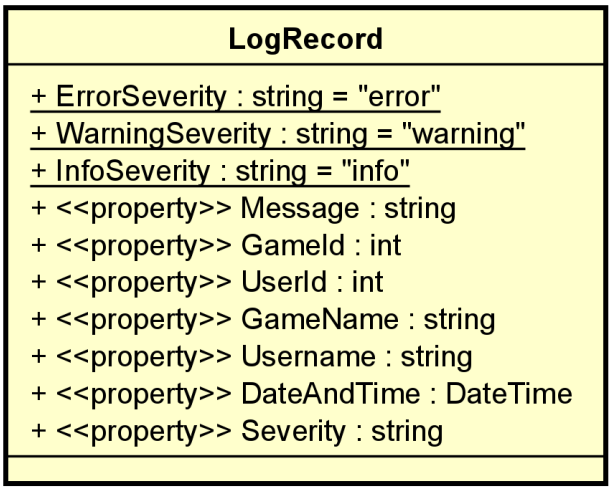
Para el requerimientos de logs se optó por utilizar la tecnología de MQ de RabbitMQ y una Web API Rest.
La MQ se utiliza para comunicar el servidor principal (donde se generan los logs) y el Servidor de logs (el consumidor de logs). En el medio está el exchange de RabbitMQ que se encarga de recibir y mandar los logs cuando sea necesario.
Una arquitectura a alto nivel sería la siguiente:



Las flechas en rojo muestran el camino que siguen los logs: Se generan en el servidor, se mandan a la MQ, de la MQ pasan el LogServer y el LogServer lo manda el usuario (Postman).
En azul se ve el pedido de los logs mediante una WebApi al LogServer.

Logs

En la clase de LogRecord se pueden observar la información que decidimos guardar.



Se encuentran los datos que se pedían: Usuario, juego, fecha.
Para la fecha:

- Se utilizó DateTime en vez de Date, ya que es más fácil probar el filtrado. Se podría cambiar fácilmente si se quiere.
- Se utiliza UTC y la apreciación es de 1 segundo.
- Se agrega cuando se manda el log, no se tiene que especificar cada vez

Severity:
Se definieron 3 nivel de logs:

- Info: cualquier evento que modifique la “base de datos” y eventos como login, logout, inicialización del servidor etc.
- Warning: Errores esperados por el sistema. Ejemplo, se intenta crear un juego con título repetido, admin intenta acceder a usuario con id no existente y otros.
- Error: Errores no esperados por el sistema y que no deberían ocurrir. Para esto se utiliza una funcionalidad de prueba implementada en el obl. 1 que hace que el cliente envié un mensaje “corrupto” al servidor.

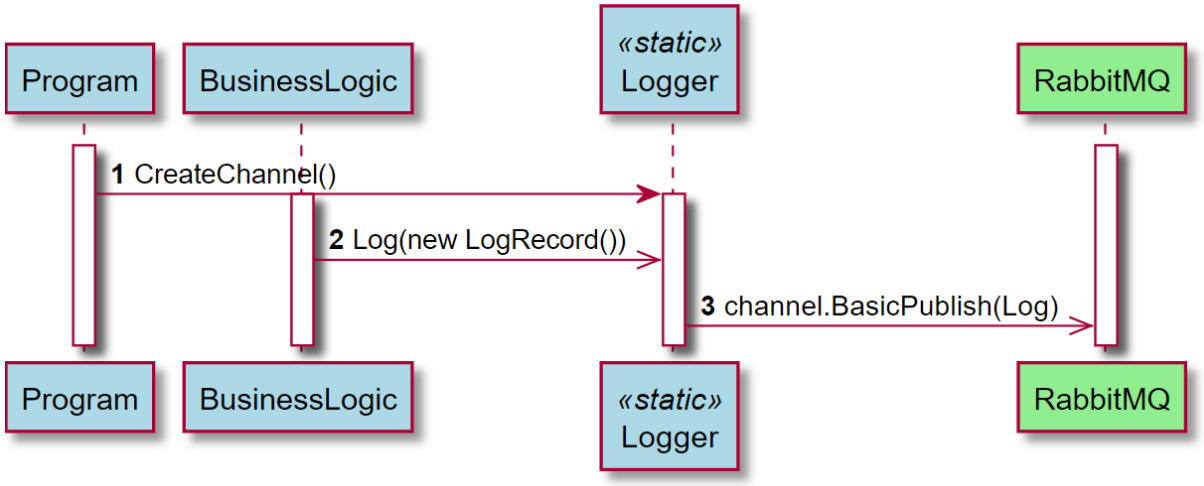
Más adelante se explicará otros usos que le dimos a la severity.

Otras notas:

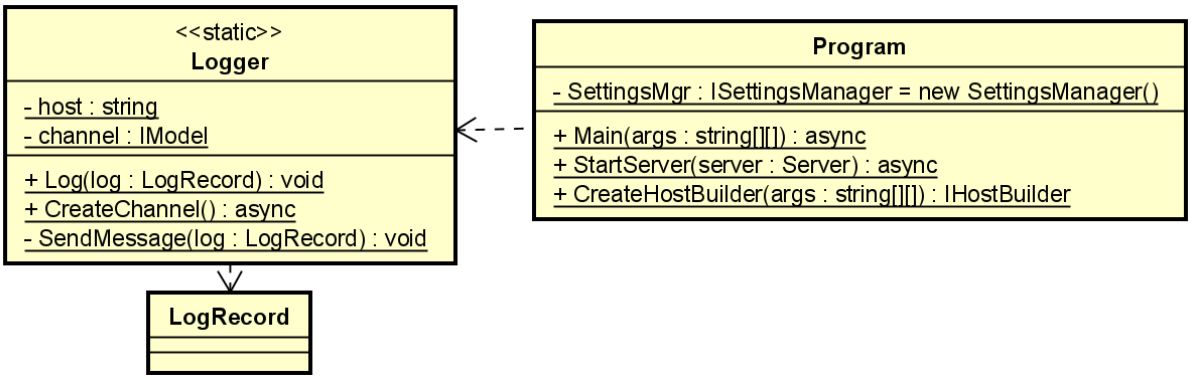
- En caso que no aplique, las propiedades se dejan por defecto. Ej. si se crea un usuario, GameName = null.
- LogRecord se encuentra en Common ya que es utilizado tanto por Server como por LogServer

Productor de Logs

El servidor principal es el encargado de producir los logs y publicarlos en la cola de mensajes. En el siguiente diagrama se muestra como el servidor principal (azul) inicializa el sistema interno de logueo, y como cuando se genera un log se manda a la RabbitMQ.



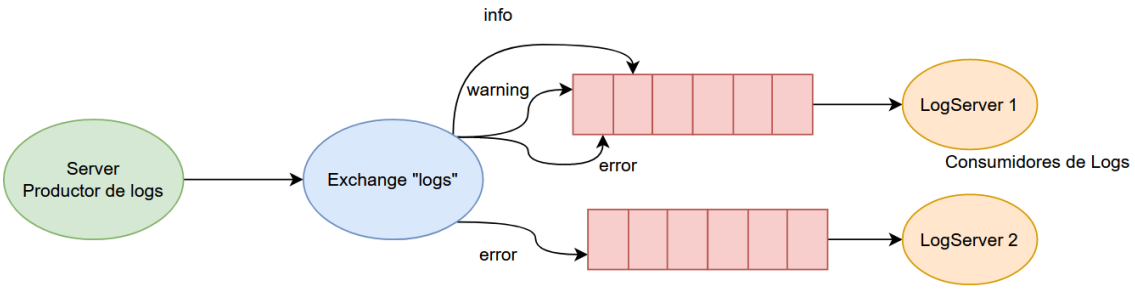
Logger es una clase estática que se instancia al iniciar la aplicación (así la conexión al MQ ya está hecha) y es la encargada de enviar los logs.



Program es la clase que contiene al Main del servidor principal. Además de inicializar el Logger también genera un log de cuando se crea prende el servidor.
El primer log siempre demora más, por lo tanto así la comunicación con el primer cliente es más fluida ya que ya se mandó un log antes (el de prender el servidor).

RabbitMQ

Para la cola de mensajes se utilizó un servicio de RabbitMQ en la nube. Cuando se crea la aplicación se define un exchange "logs" donde se publican los mensajes.
Los mensajes se publican utilizando como routing key la severity del log (error, warning, info)
Se tomó esta decisión para permitir que los consumidores de los logs elijan a qué tipo de logs subscribirse.



En el diagrama de arriba podemos ver la arquitectura del exchange. Cada cliente (LogServer) puede definir su propia cola especificando que tipo de logs quiere.
Para configurar la severity de los logs que se quien exponer mediante la Web Api, hay que utilizar el App.config del LogServer

```
<appSettings>
    <add key="MQUri" value="amqps://yqkizmwe:PLX7QHzf3_iKjXhKCgBcn_Lb3s6gbRKO@clam.rmq.cloudamqp.com/yqkizmwe"/>
    <add key="severities" value="error,warning,info"/>
</appSettings>
```

Se ponen las severities separadas por comas

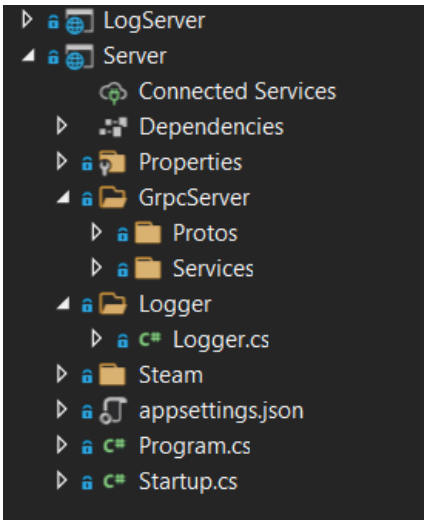
El utilizar un servicio de MOM en la nube llevó a:

1. Tomar la decisión de hacer la funcionalidad de logear asincrónica para que el resto del código no espere por un servicio que no va a devolver nada.

2. Poder probar el servicio estando en dos computadoras distintas. Cualquiera que corra el servidor puede generar logs el cual el otro va a ver. Siempre dependiendo de la configuración de “severities” que tenga. Se incluye una demostración de esto en el Anexo Demo MQCloud y Logs

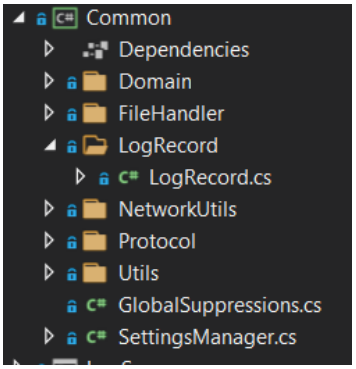
Modificaciones al Servidor Original

Agregarle el sistema de logs llevó a realizar las siguientes modificaciones al servidor original:



Se agregó la clase Logger.cs

También hay cambios en common:



Se agrega un LogRecord con la información de los Logs.

```
public class LogRecord
{
    public const string ErrorSeverity = "error";
    public const string WarningSeverity = "warning";
    public const string InfoSeverity = "info";

    11 references | IvanMonjardin, 19 hours ago | 1 author, 1 change
    public string Message { get; set; }
    8 references | IvanMonjardin, 19 hours ago | 1 author, 1 change
    public int GameId { get; set; }
    9 references | IvanMonjardin, 19 hours ago | 1 author, 1 change
    public int UserId { get; set; }
    7 references | IvanMonjardin, 19 hours ago | 1 author, 1 change
    public string GameName { get; set; }
    8 references | IvanMonjardin, 19 hours ago | 1 author, 1 change
    public string Username { get; set; }
    3 references | IvanMonjardin, 19 hours ago | 1 author, 1 change
    public DateTime DateAndTime { get; set; }
    13 references | IvanMonjardin, 19 hours ago | 1 author, 1 change
    public string Severity { get; set; }
}
```

Cambios a la lógica de negocio

Cambios en BusinessLogicGameCRUD:

Se agregaron dos funciones privadas para loguear los logs de información y warnings. Estos se utilizan en las funciones que ya existían anteriormente.

```
3 references | IvanMonjardin, 35 minutes ago | 1 author, 1 change
private void LogGameInfo(Game game, string message)...

2 references | IvanMonjardin, 35 minutes ago | 1 author, 1 change
private void LogGameWarning(Game game, string message)...
```

Por ejemplo cuando se publica un juego (PublishGame):

```
public string PublishGame(Game newGame)
{
    try
    {
        VerifyGame(newGame);
    }
    catch (Exception e) when (e is ServerError || e is TitleAlreadyExistsException)
    {
        newGame.Id = -1;
        LogGameWarning(newGame, $"Se intento crear un nuevo juego '{newGame.Title}' pero ocurrió el error: {e.Message}");
        throw;
    }
    List<Game> games = da.Games;
    lock (games)
    {
        newGame.Id = da.NextGameID;
        newGame.ReviewsRating = 0;
        newGame.Reviews = new List<Review>();
        games.Add(newGame);
        string msg = $"Se publicó el juego {newGame.Title} correctamente";
        LogGameInfo(newGame, msg);
        return msg;
    }
}
```

Se intenta verificar el juego, si no lo logra se hace el Log del warning con el mensaje y se hace el throw de la excepción. Si lo logra entonces se hace log de info con el mensaje de success.

Se utiliza a veces también el constructor de Logger, que es una clase estática, indicando la severity por parámetro.

```
string msg = " No existe el juego, tal vez haya sido eliminado";
Logger.Log(new LogRecord { GameId = gameId, Message = msg, Severity = LogRecord.WarningSeverity });
```

En BusinessLogicGameInfo agregamos tres funciones privadas para registrar los logs.

```
2 references | IvanMonjardin, 17 hours ago | 1 author, 1 change
private void LogGamePurchase(Game game, User buyer)...

1 reference | IvanMonjardin, 17 hours ago | 1 author, 1 change
private void LogGameAssociationError(Game game, User buyer)...

1 reference | IvanMonjardin, 17 hours ago | 1 author, 1 change
private void LogGameErrorBuyGameTwice(Game game, User buyer)...
```

El LogGamePurchase simplemente registra la compra para el log de info. El AssociationError hace referencia a cuando se quiere adquirir un juego que ya está asociado al usuario y este ocurre cuando se intenta hacer la asociación por la Api y es de severity warning. El ErrorBuyGameTwice hace el log cuando es el mismo tipo de error pero dado desde el servidor, es decir que podría ocurrir cuando se hace la compra desde la consola. Ya que el cliente no puede mandar parámetros incorrectos desde la consola, es muy difícil reproducir este error.

En BusinessLogicSessions se agregó una función privada para los logs de info de creación de usuario, login y logout.

```
3 references | IvanMonjardin, 17 hours ago | 1 author, 1 change
private void LogUser(User userToLog, string msg)...
```

Ejemplo en Login:

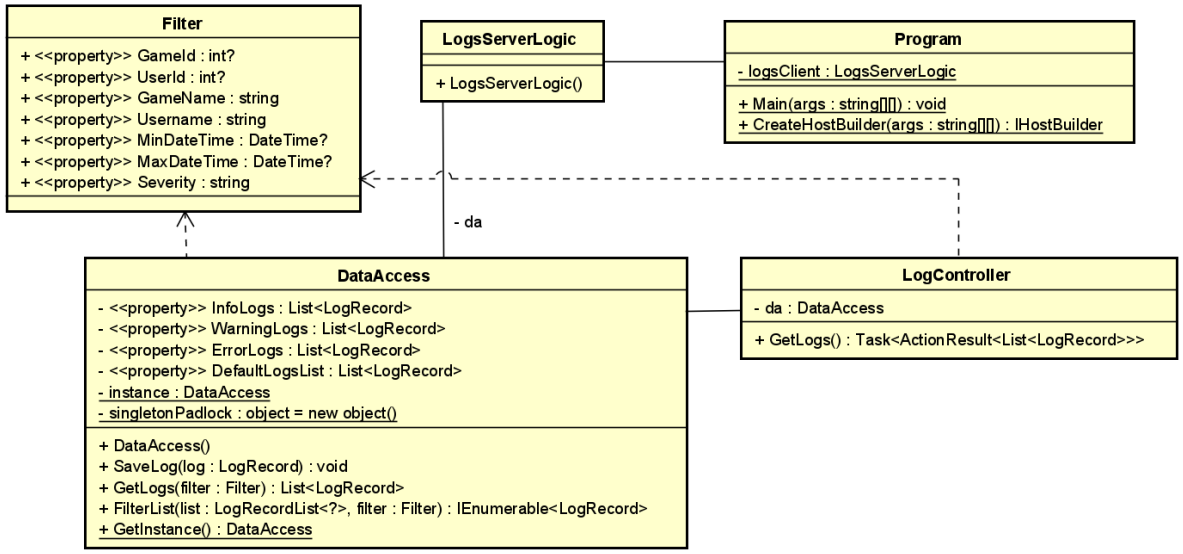
```
public bool Login(string newUserName, INetworkStreamHandler nwsh)
{
    List<User> users = da.Users;
    lock (users)
    {
        bool alreadyExists = users.Exists(u => u.Name == newUserName);

        if (!alreadyExists)
        {
            User newUser = new User()
            {
                Name = newUserName,
                Id = da.NextUserID
            };
            da.Users.Add(newUser);
            LogUser(newUser, "Usuario creado");
        }
        da.Connections.Add(nwsh, newUserName);
        User loggedInUser = users.Find(u => u.Name == newUserName);
        LogUser(loggedInUser, "Usuario inició sesióñ");

        return !alreadyExists;
    }
}
```

Se puede dar un warning por intentar crear un usuario con nombre repetido. Ahí mandamos el log de warning por constructor en CreateUser:

```
if (!alreadyExists)
{
    da.Users.Add(newUser);
    LogUser(newUser, "Usuario creado correctamente");
}
else
{
    Logger.Log(new LogRecord
    {
        Message = "Ya existe el usuario",
        UserId = newUser.Id,
        Username = newUser.Name,
        Severity = LogRecord.WarningSeverity
    });
}
```

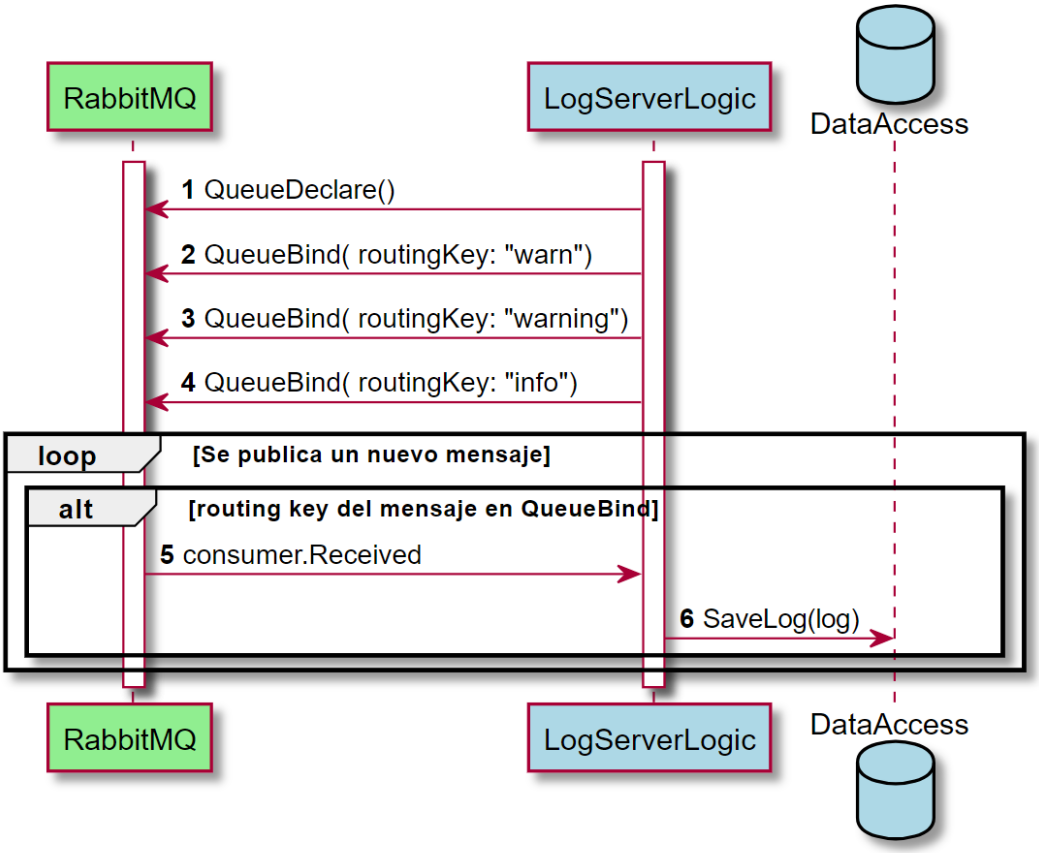
Consumidor de Logs

Se creó un nuevo proyecto el cual se encarga de consumir los logs publicados en cola de mensajes, guardarlos en una "base de datos" en memoria, y exponerlos mediante una web api. En el siguiente diagrama se puede ver las clases del LogServer

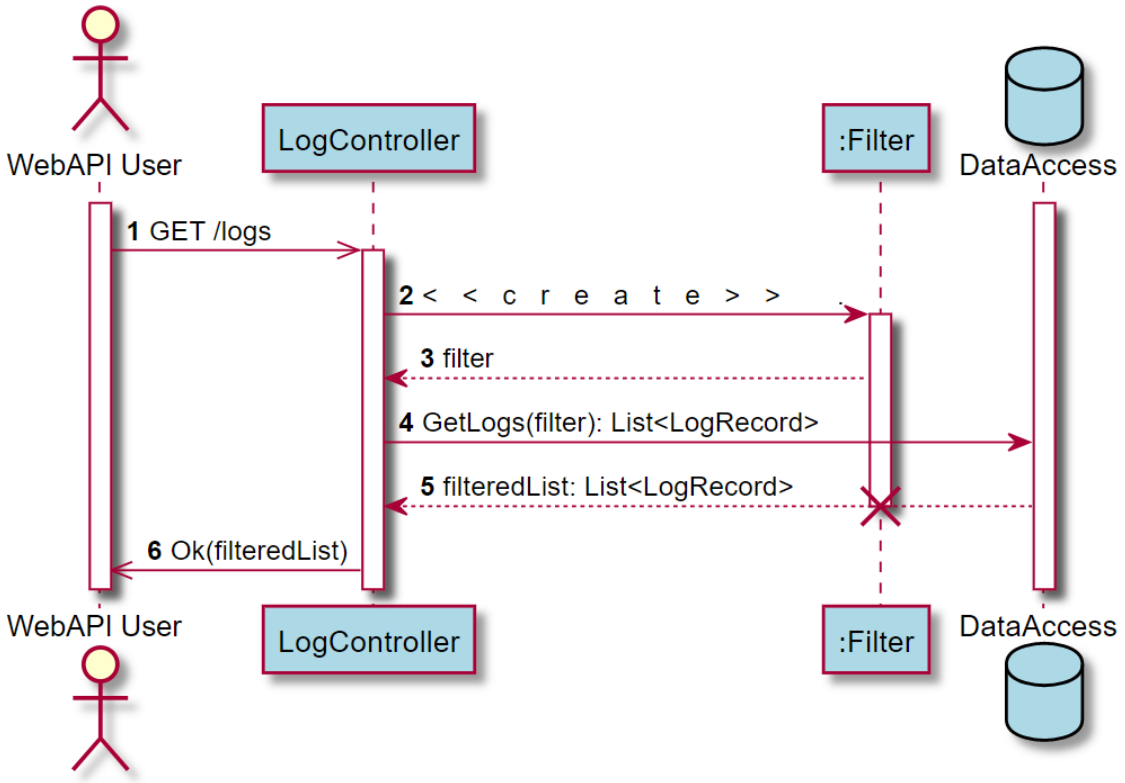


LogsServerLogic: Se encarga de conseguir los logs y guardarlos en DataAccess.
Filter: Clase con lista de los filtros pedidos por el cliente. Las propiedades son nullables para indicar que no se quiere filtrar por esa propiedad.
LogController: Se encarga de exponer la Web Api, recibir los pedidos y pedirselos al DataAccess. Además de convertir los queryParams de la WebApi request a un objeto Filter.
DataAccess: Se encarga de guardar los logs en tres listas (una por tipo de severidad). Además se encarga de devolver los bugs filtrados cuando se piden.

En el siguiente diagrama se muestra como el LogServerLogic declarara los colas en el servicio de RabbitMQ, recibe los logs del mismo y los guarda.



En el siguiente diagrama se ve cómo el usuario(Ej. por Postman) pide los logs, el controller se los pide a la web api y se los devuelve al usuario.



Busqueda de logs:

Al haber tres listas de logs, si se especifica la severidad, solo se busca por la correspondiente.
Se utilizan locks de las listas pero solo cuando se están usando para buscar/ escribir los logs.
Ej. si se están buscando logs en la lista “InfoLogs”, se pueden agregar/buscar los a la lista “ErrorLogs”. Esto invita a que solo se pueda realizar una operación de búsqueda y/o agregar logs a la vez. Al mismo tiempo, esto evita que se estén buscando logs y se agreguen a la misma lista, pudiendo generar resultados incoherentes.

```
public List<LogRecord> GetLogs(Filter filter)
{
    IEnumerable<LogRecord> result = new List<LogRecord>();

    if (filter.Severity == null || filter.Severity == LogRecord.InfoSeverity)
        result = result.Concat(FilterList(InfoLogs, filter));
    if (filter.Severity == null || filter.Severity == LogRecord.WarningSeverity)
        result = result.Concat(FilterList(WarningLogs, filter));
    if (filter.Severity == null || filter.Severity == LogRecord.ErrorSeverity)
        result = result.Concat(FilterList(ErrorLogs, filter));

    return result.ToList();
}

3 references | IvanMonjardin, 20 hours ago | 1 author, 2 changes
public IEnumerable<LogRecord> FilterList(List<LogRecord> list, Filter filter)
{
    lock (list)
    {
        return list.Where(l => filter.GameId == null || l.GameId == filter.GameId)
                   .Where(l => filter.GameName == null || l.GameName.Contains(filter.GameName))
                   .Where(l => filter.UserId == null || l.UserId == filter.UserId)
                   .Where(l => filter.Username == null || l.Username.Contains(filter.Username))
                   .Where(l => filter.MinDateTime == null || l.DateAndTime >= filter.MinDateTime)
                   .Where(l => filter.MaxDateTime == null || l.DateAndTime <= filter.MaxDateTime);
    }
}
```

No se buscó una solución extensible en el sentido de los niveles de severity. En este caso se priorizó la simplicidad del código. Se supuso que la severidad es algo que no va a cambiar mucho en el futuro ni que se va a llegar a una cantidad muy grande.
De todos modos, si se fuesen a agregar más tipos de severidad esta es de las pocas clases que sufrirán cambios.

Motivación MOM para Logs

Se decidió utilizar MOM para la comunicación entre logs por diversas razones. La principal es el método de comunicación.
No es necesario que el servidor de Logs le pida al Servidor principal los logs. El servidor principal va publicando los logs a la cola y el de Logs los va consumiendo.

No se sabe cuando se va a publicar un nuevo log, y al usar MOM no es necesario que el servidor de logs esté preguntando constantemente si se creó un nuevo log o no. Si usariamos por ejemplo GRPC sería más complicado tener esta funcionalidad, lo mismo con WebApi, ya que estas dos tecnologias están más orientadas a mandar la request y recibir la respuesta

correspondiente. Si todavía no se creó un log cuando mandar la request, tendría que esperar hasta que se publique uno.
No solo el que pide los logs no tiene que esperar una respuesta, pero el servidor tampoco. Si usariamos una web api, el servidor podria mandar un POST cada vez que se genera un log y si bien no es una mala solución, no dependes del LogServer para que te avise que llego bien, sino dependes de la MQ que en este caso fue hosteado por un tercero.

La segunda ventaja es permite tener muchos servidores de Logs conectados a la misma MQ y que cada uno consuma los tipos de logs dependiendo de la severidad que quieran.
Además, está la ventaja de donde se guardan los logs. Al mandar todos los logs directamente al MQ no es necesario que el servidor principal gaste memoria/espacio en la db para guardarlos. El que es responsable de guardarlos es el servidor que los quiera usar (LogServer). Si fuera con GRPC o WebApi, el servidor tendría que guardar los logs hasta que se los pidan.

Web API

Se utilizó una Web API Rest para que el cliente del servidor de logs puede acceder a lo mismos.

Decisiones de verbos HTTP

Para diseñar la api se diseñaron endpoint que cumplan con las recomendaciones de nombre de Rest Api, por ejemplo lo que estan detallado aca: https://restfulapi.net/resource-naming/
Algunas decisiones que se tomaron:

- Se definió un recurso general logs en plural al principio del endpoint, ya que solo hicimos un controlador para el servidor de logs con un único endpoint
- Se utilizaron los métodos HTTP para indicar acciones sobre los recursos y no se incluyeron verbos en la URI. Por ejemplo:
  - GET /logs
- Todos los endpoints son en minúscula y no se incluye “/” al final
- Se utilizan parámetros en los query/params de la request para seleccionar el filtro.

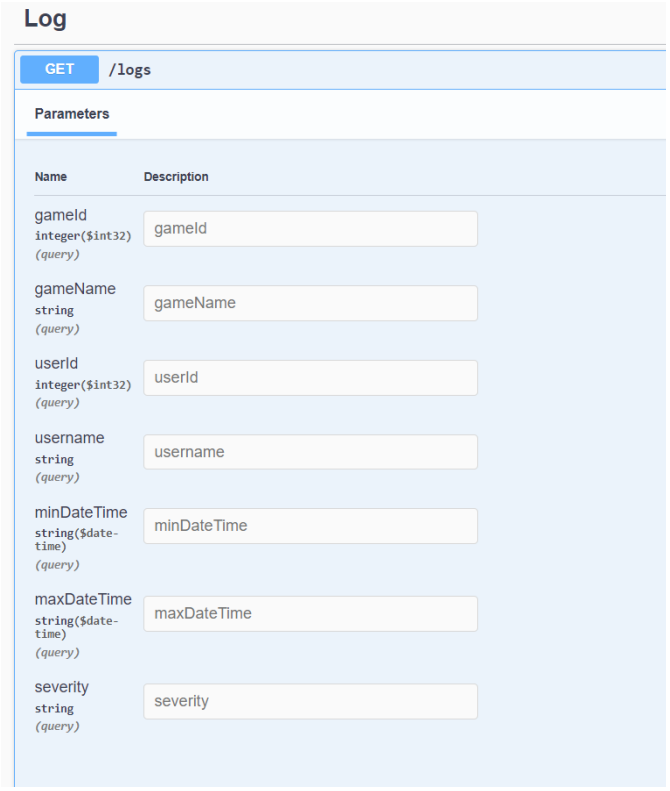
URL base: http://localhost:5002/

Códigos de error

Siempre se devuelve 200 OK. No hay errores ya que el default está definido para que devuelva todos los logs. En caso de no haber logs se verá la lista vacía.

Información general

Foto de la request en swagger:

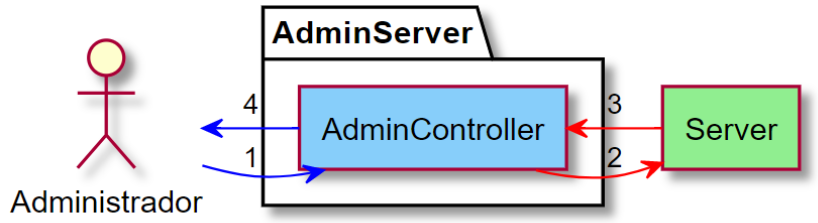


Notas:

- Los filtros de obtener los logs se da por query/params.
- Por default se obtienen todos los logs.
- Para los DateTime se usa el formato por defecto de swagger ,el ISO 8601 hasta los segundos. Ejemplo: 2017-07-21T17:32:28Z. Si se hace un pedido de los logs ya se devuelve la fecha en el formato especificado.
- Se utiliza un parser de JSON para poder utilizar la MQ de Rabbit. Tanto para serializar como desezializar los LogRecord.

Servidor Administrativo

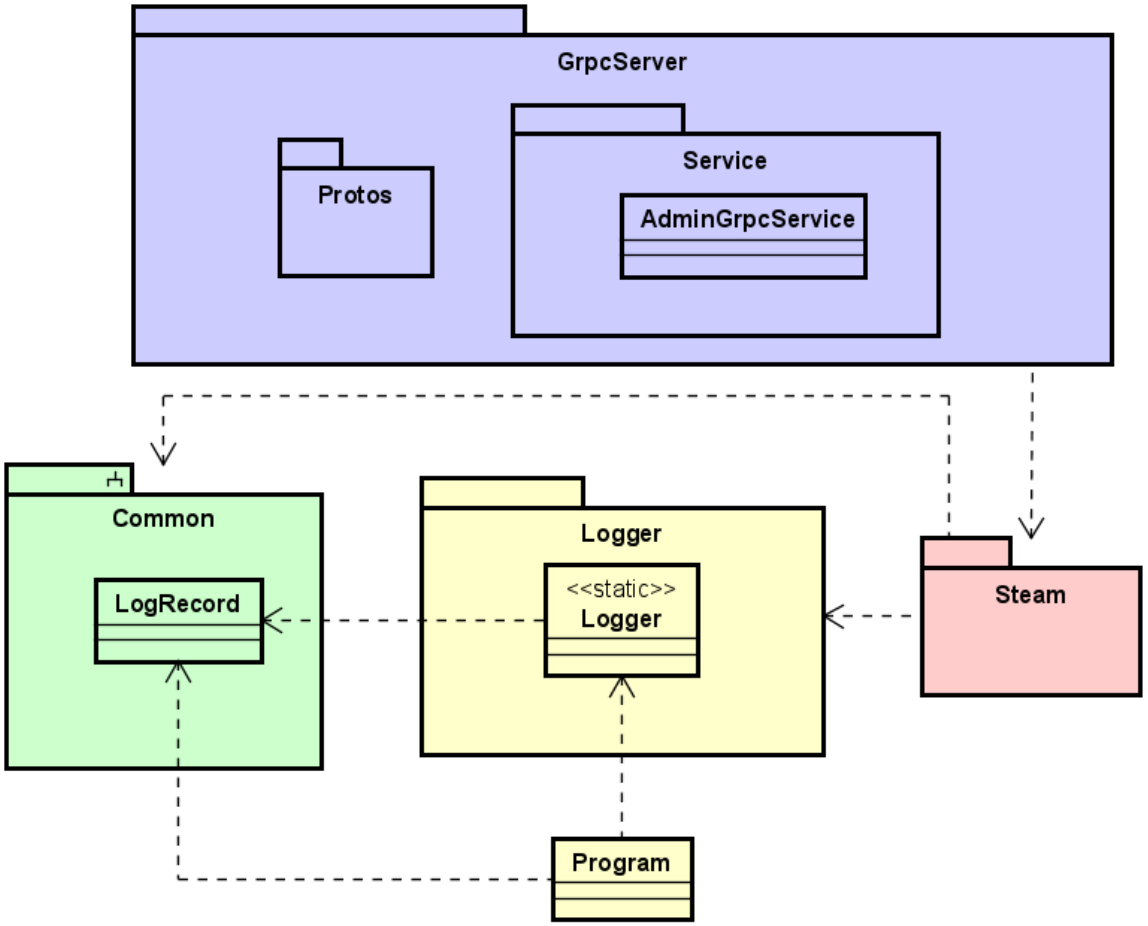
Para el requerimiento de Servidor administrativo se decidió utilizar la tecnología de GRPC y WebApi .
GRPC para comunicar el nuevo servidor administrativo (AdminServer) y el servidor original (Server). La Web Api se publica para permitir que los administradores accedan a las acciones.
A continuación un diagrama con la comunicación básica entre las distintas partes:



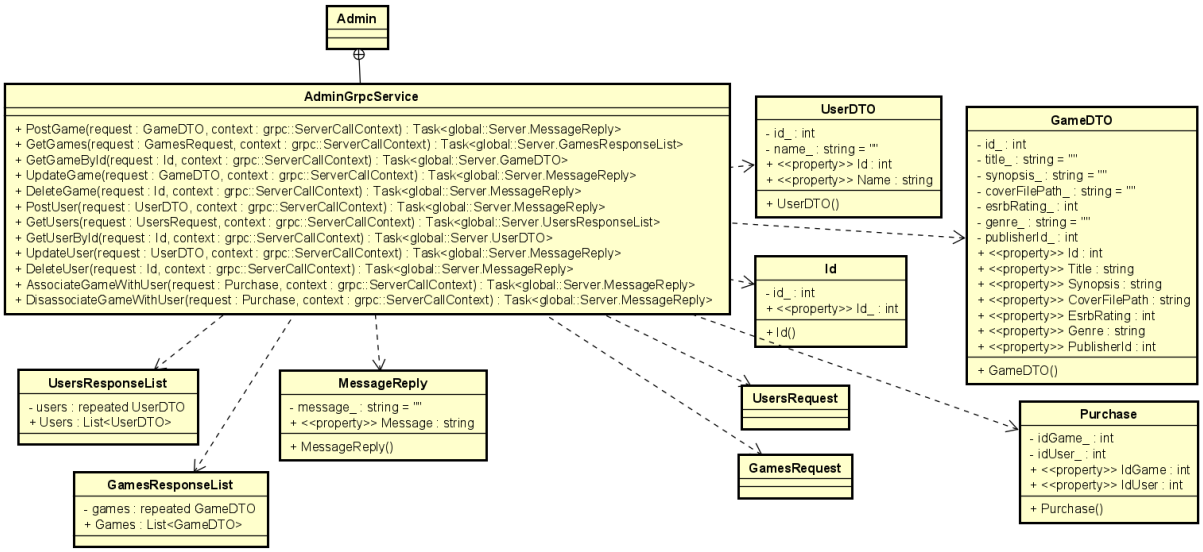
En rojo las comunicaciones mediante GRPC
En azul las comunicaciones mediante Http/WebApi
Al utilizar una web api y grpc, los pedidos al servidor se realizan una vez que se hacen los pedidos a la web api. El AdminServer no guarda la información como si lo hace el LogServer. Esto es porque no son solo pedidos de información que no se modifica sino que son pedidos de información dinámicas y acciones que se quieren realizar al Server principal.

Servidor GRPC

En el siguiente diagrama del paquete Server, se puede ver en violeta las nuevas clases que se agregaron para permitir que se pueda utilizar como servidor de GRPC. Además estas usan Steam (la lógica original) para realizar las acciones sobre dicho servidor.



En el AdminGrpcService es donde se definen las funcionalidades que se puede utilizar mediante Grpc. A continuación un diagrama más detallado del mismo:



.proto:
Aquí definimos todos los servicios del Servidor Administrativo

```
service Admin {
  rpc PostGame (GameDTO) returns (MessageReply);
  rpc GetGames (GamesRequest) returns (GamesResponseList);
  rpc GetGameById(Id) returns (GameDTO);
  rpc UpdateGame(GameDTO) returns (MessageReply);
  rpc DeleteGame(Id) returns (MessageReply);
  rpc PostUser(UserDTO) returns (MessageReply);
  rpc GetUsers (UsersRequest) returns (UsersResponseList);
  rpc GetUserById(Id) returns (UserDTO);
  rpc UpdateUser(UserDTO) returns (MessageReply);
  rpc DeleteUser(Id) returns (MessageReply);
  rpc AssociateGameWithUser(Purchase) returns (MessageReply);
  rpc DisassociateGameWithUser(Purchase) returns (MessageReply);
}
```

Si bien sólo se pedía alta, baja y modificación de usuarios y juegos, decidimos implementar el CRUD completo.

Data Transfer Objects utilizados para hacer la request y reply de los servicios:
El GameDTO contiene casi todos los atributos del Game de Domino. No contiene la lista de Reviews ni ReviewRating ya que estos comienzan vacío y en 0 respectivamente cuando se publica un juego por primera vez, y en el caso de modificación estos no se modifican, por lo tanto no es necesario mandarlos en la request. En cuanto al ESRBRating, en el Dominio es un Enum y en el DTO simplemente se manda el número y este se castea. En cuanto al Publisher, en el Domino es de tipo User, en el DTO decidimos usar el Id del User y luego conseguir el usuario.
En cuanto al Post de juego este se manda sin un Id y este se autogenera. En el Update se manda con el Id del juego que se quiere modificar.

La imagen no se sube mediante Grpc, por eso recibimos solo el path como string, que ya se debe encontrar en el servidor.

```
message GameDTO {
    int32 id = 1;
    string title = 2;
    string synopsis = 3;
    string coverFilePath = 4;
    int32 esrbRating = 5;
    string genre = 6;
    int32 publisherId = 7;
}
```

En cuanto al UserDTO este coincide con el Usuario de Dominio sin la lista de juegos del usuario, ya que esta comienza vacía en el Post y no se modifica con el Update.
En cuanto al Post de usuario este se manda sin un Id y este se autogenera. En el Update se manda con el Id del usuario que se quiere modificar.

```
message UserDTO{
    int32 id = 1;
    string name = 2;
}
```

Para los Deletes tanto de User y Game ambos se hace la request con solo un parámetro que es el Id.

Para el Get de listas, tanto para conseguir todos los juegos o todos los usuarios estos se hacen sin parámetros, por lo tanto definimos dos requests vacías. Se podría haber implementado una única request más genérica pero creemos que de esta forma la request es más nemotécnica y si existen cambios a futuro se pueden cambiar individualmente.

```
message UsersRequest {}

message GamesRequest{}
```

Otra opción hubiera sido utilizar el Empty definido por Google, pero por el mismo motivo anterior preferimos definir nuestras propias requests.

```
rpc GetGames (google.protobuf.Empty) returns
```

Después para asociar y desasociar un juego a un usuario tuvimos la ocurrencia de nombrar Purchase a la request, que contiene el Id del usuario y el Id del Juego.

```
message Purchase {
    int32 idGame = 1;
    int32 idUser = 2;
}
```

Para la mayoría de las responses se utiliza un MessageReply, que tiene un mensaje de tipo string en donde utilizamos los mensajes de success que ya se utilizaban en el obligatorio anterior y en algunos casos se definieron nuevos para los casos de success o falla.

```
message MessageReply {
    string message = 1;
}
```

Para las responses de los Get utilizamos los mismos GameDTO o UserDTO que ya mencionamos antes para hacer algunas requests, o listas de estos.

```
message UsersResponseList{
    repeated UserDTO users = 1;
}

message GamesResponseList{
    repeated GameDTO games = 1;
}
```

Implementaciones de los servicios:

```
3 references | Agustina, 2 hours ago | 1 author, 1 change
public override Task<MessageReply> PostGame(GameDTO request, ServerCallContext context)
{
    BusinessLogicGameCUD cud = BusinessLogicGameCUD.GetInstance();
    BusinessLogicUtils utils = BusinessLogicUtils.GetInstance();
    try
    {
        Game game = new Game()
        {
            Title = request.Title,
            ESRBRating = (Common.ESRBRating)request.EsrbRating,
            CoverFilePath = request.CoverFilePath,
            Genre = request.Genre,
            Publisher = utils.GetUser(request.PublisherId),
            Synopsis = request.Synopsis,
            Reviews = new List<Review>()
        };
        string message = cud.PublishGame(game);
        return Task.FromResult(new MessageReply { Message = message });
    }
```

Se utiliza la lógica de negocio ya implementada en el Server del obligatorio anterior. Se crea un Game de Dominio con los datos del DTO de la request.

```
    catch (ServerError e)
    {
        throw new RpcException(new Status(StatusCode.InvalidArgument, e.Message));
    }
    catch (TitleAlreadyExistsException e)
    {
        throw new RpcException(new Status(StatusCode.AlreadyExists, e.Message));
    }
}
```

Se catchean dos posibles excepciones, excepciones del servidor y excepciones de validación por nombre de juego repetido. Sabemos que las excepciones arrojadas por el nuevo Server grpc siempre son RpcException, entonces aquí catcheamos las excepciones definidas en nuestro viejo server y arrojamos las nuevas, que son filtradas con un ExceptionFilter de la WebApi. En nuestro caso utilizamos StatusCode InvalidArgument para las excepciones del servidor, para cuando el cliente especificó un argumento no válido. También utilizamos StatusCode AlreadyExists para cuando el cliente intenta crear una entidad que ya existe, en nuestro caso por nombre de usuario repetido y nombre de juego repetido.

Estas excepciones luego son traducidas por el ExceptionFilter de la Api a los StatusCode que nos parecieron más apropiados.
En el caso de InvalidArgument el código de estado será 500: internal Server Error con el mensaje de error que teníamos definido del obligatorio anterior.
Para el caso de AlreadyExists el código de estado será 400: Bad Request con el mensaje de la excepción.

```
3 references | Agustina, 3 hours ago | 1 author, 1 change
public override Task<MessageReply> DeleteGame(Id request, ServerCallContext context)
{
    BusinessLogicGameCUD cud = BusinessLogicGameCUD.GetInstance();
    bool couldDelete = cud.DeleteGame(request.Id_);
    string message = couldDelete ? "Juego eliminado corretamente" : "No se pudo eliminar el juego";
    return Task.FromResult(new MessageReply{Message = message});
}
```

Existen funciones como borrar juego que no necesitan catchear ninguna excepción pues siempre se verifica la información con un booleano.

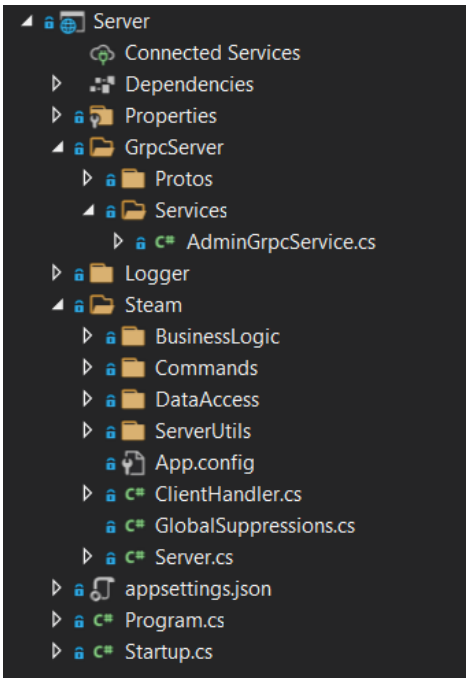
Motivación GRPC

Se decidió utilizar GRPC para la comunicación entre el AdminServer y el Server principal por diferentes motivos:

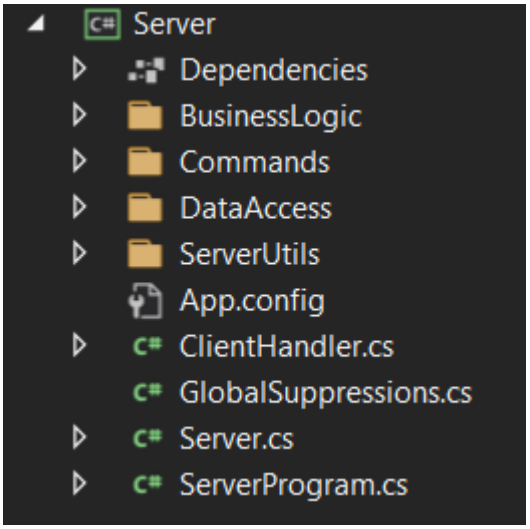
- Como nosotros controlamos ambos servidores, es más fácil pasar un archivo .proto de uno al otro y asegurarnos que usen la misma versión
- Es más rápido que usando una MQ o WebAPI
- Tiene manejo de errores. Utiliza GrpcException la cual ya contiene status code y mensajes los cuales podes utilizar para determinar los status code de la web api y los mensajes. Sería más complicado agregarle manejo de errores a la MQ
- El controlador de la WebApi utiliza el código como si fuese una clase interna, no es necesario realizar muchas configuraciones. Por ejemplo en MOM tendríamos que configurar las colas etc.
- Además si usariamos MOM tendríamos que implementar un sistema de Request/Response, el cual se puede pero no es lo ideal.

Modificaciones al Servidor Original

Server actual:



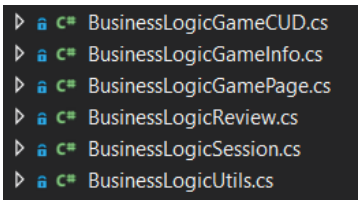
Server viejo:



Básicamente nuestro nuevo Server es un proyecto Grpc y se le agregó todo lo que estaba en el Server anterior en una carpeta dentro del proyecto llamada “Steam” (Vapor en inglés). El ServerProgram que teníamos antes en el Server se fusionó con el Program del nuevo servidor, dejando solo un Main.

Cambios a la lógica de negocio

Los cambios de la lógica de negocio se dieron principalmente en GameCUD, Utils y Session y GameInfo.



En cuanto al CUD terminamos implementando todo el CRUD, es decir fue agregada la función de GetGames que devuelve todos los juegos de la base de datos y el GetGameById, que recibe por parámetro un Id específico de un juego y lo retorna si lo encuentra.

En cuanto a Utils se implementó una función GetUser nueva. La función GetUser ya existía, y recibe por parámetro el networkStreamHandler de la conexión actual, que nos sirve cuando queremos agregar un juego y el Publisher tiene que ser el usuario que está logueado en el momento. Se hizo ahora una función con el mismo nombre que recibe el Id del usuario como parámetro, para poder añadir juegos a través del endpoint. Esta función era necesaria porque como mencionaré más adelante, la request de publicar un juego desde la Api se hace con el Id del usuario, y se tienen que hacer las verificaciones del usuario. Para poder seguir utilizando el PublishGame original de la business logic tuvimos que modificar el command handler para que se haga el set de publisher ahi mismo en lugar de como estaba antes, que era a nivel de

business logic y se mandaba el networkStreamHandle. Ahora como se puede utilizar el servicio desde la Api, no siempre hay un usuario logueado que sea el que publica el juego, si no que puede ser publicado por cualquier usuario existente.

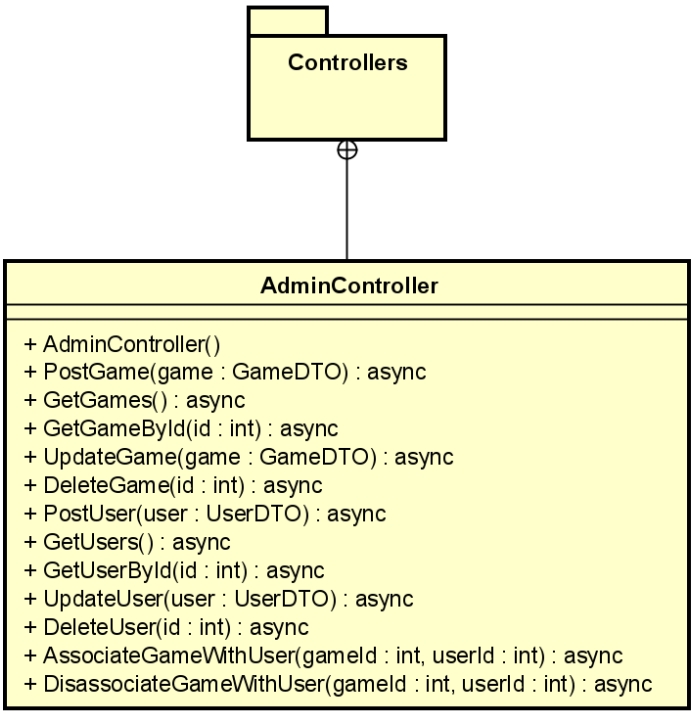
```
public override async Task ParsedRequestHandler(string[] req)
{
    BusinessLogicUtils utils = BusinessLogicUtils.GetInstance();
    Game newGame = new Game
    {
        Title = req[0],
        Synopsis = req[1],
        Genre = req[3],
        Publisher = utils.GetUser(networkStreamHandler)
};
```

Session para un usuario en nuestro Server viejo tenía un Login y un Logout nada más. Esto era porque nunca se creaba el usuario antes de loguearse, simplemente lo que se hacía en el Login era que si el usuario existe se accedía a ese y de lo contrario se creaba ahí mismo. Ahora tenemos CreateUser por separado del Login que simplemente crea el usuario. De todas formas, no se modificó el funcionamiento del Login anterior.

Se implementó en Session también el resto del CRUD del usuario, como ModifyUser, DeleteUser y GetUsers. En cuanto al obtener un usuario por su Id, existe la función GetUser en Id de la que se habló anteriormente que se encuentra en Utils. Esta decisión fue tomada porque originalmente, en los requerimientos de obligatorios pasados, no teníamos el CRUD de usuario, y el GetUser se utilizaba como utilidad para setear el Publisher en un juego.

En Info se agregaron las funciones AssociateGameToUser y ReturnGame. En un principio habíamos nombrado a AssociateGameToUser como BuyGame, función que ya existía en nuestro obligatorio anterior, pero que recibía un networkStreamHandler para conseguir el usuario, mientras que AssociateGameToUser recibe como parámetro los dos Ids, del juego y del usuario. ReturnGame hace referencia a desasociar un juego de un usuario. Consideramos que si asociar era comprar, entonces desasociar era devolver. Esta función quita el juego de la lista de juegos adquiridos por el usuario.

Cliente GRPC



Aca se pueden ver todas las funciones que manejan los distintos endpoints.

Web API

Decisiones de verbos HTTP

Para diseñar la api se diseñaron endpoint que cumplan con las recomendaciones de nombre de Rest Api, por ejemplo lo que estan detallado aca: https://restfulapi.net/resource-naming/
Algunas decisiones que se tomaron:

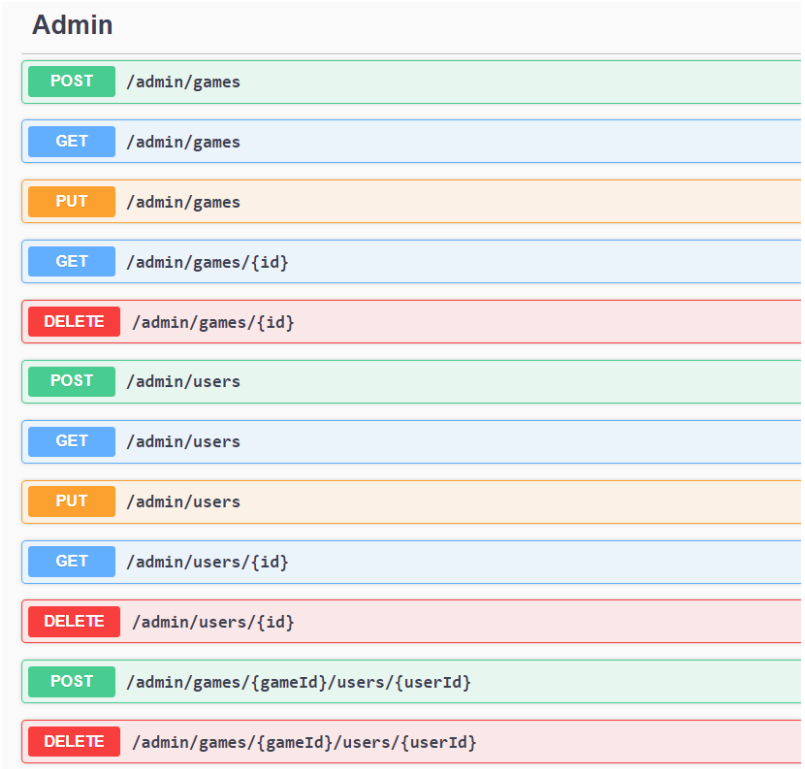
- Se definió un recurso general admin en singular al principio del endpoint, ya que solo hicimos un controlador para todos los endpoints de los servicios del servidor administrativo.
- Se definieron recursos como games y users y se escriben en plural al principio del endpoint
- Se utilizaron los métodos HTTP para indicar acciones sobre los recursos y no se incluyeron verbos en la URI. Por ejemplo:
  - GET /games/{game_id} devuelve un juego específico
  - DELETE /games/{games_id} lo elimina
- Todos los endpoints son en minúscula y no se incluye "/" al final
- Se utiliza tanto los parámetros por el body, los header o la route de la request dependiendo de que es mejor para cada caso.

URL base: http://localhost:5000/

Códigos de error:
Como se mencionó antes se establecieron códigos de error como 400 y 500 para cuando ocurre una falla en la petición del recurso. En caso que no ocurra error se devuelve

200 OK.

Endpoints:



Post de un juego

Ejemplo de la Request en el body
El Id del usuario se genera automáticamente


```
{
 "title": "Pubg",
 "synopsis": "Accion and weapons",
 "coverFilePath": "C:/Users/Agustina/Desktop/drawings/billie.jpg",
 "esrbRating": 3,
 "genre": "Otros",
 "publisherId": 1
}
```


Posibles errores:
El usuario con el Id indicado no existe:


```
{
   "message": "No se encontró el usuario, rehacer login"
}
```


Campo vacío (sucede para todos los campos indicando cual es el erróneo):


```
{
   "message": "Título no válido"
}
```

EsrbRating no válido:
El EsrbRating es un Enum. En el body de la request se utiliza el DTO del juego, en donde el esrb pasa a ser un número que luego se castea al esrb correcto, por lo tanto el número debe corresponderse a un esrb válido que son entre 0-6 inclusive.


```
{
  "message": "Clasificación ESRB no válida"
}
```


CoverPath no valido:
Se verifica que exista el path en la computadora del usuario.


```
{
  "message": "CoverPath no válido"
}
```


Genero no valido:
Se verifica que el string de género está incluido entre los géneros disponibles
{ "Acción", "Aventura", "Juego de Rol", "Estrategia", "Deporte", "Carreras", "Otros" }


```
{
  "message": "Genero no válido"
}
```


Nombre repetido:


```
{
  "message": "Ya existe un juego con ese título"
}
```


Put de un juego
Ejemplo de la Request en el body


```
{
 "id": 1,
 "title": "Pubg",
 "synopsis": "Accion and weapons",
 "coverFilePath": "C:\\Users\\Agustina\\Desktop\\drawings\biliee.jpg",
 "esrbRating": 3,
 "genre": "Otros",
 "publisherId": 1
}
```


El Id corresponde al Id del juego que se quiere modificar. Los campos se pueden dejar vacíos si estos no se quieren modificar. Si no se quiere modificar el EsrbRating se utiliza el -1.

El Delete de juego recibe el Id del juego que se quiere eliminar por ruta y no tiene body.

Paralelo a games ocurre lo mismo para users.
Ejemplo de una request por body:

Post de user:

```
{
  "name" : "agus"
}
```

Para asociar y desasociar un juego a un usuario se utiliza el mismo endpoint con un Post o Delete respectivamente.
Los datos Id se obtienen solo de la ruta y los errores posibles son si no existe el usuario, si el usuario no tiene el juego que se quiere desasociar, o si ya tiene el juego que se quiere asociar.

Manejo de errores

Para el manejo de errores se utilizó un ExceptionFilter para no tener que hacer el manejo de excepciones en el controlador y que fuera más sencillo el código.

Motivación Web Api

Se utilizaron web apis para que tanto el servidor de log como el servidor administrativo publiquen sus funcionalidades a los usuario.
Utilizar una Web Api tiene varias ventajas:

- Son muy utilizadas y fáciles de usar por un cliente
- Existen varios programas que pueden ser clientes de los servicios sin mucha configuración (ej. Postman)
- No es necesario pasarle un .proto o un archivo a los clientes
- Al seguir/utilizar REST y HTTP alguien que ya haya usado una API similar ya puede saber cómo usarla
- Es más fácil para los clientes tener solo una tecnologías como interfaz de ambos servidores y no por ejemplo, tener Grpc para uno y MQ para el otro.
- Es más fácil integrarlo a un página web donde capaz se podría hacer la interfaz del servidor administrativo o el visualizador de logs
- Es una comunicación Request/Response, lo cual es lo que estábamos buscando. El cliente hace el pedido y nosotros le respondemos.

Colección de Postman

Se incluye una colección de Postman con ejemplos de pedidos al servidor Administrativo y al servidor de Logs.

Puertos y Configuración

Se utilizaron archivos de configuración para poder editar algunos valores sin tener que recompilar la solución. Se buscó incluir cualquier valor que puede cambiar dependiendo de la máquina que se esté usando, por ejemplo, puertos, URI del Servidor de Rabbit MQ y otros.

Puertos:

Descripción	Valor por defecto	Paquete	Archivo de Config.
API Admin Server	5000	AdminServer	/Properties/launchSetting.json
API Logs Server	5002	LogServer	/Properties/launchSetting.json
GRPC	5007	Server	/Properties/launchSetting.json
GRPC	5007	AdminServer	App.config
Servidor	6000	Server y Client	App.config
Cliente	0 (Cualquiera disp.)	Client	App.config

Los últimos ya estaban en la primera entrega

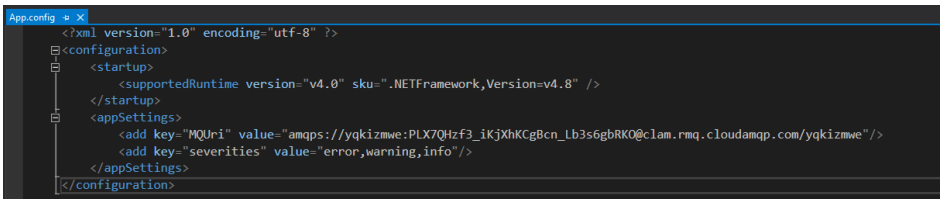
Otras posibles configuraciones (Todas en los App.config correspondientes):

Key	Descripción	Valor por defecto	Paquete
MQUri	Uri al servidor de RabbitMQ	Uri a nuestro servidor RabbitMQ en la nube	Server
MQUri	Uri al servidor de RabbitMQ	Uri a nuestro servidor RabbitMQ en la nube	LogServer
severities	Tipo de logs que se quieren recibir, separados por coma	error,warning,info	LogServer
GameCoverPath	Dirección a la carpeta donde se guardan las portadas de los juegos	C:\GameCovers\	Server
ServerIpAddress	Ip del servidor	127.0.0.1	Server y Client
ClientIpAddress	Ip del cliente	127.0.0.1	Client

Anexos

Demo MQCloud y Logs

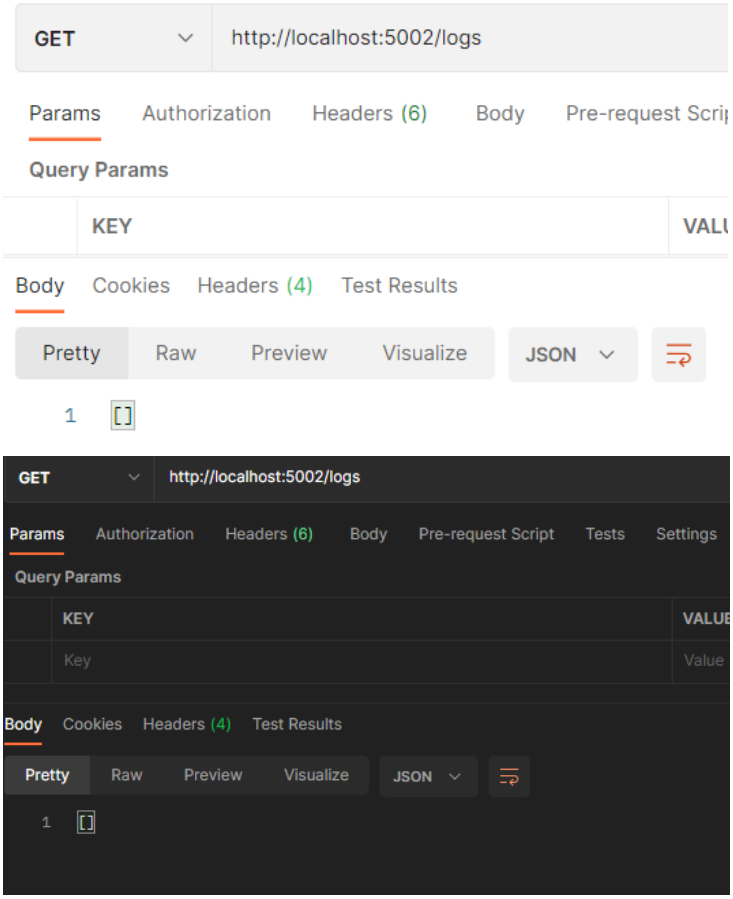
App.Config de Ivan :



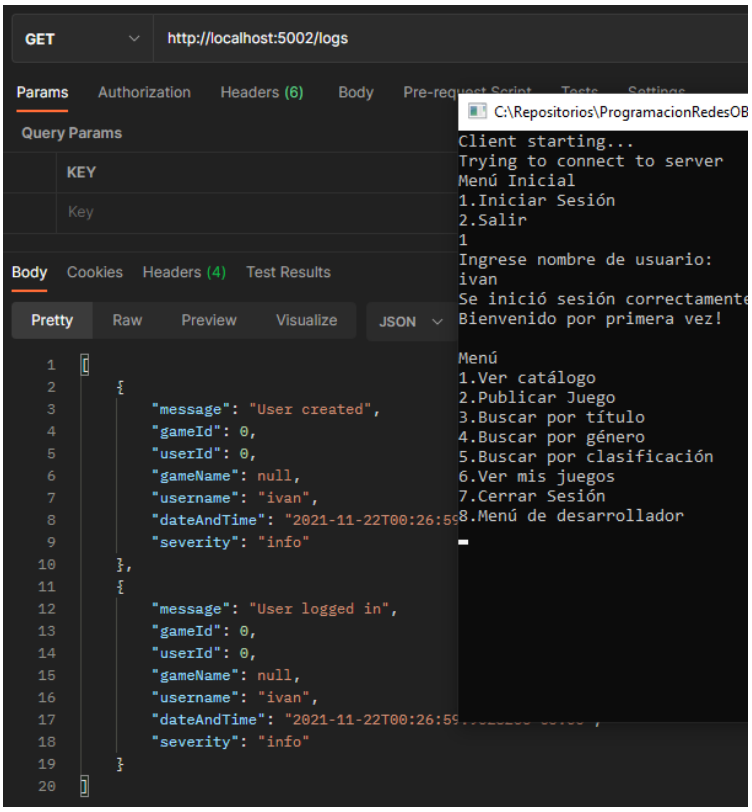
App.Config de Agustina :

```
<appSettings>
    <add key="MQUri" value="amqps://yqkizmwe:PLX7QHzf3_iKjXhKCgBcn_Lb3s6gbRKO@clam.rmq.cloudamqp.com/yqkizmwe"/>
    <add key="severities" value="warning,info"/>
</appSettings>
```

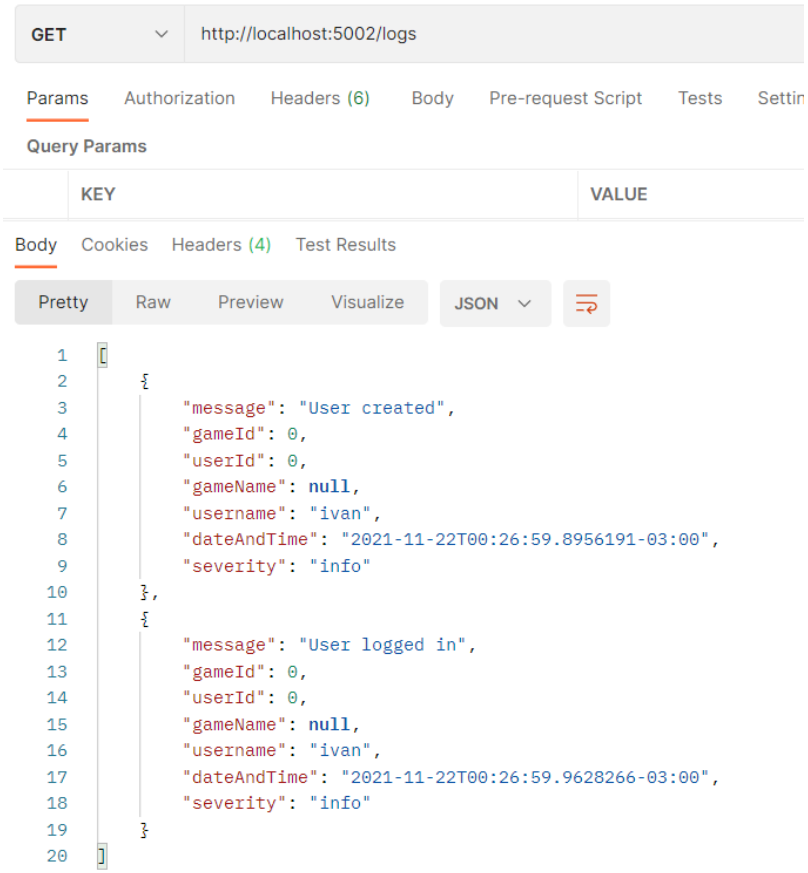
Misma key MQUri, utilizamos la nube.
Key severities: Ivan tiene todas: error, warning, info; Agustina solo warning e info.
Agustina e Ivan piden los logs desde Postman. (Agustina fondo blanco, Ivan fondo oscuro)



Ivan se loguea desde la consola como “ivan” y en los logs de info queda el registro de creación y login del usuario y procede a pedir los Logs otra vez.

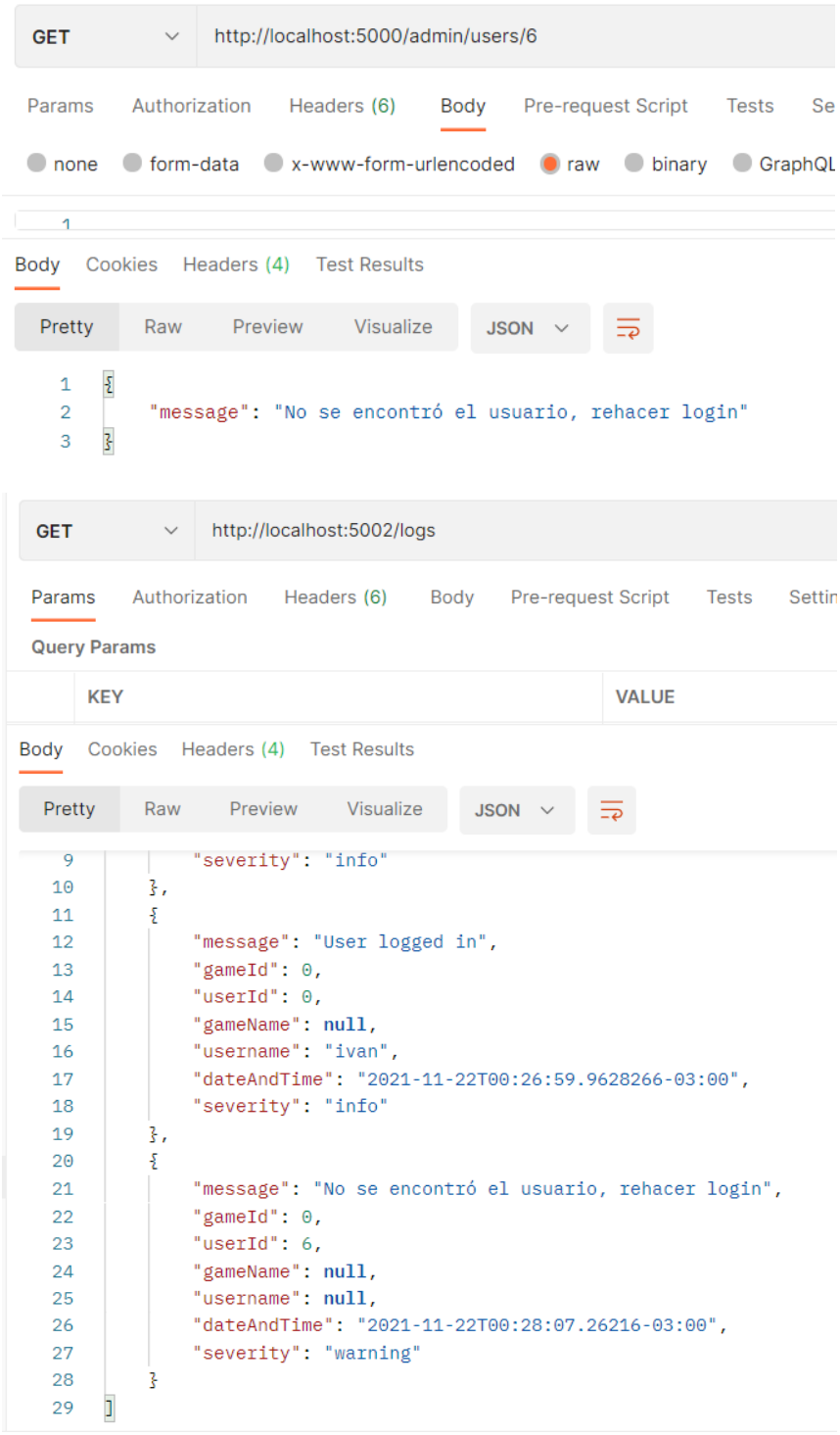


Agustina pide los Logs y se obtienen los mismos.

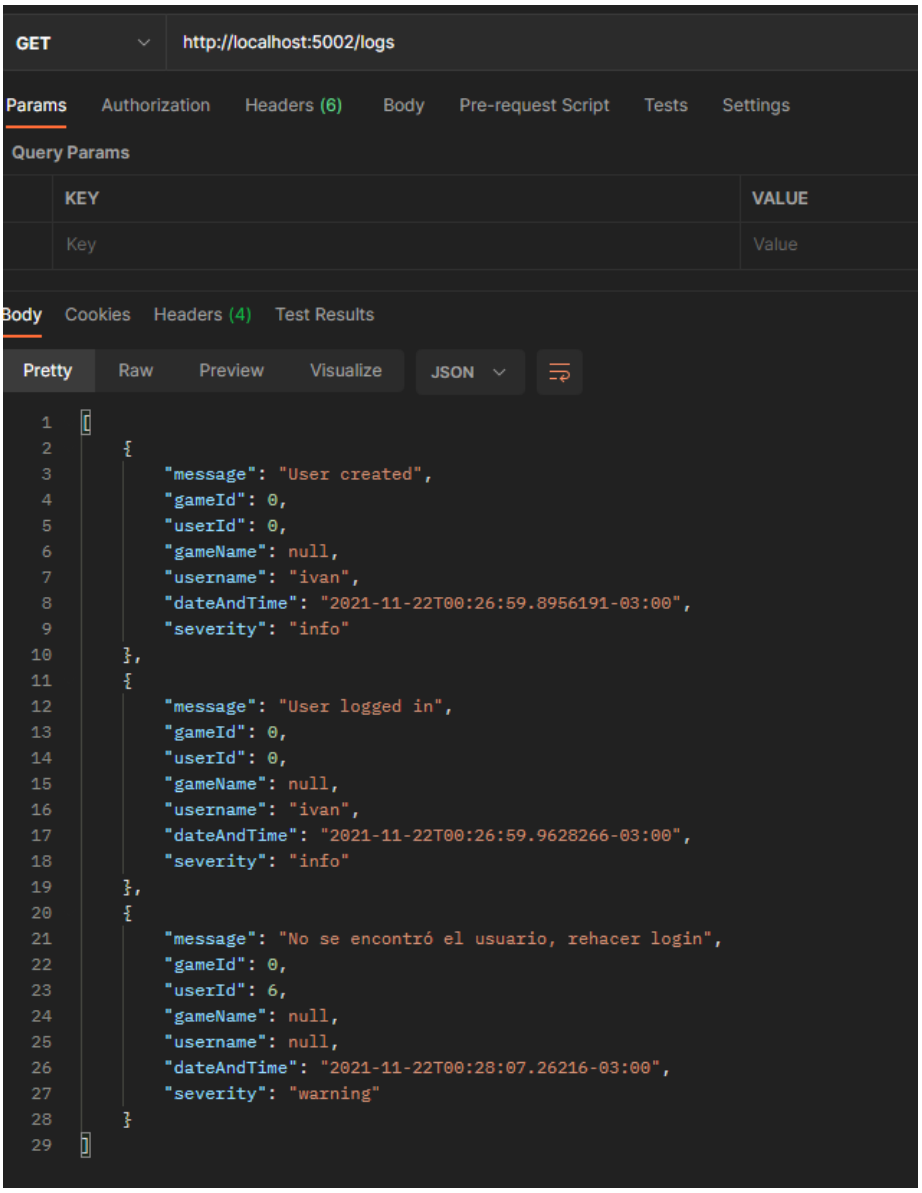



```
[
    {
        "message": "User created",
        "gameId": 0,
        "userId": 0,
        "gameName": null,
        "username": "ivan",
        "dateAndTime": "2021-11-22T00:26:59.8956191-03:00",
        "severity": "info"
    },
    {
        "message": "User logged in",
        "gameId": 0,
        "userId": 0,
        "gameName": null,
        "username": "ivan",
        "dateAndTime": "2021-11-22T00:26:59.9628266-03:00",
        "severity": "info"
    }
]
```

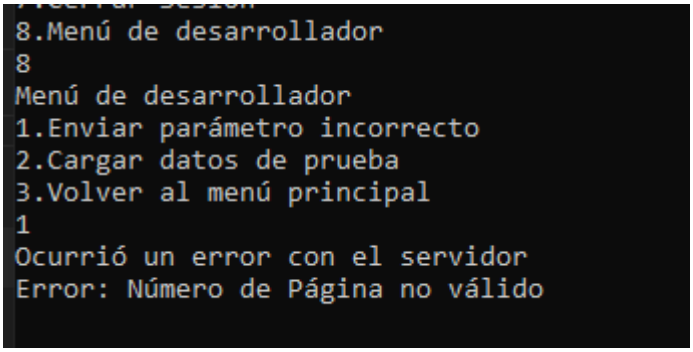
Agustina manda una request para obtener un usuario de Id inexistente, entonces se agrega un Log de tipo warning, que se ve a continuación al pedir los logs.



Ivan procede a pedir los Logs y obtiene los mismos.



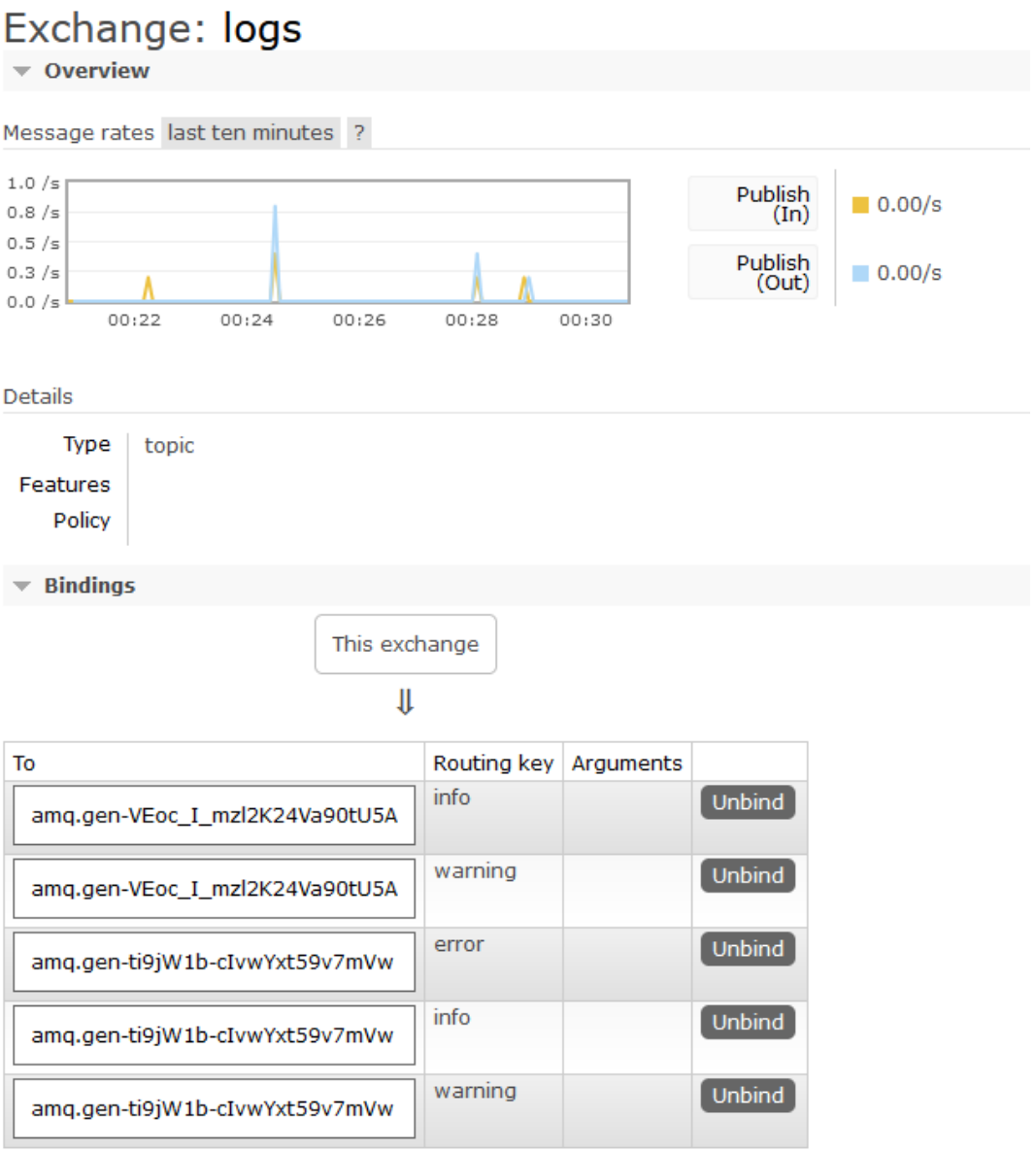
Ivan procede a mandar un parametro incorrecto desde la consola (menú del desarrollador)



Agustina e Ivan envian la request otra vez y esta vez obtienen listas distintas:


```
        "gameName": null,
        "username": null,
        "dateAndTime": "2021-11-22T00:28:07.26216-03:00",
        "severity": "warning"
    },
    {
        "message": "Sever error: Número de Página no válido",
        "gameId": 0,
        "userId": 0,
        "gameName": null,
        "username": null,
        "dateAndTime": "2021-11-22T00:29:00.5887362-03:00",
        "severity": "error"
    }
```

```
    {
        "message": "User logged in",
        "gameId": 0,
        "userId": 0,
        "gameName": null,
        "username": "ivan",
        "dateAndTime": "2021-11-22T00:26:59.9628266-03:00",
        "severity": "info"
    },
    {
        "message": "No se encontró el usuario, rehacer login",
        "gameId": 0,
        "userId": 6,
        "gameName": null,
        "username": null,
        "dateAndTime": "2021-11-22T00:28:07.26216-03:00",
        "severity": "warning"
    }
]
```



Ahi se pueden ver la interfaz en la págin de cloudMQ, donde se ve tres bindings para Iván(últimos tres) y dos bindis para Agustina (los primeros dos).