

# Universidad ORT Uruguay

## **Obligatorio 3**

### **Programación de Redes**

Agustín Ferrari 240503

Francisco Rossi 219401

2021

## *Índice:*

<b>Descripción General:</b>	<b>4</b>
<b>Alcance de la aplicación</b>	<b>5</b>
<b>Arquitectura y Mecanismo de Comunicación:</b>	<b>6</b>
RabbitMQ	7
Server	7
ServerLogs	8
REST API	9
ServerAdmin y ServerLogs	9
Manejo de errores	10
Endpoints extra	11
GRPC	12
Server y ServerAdmin	12
ServerAdmin	12
Protos	12
Server	13
<b>Control de concurrencia</b>	<b>14</b>
Mecanismos de mutua exclusión	14
<b>Descripción y justificación de diseño:</b>	<b>15</b>
Paquetes	15
Diagramas de clases:	16
Proyecto ServerLogs	16
Controllers	16
Logs	16
Models	17
Services	17
ServerLogs	17
Proyecto ServerAdmin	18
Controllers	18
Services	18
Factory	18
Filter	19
Proyecto Server	19
BusinessLogic	20
Domain	20
Logic.Commands	20
Logic	21
Proto	21

Services	21
Server	21
Proyecto CommonModels	22
Proyecto LogsModels	23
Proyecto Common	24
Configuración	25
Proyecto CommonProtocol	26
Diagramas de Interacción	27
<b>Estándares de codificación</b>	<b>28</b>
<b>Pruebas Postman</b>	<b>29</b>

## *Descripción General:*

En este proyecto el equipo se propuso como objetivo, desarrollar código profesional, fácil de entender, consistente y extensible. Para llevar a cabo esto, se implementaron técnicas aprendidas durante la carrera, como, recomendaciones de Clean Code, seguir algunos patrones de diseño, entre otras.

El equipo cuenta con 2 integrantes, por lo que se decidió en conjunto, trabajar utilizando GitHub y Git Flow, creando ramas individuales para ciertas funcionalidades y/o refactors necesarios. Se planteó de esta manera para tener la posibilidad de avanzar sin tener que ser necesario la disponibilidad horaria de los otros integrantes todo el tiempo. Para que el código pueda ser consistente y ambos estemos al tanto de los avances, se priorizó el reunirse al menos 1 o 2 veces al día para discutir y sacar dudas entre el equipo.

Se establecieron algunos estándares para trabajar en el repositorio, como por ejemplo:

- El nombre de las ramas feature es "feature-NombreDeLaRama". Se utiliza camelCase y la primera letra del nombre va con mayúscula.
- Para arreglar una funcionalidad se crea una rama "fix-NombreDeLaRama"
- Si se generó un problema al mergear a master, se puede arreglar y commitear directo a master o crear una rama Fix.

En esta documentación, nos enfocaremos en plasmar nuevas incorporaciones o cambios que se realizaron al obligatorio. Esto se debe a que mucho de lo ya implementado, no sufrió alteraciones para poder agregar los nuevos requerimientos. En caso de que se quiera revisar algún elemento en particular, recomendamos referir a las documentaciones pasadas.

## *Alcance de la aplicación*

La aplicación ofrece varias funcionalidades para los clientes. Se puede ver un listado de los juegos en el sistema sin la necesidad de iniciar sesión, y preguntando al cliente si quiere filtrar los mismos por nombre, género y/o rating. Reservamos el nombre de usuario que se elija para loguearse para que sea único en el sistema, con lo que se habilitan 8 acciones más a las de loguearse y mostrar el catálogo de juegos, que también se puede realizar una vez que inicie sesión.

Se tiene la opción de cerrar sesión, y poder volver a conectarse manteniendo los datos del usuario, como por ejemplo, ver la biblioteca, que es una funcionalidad extra que consideramos importante, para saber que juegos compró el usuario. También se puede comprar un juego, o publicarlo al sistema como desarrollador y dueño del mismo, subiendo todas sus características como la carátula o imagen de forma que todos los datos estén completos. Esto le genera permisos para modificar algún atributo del juego o eliminar el juego completamente si el usuario que requiere esta acción es el dueño.

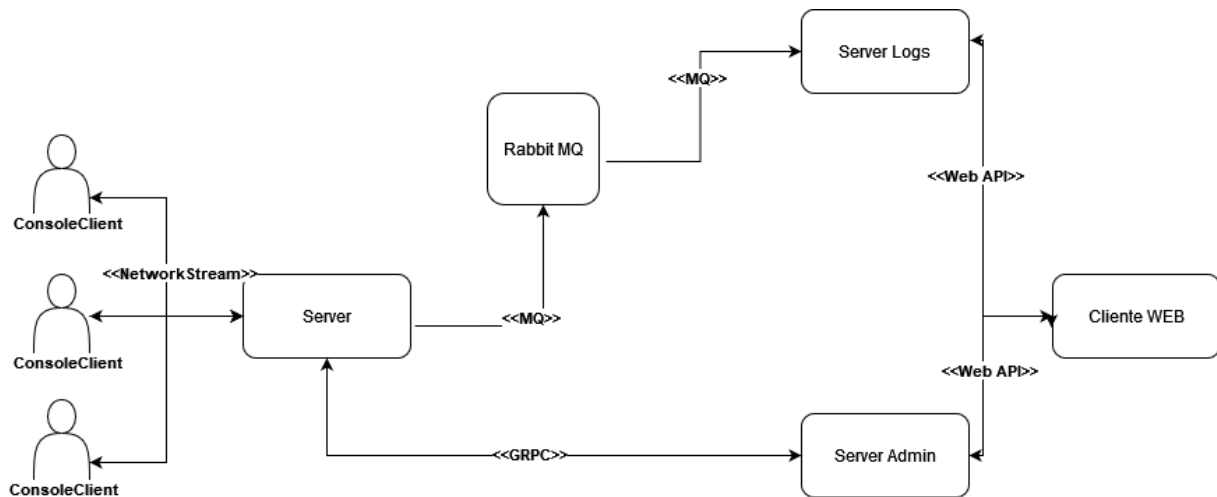
Se pueden hacer reviews del juego, no es necesario comprarlo para poder realizar esto, y luego si se quiere revisar sus calificaciones antes de comprarlo, se puede ver el detalle del juego deseado, generando un promedio de todas las reviews que los usuarios realizaron sobre este juego.

Todas estas funcionalidades mencionadas fueron realizadas para la primera entrega de este obligatorio. Se agregaron nuevas funcionalidades para esta entrega, como por ejemplo crear registros de acciones tomadas por los usuarios, generando logs para cada interacción relevante. También se añadió la baja de usuario como la baja de juegos para la biblioteca de un usuario en particular y que siga existiendo en el sistema, desde pedidos HTTP al servidor original del sistema.

Todo esto se desarrolló principalmente agregando 3 tecnologías principales: GRPC, RabbitMQ y REST API, las cuales comentaremos más en detalle a lo largo de este documento.

## Arquitectura y Mecanismo de Comunicación:

Basándonos en el obligatorio que entregamos en la primera parte, con una arquitectura cliente-servidor. Decidimos continuar con esta postura implementando dos nuevas relaciones con el servidor ya existente (Server), uno es ServerLog y otro es ServerAdmin. Ambos servidores implementan una tecnología en particular que no se había usado en el proyecto hasta ahora.



Nos basamos en este diagrama para comenzar a implementar las nuevas tecnologías requeridas para esta entrega. Se puede observar que entre los clientes y el Server no cambió nada.

Se agregaron nuevos Servers para funcionalidades distintas. ServerLogs que decidimos utilizar RabbitMQ porque no precisamos que fuera sincrónica esta comunicación. Este Server también utiliza Web API, para que desde un cliente WEB como puede ser Postman, pedir los logs del sistema según algún filtro por ejemplo.

Para el otro Server, ServerAdmin, implementamos GRPC, no solo porque era la tecnología que nos faltaba implementar, sino porque si no interesaba que esta comunicación sea performante, para que el cliente WEB, no tenga un tiempo de espera mientras se ejecuta la acción y devuelva una respuesta del resultado de la operación.

## RabbitMQ

Se utilizó RabbitMQ para crear una cola de mensajes entre el servidor principal y el servidor de logs, de esta forma, podremos delegar la lógica de guardado y consulta de mensajes al servidor de logs. La implementación de el mismo no requirió la creación de ningún protocolo, como si tuviéramos que hacer para sockets y network streams, lo cual le restó complejidad y redujo la cantidad de modificaciones y clases creadas. El protocolo que usa RabbitMQ es AMQP, por lo cual codifica los objetos que enviamos en la cola de mensajes en binario.

## Server

Para poder usar la cola de mensajes para enviar los logs que quiere guardar el sistema, tuvimos que crear la clase LogLogic, encargada de recibir los logs y enviarlos a la cola.

```
1 reference | Agustin Ferrari Luciano, 15 hours ago | 2 authors, 2 changes
public async Task SendLog(LogGameModel log)
{
    if (!_unwantedCommands.Contains(log.CommandConstant))
    {
        string message = JsonSerializer.Serialize(log);
        try
        {
            byte[] body = Encoding.UTF8.GetBytes(message);
            _channel.BasicPublish(exchange: "",
                                routingKey: "log_queue",
                                basicProperties: null,
                                body: body);
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }
    }
}
```

Para enviarlo se utiliza el simplemente el método de la foto y el mismo es llamado en la clase de ClientHandler, para aprovechar el patrón strategy y reducir las llamadas a este método.

Al tomar esta decisión de diseño, también creamos una lista de comandos que no nos interesaba logear, para esto creamos una lista de los mismos en LogLogic y los filtramos cuando se intentan agregar. Consideramos que esta fue la mejor solución ya que si hubiésemos llamado a los logs en las strategies directamente, sería más complicado cambiarlas y esto reduce la extensibilidad, de esta forma sólo necesitamos sacarlos o agregarlos a la lista de unwantedCommands.

```
CommandStrategy commandStrategy = CommandFactory.GetStrategy(header.ICommand);
LogGameModel log = await commandStrategy.HandleRequest(header, clientNetworkStreamHandler);
_logLogic.SendLog(log);
```

Para logear las acciones del servidor nos nos quedó otra que hacerlo en cada funcion que sea llamada por GRPC, si bien no cumple con la decisión que describimos anteriormente, al no

poder acceder a la función que llama cada una de las acciones que expone el servidor GRPC, no podemos hacer uso del polimorfismo ni strategy para hacer una solución más prolija.

A continuación adjuntamos una foto de los métodos, que se parecen mucho, también por el envío de errores, que en caso de haber podido acceder a la función que llama a éstas pudiéramos haberlo hecho de una manera más prolija, sin tener que repetir los try y catch en cada una de las funciones.

```
3 references | Agustin Ferrari Luciano, Less than 5 minutes ago | 2 authors, 4 changes
public override Task<UsersReply> GetUsers(UsersRequest request, ServerCallContext context)
{
    LogGameModel log = new LogGameModel(CommandConstants.ModifyPublishedGame);
    log.User = request.UserAsking;
    string response;
    try
    {
        response = "Usuarios en el sistema: \n" + _usersController.GetAllUsers();
        log.Result = true;
        _logLogic.SendLog(log);
        return Task.FromResult(new UsersReply
        {
            Response = response
        });
    }
    catch (InvalidUsernameException e)
    {
        log.Result = true;
        _logLogic.SendLog(log);
        throw new RpcException(new Status(StatusCode.NotFound, e.Message));
    }
}
```

## ServerLogs

Este proyecto se encarga de consumir la cola de mensajes y almacenarlos, esperando a que se soliciten a través de un pedido http.

Para consumir la misma se creó un paquete RabbitMQService, RabbitHutch crea la conexión, RabbitBus es el encargado de consumirla, por último está la clase Queue que define la queue a consume.

```
2 references | FranRossi, 12 days ago | 1 author, 3 changes
public async Task ReceiveAsync<T>(string queue, Action<T> onMessage)
{
    _channel.QueueDeclare(queue, false, false, false, null);
    var consumer = new AsyncEventingBasicConsumer(_channel);
    consumer.Received += async (s, e) =>
    {
        var body = e.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        var item = JsonConvert.DeserializeObject<T>(message);
        onMessage(item);
        await Task.Yield();
    };
    _channel.BasicConsume(queue, true, consumer);
    await Task.Yield();
}
```

De este lado solo nos interesa consumir la cola, por lo cual este es el único método definido en RabbitBus.



# REST API

## ServerAdmin y ServerLogs

ServerAdmin cumple las funcionalidades de alta, baja y modificación de usuario. También se le agregó alta, baja y modificación de juego incluyendo poder comprar o eliminar un juego para un usuario en particular, si este lo había comprado.

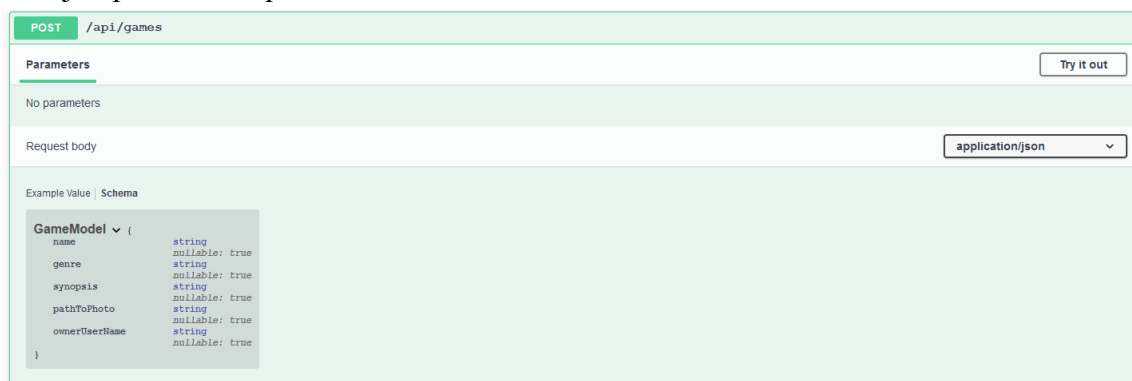
Para la implementación de este servidor, se utilizó REST API, el cuál es un estilo de arquitectura que plantea ciertas restricciones y algunas propiedades que cumplir.

Algunos conceptos que manejamos al crear nuestro endpoints siguiendo REST:

- Uniform Interface:

La interfaz se tiene que basar en recursos, para saber con que se está queriendo interactuar. Debe contar con suficiente información de cómo procesar el mensaje por ejemplo si se envía un objeto JSON.

Por ejemplo este endpoint:



The screenshot shows a REST client interface for a POST request to the endpoint `/api/games`. The interface includes a "Parameters" section with "No parameters" and a "Request body" section with a dropdown menu set to "application/json". Below these sections, there is a "Schema" tab showing a JSON schema for a `GameModel` object. The schema defines the following fields: `name` (string, nullable), `genre` (string, nullable), `synopsis` (string, nullable), `pathToPhoto` (string, nullable), and `ownerUsername` (string, nullable).

```
GameModel {
  name: string (nullable: true)
  genre: string (nullable: true)
  synopsis: string (nullable: true)
  pathToPhoto: string (nullable: true)
  ownerUsername: string (nullable: true)
}
```

Su Uniform Resource Identifier (URI) indica que se trata de un juego, en este caso al tener un verbo post, se entiende que se desea crear un juego al utilizar este endpoint. Se indica un modelo propio en formato JSON para poder ser interpretado.

- Stateless:

Cada request o solicitud debe contener toda la información necesaria para comprender la solicitud. Cada request/response entre el cliente y el servidor es independiente al anterior. No hay forma de mantener “un canal” de comunicación “abierto” permanentemente. Esto permite darle escalabilidad al sistema, se puede redirigir la request por distintos caminos para obtener el mismo resultado.

- Client-Server:

Los cliente no se deben preocupar de cómo se guarda la información, en este caso se utiliza swagger y Postman para obtener y modificar recursos, pero si se quisiera implementar un sistema para manejar los resultado del lado del cliente, no cambiaría como se utiliza los endpoints ya desarrollados. A su vez el server, no le interesa como el cliente maneja los datos solicitados por medio de request.

ServerLog también utiliza REST complementando con RabbitMQ. Este servidor, contiene solo un endpoint para poder solicitar los logs del sistema, filtrando por distintos parámetros como lo son juegos, usuarios, fecha o alguna combinación de estas.

Algunas recomendaciones que decidimos seguir en la mayoría de los casos:

- Nunca usar verbos en las URI, ya sabemos que acción realizar dado el verbo del HTTP request, por ejemplo GET, POST, DELETE, PUT(modificar). Si usar sustantivos. Por ejemplo no utilizar DELETE /deleteUser, sino DELETE /user/identificador.
- Utilizar query parameters, por ejemplo cuando se está filtrando en algún request. ?game=fifa por ejemplo.
- No extender más de 3 niveles los recursos: recurso/identificador/subrecurso por ejemplo POST /users/{gameToBuy} para la acción de comprar un juego.

**POST** /api/users/{gameToBuy}

**Parameters**

Name	Description
userBuying string (header)	userBuying
gameToBuy * required string (path)	gameToBuy

## Manejo de errores

Actualmente indicamos en cada request un mensaje de éxito o de algún en específico, por ejemplo:

**Curl**

```
curl -X DELETE "https://localhost:5001/api/games?game=fifa" -H "accept: */*" -H "userAsking: pepe"
```

**Request URL**

```
https://localhost:5001/api/games?game=fifa
```

**Server response**

Code	Details
200	<b>Response body</b> El juego fifa fue borrado correctamente

Pero cuando ocurre un error en el sistema, controlamos el error con un filtro. Si el Server lanza una `RPCException`, definimos un código correspondiente y mostramos un mensaje de error adecuado. Hasta el momento contamos con 3 códigos de estado de error, 404 si el recurso no se encuentra en el sistema, 409 si el recurso que se quiere agregar ya existe y 500 si el sistema lanza un error no manejado.

Por ejemplo, luego de haber eliminado el juego Fifa del sistema por el dueño del mismo, si otro usuario lo intenta comprar, pasa lo siguiente:

**Responses**

Curl

```
curl -X POST "https://localhost:5001/api/users/fifa" -H "accept: */*" -H "userAsking: kaka" -d ""
```

Request URL

```
https://localhost:5001/api/users/fifa
```

Server response

Code	Details
404 <i>Undocumented</i>	Error: Not Found Response body Juego no registrado en el sistema

El mensaje es autodescriptivo y el número de error indica que hubo un problema al ejecutar esta acción.

## Endpoints extra

Algo que sí hicimos extra con esta tecnología, fue añadir endpoints extras. Creamos dos adicionales para poder verificar muchas acciones que otros endpoints realizan al sistema.

Implementamos que se puedan traer todos los juegos del sistema y además todos los usuarios registrados en nuestro Server.

**Responses**

Curl

```
curl -X GET "https://localhost:5001/api/games" -H "accept: */*" -H "userAsking: pepe"
```

Request URL

```
https://localhost:5001/api/games
```

Server response

Code	Details
200	Response body Juegos en el sistema: Minecraft Call of Duty Hades Response headers content-type: text/plain; charset=utf-8 date: Sun21 Nov 2021 23:56:09 GMT server: Kestrel x-firefox-spdy: h2

```
Curl
curl -X GET "https://localhost:5001/api/users" -H "accept: */*" -H "userAsking: pepe"

Request URL
https://localhost:5001/api/users

Server response

Code    Details
200

Response body
Usuarios en el sistema:
pepe
kaka

Response headers
content-type: text/plain; charset=utf-8
date: Sun21 Nov 2021 23:57:09 GMT
server: Kestrel
x-firefox-spdy: h2
```

## GRPC

### Server y ServerAdmin

Se utilizó RPC para la comunicación entre ServerAdmin y Server. Más en concreto se utilizó GRPC, tecnología desarrollada por Google para la comunicación e interacción entre dos aplicación que viven en diferentes contextos, como en este caso es ServerAdmin y Server.

### ServerAdmin

En nuestro caso ServerAdmin envía una solicitud luego de ser procesada por un endpoint para conocer qué acción se solicitó y luego el Server, maneja esa solicitud por medio de esta tecnología otorgando una respuesta al finalizar. El mecanismo de ejecución es sincrónico, cuando el ServerAdmin realiza la solicitud se suspende hasta que el Server devuelve una respuesta y haya terminado de procesar el pedido.

### Protos

Para poder implementar esta tecnología, se necesita utilizar Protocol buffers, un mecanismo eficiente para serializar la comunicación entre los Servidores viviendo en un contexto distinto. Para hacerlo funcionar, se define en archivos .proto cómo se va a serializar la información a mandar por medio de mensajes, estos mensajes es una entrada que contiene un par clave-valor. Este archivo se define en ambos puntos de comunicación para poder descifrar como es la comunicación y que mensajes esperan recibir según las clave-valor.

Por ejemplo estos son los mensajes que definimos para que se pueda comunicar el ServerAdmin con el Server, luego de accionar el endpoint de modificar un juego:

```
message ModifyGameRequest{
    string Name = 1;
    string Genre = 2 ;
    string Synopsis = 3 ;
    string PathToPhoto = 4 ;
    string OwnerUserName = 5 ;
    string GameToModify = 6;
}
```

```
message GamesReply {
    string response = 1;
}
```

Existe un mensaje de solicitud o request y otro de respuesta. Se define el mismo mensaje en ambos .proto de los servidores, y se define un servicio, en este caso

```
rpc ModifyGame (ModifyGameRequest) returns (GamesReply);
```

que será el expone el servicio para poder realizar una implementación en el Server.. En nuestro caso implementamos un servicio unario, una llamada simple que realiza el cliente y el servidor responde.

## Server

Definimos un paquete Services en el proyecto Server. Para poder implementar los servicios declarados en el .proto.

Al realizar una petición desde el ServerAdmin, por ejemplo modificar un juego del sistema:

```
public async Task<string> ModifyGame(string gameToModify, GameModel model)
{
    ModifyGameRequest request = new ModifyGameRequest()
    {
        Name = model.Name,
        Genre = model.Genre,
        Synopsis = model.Synopsis,
        OwnerUserName = model.OwnerUserName,
        PathToPhoto = model.PathToPhoto,
        GameToModify = gameToModify
    };
    var response :ModifyGameReply = await _client.ModifyGameAsync(request);
    return response.Response;
}
```

El servidor se queda esperando (await) hasta que el Server le responda. Para esto debemos realizar una implementación del servicio definido en el archivo .proto (ModifyGame)

```

3 references | Agustín Ferrari Luciano, 5 minutes ago | 2 authors, 3 changes
public override Task<GamesReply> ModifyGame(ModifyGameModelRequest modifyGameModelRequest, ServerCallContext context)
{
    LogGameModel log = new LogGameModel(CommandConstants.ModifyPublishedGame);
    log.User = modifyGameModelRequest.OwnerUserName;
    log.Game = modifyGameModelRequest.Name;
    string response;
    try
    {
        Game newGame = ParseGameModelToGame(modifyGameModelRequest);
        Game oldGame = _gamesController.GetGame(modifyGameModelRequest.GameToModify);
        _gamesController.ModifyGame(oldGame, newGame);
        response = "El juego " + modifyGameModelRequest.Name + " fue modificado correctamente";
        log.Result = true;
        _logLogic.SendLog(log);
        return Task.FromResult(new GamesReply
        {
            Response = response
        });
    }
    catch (Exception e) when (e is InvalidGameException || e is InvalidUsernameException)
    {
        log.Result = false;
        _logLogic.SendLog(log);
        throw new RpcException(new Status(StatusCode.NotFound, e.Message));
    }
}

```

Luego de realizar el pedido, GRPC automáticamente detecta la implementación del servicio en el Server y se ejecuta. Al mandar por mensajes de clave-valor, tenemos que parsearlo a una entidad como un juego en este caso para poder realizar la acción en la BusinessLogic correspondiente.

## *Control de concurrencia*

### Mecanismos de mutua exclusión

Al permitir la conexión de múltiples clientes que potencialmente puedan querer acceder a los mismos recursos como listas de usuarios, juegos, etc. Esto puede traer errores como lectura sucia o actualización perdida. Para solucionar este problema se decidió implementar un mecanismo de mutua exclusión basado en locks, bloqueando la lectura y escritura de las listas en caso de que otro thread quiera acceder a las mismas. De esta manera, sólo un proceso puede acceder a las listas a la vez y nos aseguramos de que no ocurran estos problemas.

Las dos listas principales del sistema son la de juegos y usuarios, las mismas se encuentran en business logic dentro de sus controladoras respectivas. Por cada una de estas listas se utiliza un lock, es decir, se puede leer/escribir un juego, aunque el lock de los usuarios esté activado.

Este mecanismo se amplió también para utilizarlo en el ServerLogs donde se manejan diccionarios para mantener logs según distintos criterios, y queremos prevenir que se realicen accesos indebidos a los mismos ya que soportamos multiconexión en nuestra aplicación.

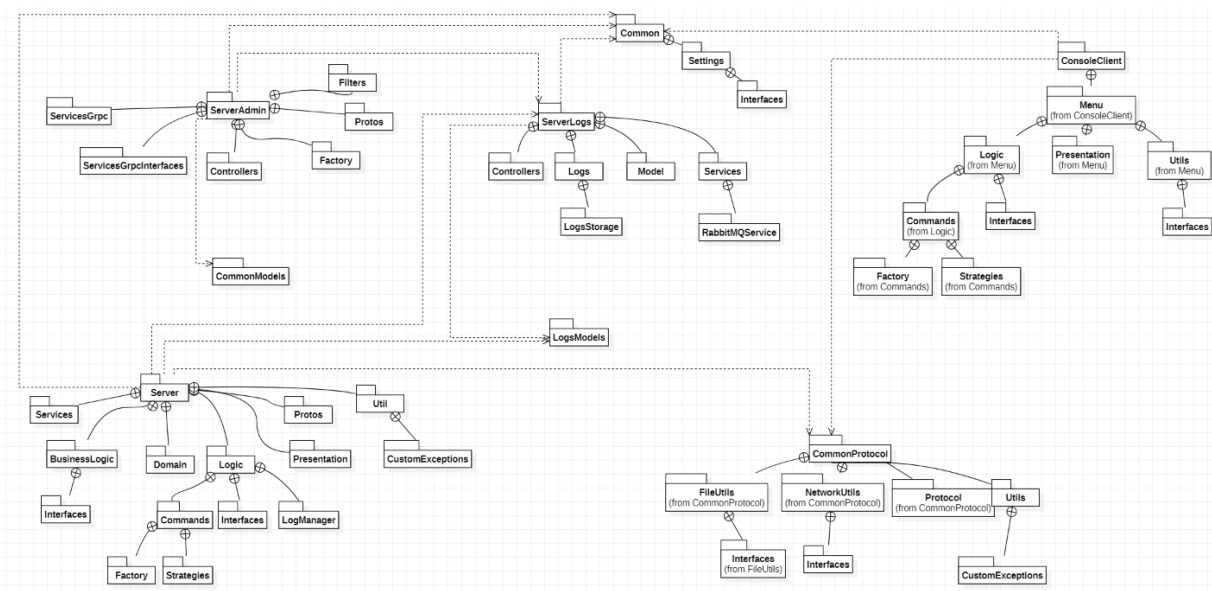
## Descripción y justificación de diseño:

### Paquetes

Basándonos en las entregas pasadas, el proyecto Common y ConsoleClient, casi no sufrieron cambios para implementar las nuevas tecnologías solicitadas (REST, RabbitMQ y GRPC).

En el proyecto ConsoleClient, para resolver el problema de no contar con un switch extenso, se conservó la solución ya implementada para que el sistema sea escalable. En el paquete Logic\Commands, se crearon dos paquetes, Factory y Strategies. Se decidió utilizar el patrón Factory, encargado del manejo de las strategies. Factory es el encargado de recurrir a la strategy correcta según el caso de la solicitud, donde el comando de acción involucra una implementación concreta de la misma. Estas decisiones de diseño fueron implementadas tanto en el cliente como en el Server. Esto sirvió y demostró que para agregar nuevas opciones dentro del menú e implementar nuevas funcionalidades, era simple poder realizarlo sin tener que editar código ya existente.

Para esta entrega, se agregaron proyectos nuevos con varios paquetes dentro de ellos. Entre ellos se encuentra ServerAdmin, ServerLogs, CommonModels, LogsModels y Common, nombrando nuestro viejo Common a CommonProtocol. A continuación se dividirá en diagramas de clases por proyectos, explicando los cambios o incorporaciones a la solución.



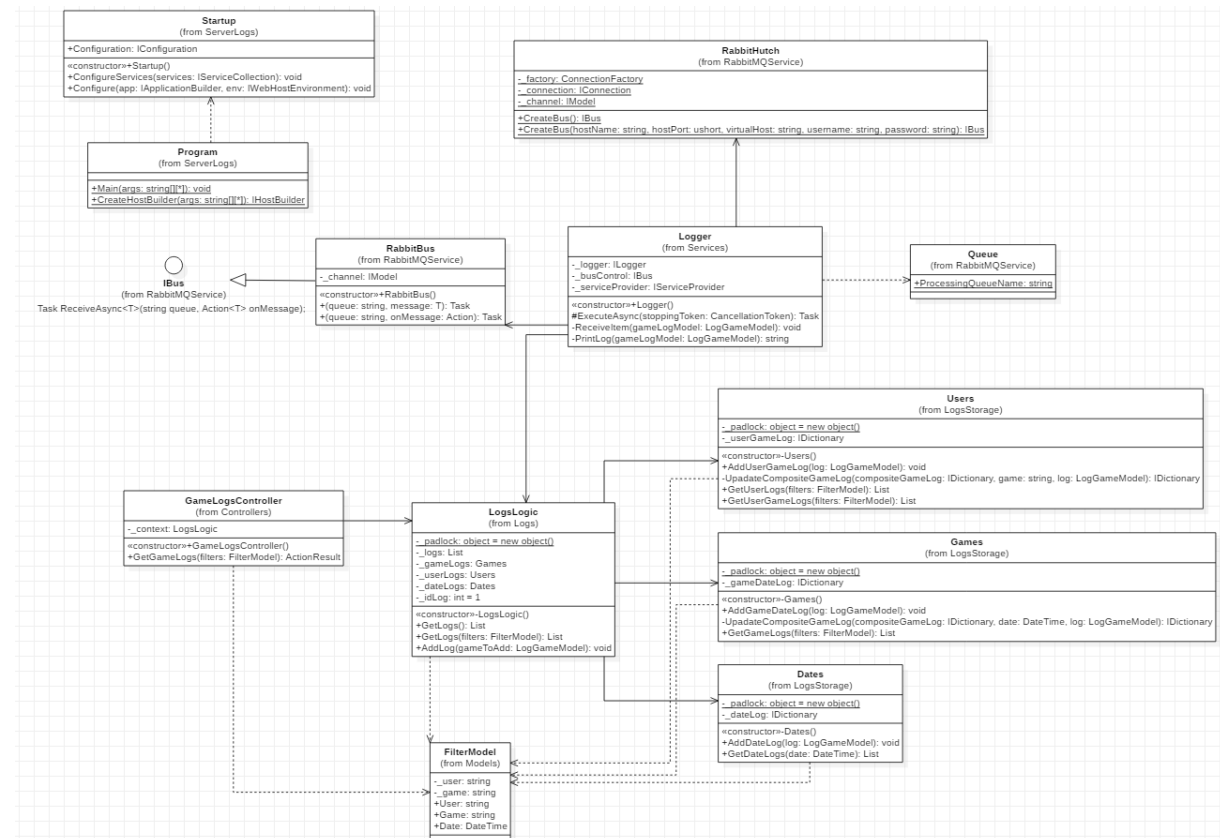
1

<sup>1</sup> Diagrama encontrado en Diagrams/Packages.SVG

## Diagramas de clases:

Para entender más específicamente cómo diseñamos nuevo cada paquete, podemos observar los siguientes Diagramas de Clases, comprendidos en diferentes proyectos:

### Proyecto ServerLogs



2

### Controllers

En este paquete se puede encontrar la controladora encargada de responder las consultas http del servidor de logs. En este caso es solo una clase, GameLogsController y es encargada de responder a el único endpoint de este proyecto, el de obtener logs.

### Logs

Este paquete es el encargado de manejar la lógica de logs, esta dividido en 4 clases:

- LogsLogic: Esta clase se encarga de recibir las consultas de la base de datos y agregar los logs, delegando la responsabilidad de los logs con filtros específicos a las clases que siguen y manteniendo una lista de logs ordenada únicamente por orden de adición dentro de esta clase.

<sup>2</sup> Diagrama encontrado en Diagrams/ServerLogs.svg



- Dates: Esta clase como lo dice el nombre se encarga de guardar los logs indexados por fecha, para esto decidimos usar un diccionario, con la clave fecha y una lista de logs como valor, conteniendo todos los logs de un día dado.
- Games: Al igual que la clase anterior, esta clase se encarga de guardar los logs indexados por juego, para facilitar la búsqueda por los mismos. La diferencia con el diccionario anterior es que si bien este también tiene como clave el nombre del juego, el valor no es simplemente una lista de logs para dicho juego, sino que otro diccionario igual al anterior, es decir con clave fecha y valor lista de logs.

De esta forma estamos optimizando mucho más la búsqueda por los logs de un juego en una fecha dada, ya que intuimos que es más probable que se realicen este tipo de búsquedas que las de los logs de un juego únicamente, ya que pueden haber miles por día de los mismos. Cabe destacar que si bien pensamos la estructura para que funcione mejor en este caso, también es posible buscar únicamente por juego, como por cualquier otra combinación de los posibles parámetros de filtro (juego, fecha y usuario).

User: Por último pero no menos importante tenemos la clase user, encargada de guardar los logs indexados por user, al igual que la clase anterior, contiene un diccionario con clave de nombre del usuario y valor otro diccionario con clave nombre de juego y valor una lista de logs. Análogo a la justificación anterior, se pensó en esta estructura para optimizar la búsqueda por usuario y juego, ya que pensamos que cuando estén presentes los tres filtros, el filtro más efectivo para aplicar primero sería el de usuario, luego el de juego y por último el de fecha, hay que destacar que no se decidió hacer un diccionario de tres niveles, ya que pensamos que una vez se filtre por usuario y luego por juego, es probable que la cantidad de logs se haya reducido en una cantidad muy considerable, por agregar un diccionario mas por fecha para las pocas que quedarían no iba a ser necesario.

## Models

Este paquete únicamente contiene una clase concreta, FilterModel, que actúa como DTO para cargar los datos por los que el usuario quiera filtrar cuando utilice el endpoint para obtener los logs.

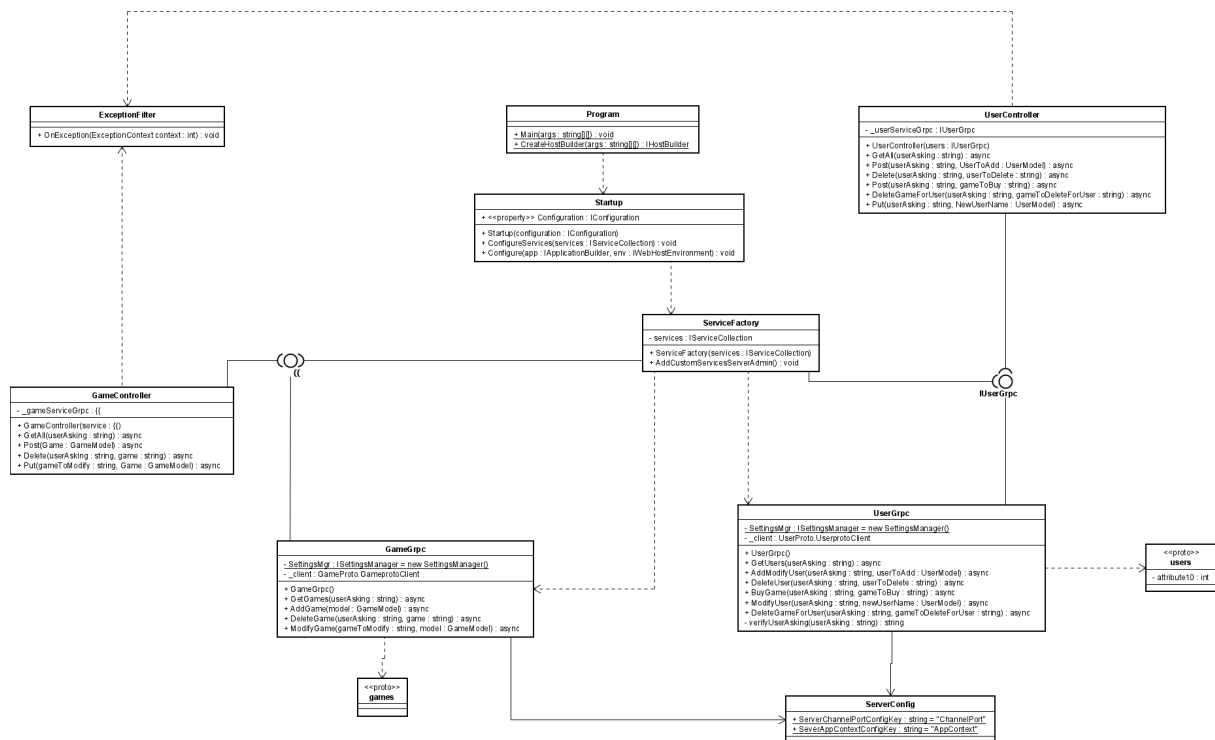
## Services

Este paquete se encarga de consumir los logs ingresados a través de RabbitMQ, para eso se utilizan las clases de RabbitMQService para crear el canal de conexión y poder consumir los datos de la cola de mensajes, RabbitHutch crea la conexión, RabbitBus es el encargado de consumirla, por ultimo esta la clase Queue que define la queue a consumir.

## ServerLogs

Al tratarse de una WebApi, al crear la misma se crearon las clases de Program y Startup pero no se realizaron modificaciones a ninguna de ellas.

# Proyecto ServerAdmin



3

En este diagrama se encuentran las controladoras como UserController, GameController y también los servicios UserGrpc, GameGrpc.

## Controllers

Las controladoras fueron implementadas para definir los endpoints. Estos se ejecutan por HTTP requests para cumplir los nuevos requerimientos solicitados.

Dentro de estos endpoints se encuentran ABM de juegos, ABM de usuarios y la compra de un juego para un usuario particular o dar de baja un juego en su biblioteca.

## Services

Los servicios UserGrpc y GameGrpc, se crearon para poder utilizar GRPC entre dos proyectos que están en contextos diferentes. Para esto, utilizan las declaraciones del archivo .proto correspondiente y hacer poder enviar y recibir mensajes entre ServerAdmin y Server. Ambos servicios implementan una interfaz específica para cada uno para exponer sus métodos.

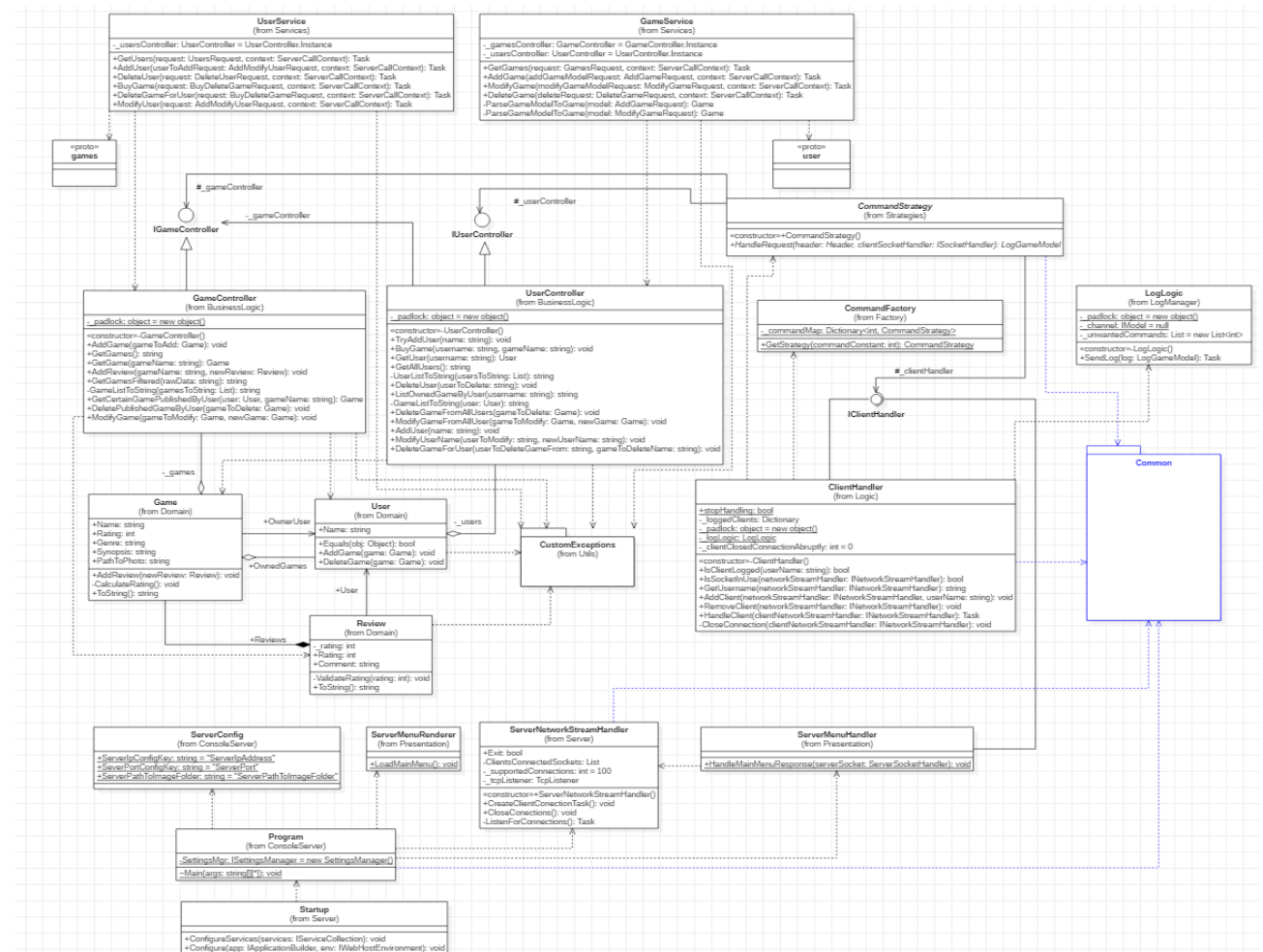
## Factory

Se creó este paquete, con ServiceFactory para poder desacoplar las controladoras de los servicios. Eliminamos una dependencia directa con los servicios concretos, ya que ahora las

<sup>3</sup> Diagrama encontrado en Diagrams/ServerAdmin.SVG

Filter

## Proyecto Server



<sup>4</sup> Diagrama encontrado en Diagrams/Server.svg

## BusinessLogic

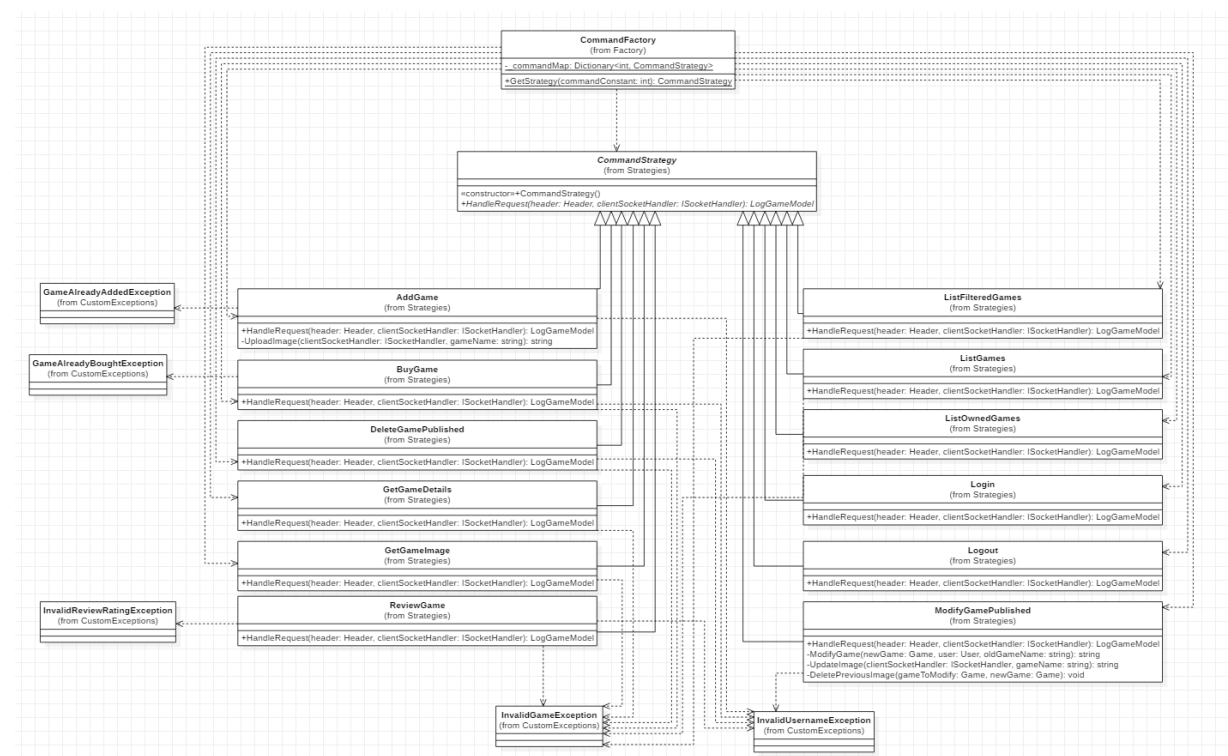
En esta entrega se tuvo que modificar únicamente la clase UserController, agregando un método para agregar usuarios que devuelve una excepción si ya está creado y otro para eliminar o “devolver” un juego que el usuario había comprado. Ambos cambios fueron productos de los nuevos requerimientos de Server Admin.

## Domain

Al igual que en business logic, la única clase que se modificó para esta entrega fue la de usuario, agregando la funcionalidad de poder eliminar un juego de la lista de juegos que había adquirido.

## Logic.Commands

Para poder hacer los logs de la parte de los clientes por consola, se tomó la decisión de agregarle a los métodos de la strategy creada anteriormente, que eran void, la capacidad de devolver un log. De esta forma únicamente tenemos que llamar a la clase que se encarga de hacer los logs una única vez en el ClientHandler. Como no nos interesan todos los logs, decidimos filtrar algunos como la obtención de detalles de un juego o pedir la lista de juegos en LogLogic.



## Logic

El mayor cambio de logic fue la adición de LogLogic, clase que se encarga de enviar los logs a la cola de mensajes de RabbitMQ, también como fue explicado anteriormente, se tuvo que actualizar ClientHandler para hacer uso del factory y facilitar también el envío de logs con una única llamada a la clase LogLogic.

```
CommandStrategy commandStrategy = CommandFactory.GetStrategy(header.ICommand);
LogGameModel log = await commandStrategy.HandleRequest(header, clientNetworkStreamHandler);
_logLogic.SendLog(log);
```

También nos aprovechamos de que el método era asíncrono para poder mandar el log de esta manera y no tener que esperar para poder seguir procesando requests del cliente.

## Proto

El paquete proto es el encargado de contener los protos que se utilizarán para la comunicación GRPC.

Cuenta con dos protos, user y games, encargados de definir los messages y servicios GRPC que se utilizan, de usuarios y juegos respectivamente.

## Services

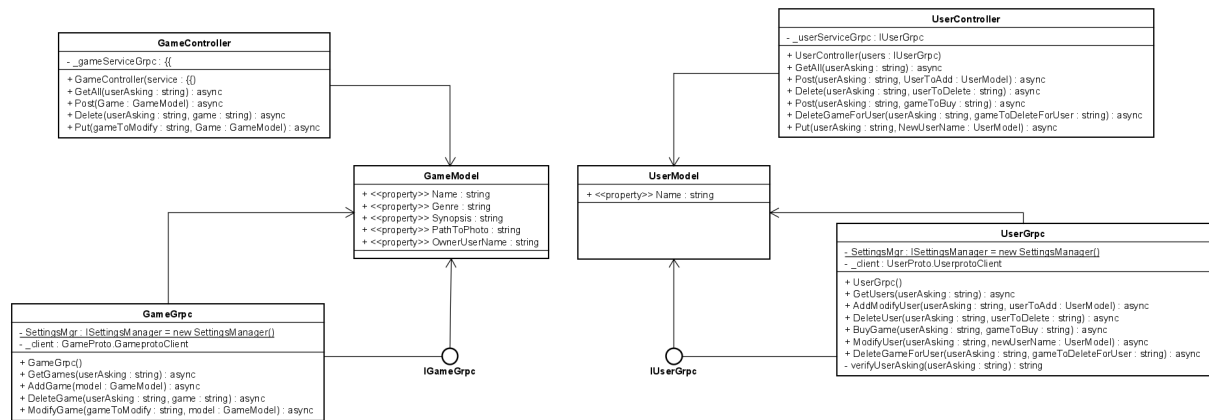
Este paquete es el encargado de recibir las consultas GRPC, el mismo cuenta con dos clases, UserService y GameService, encargados de recibir las consultas de usuarios y juegos, como su nombre lo indica. Una vez reciben la consulta se comunican con la business logic y usando las clases y métodos que habíamos ya definido para la primera entrega, hacen los pedidos pertinentes. También se encargan de capturar los casos excepcionales, haciendo uso de las excepciones custom que tenemos definidas en utilities.

## Server

Por último, el paquete root del proyecto, al cual se le agregaron clases como startup, ya que ahora actúa también como servidor GRPC y se modificó el program también por esta misma razón.

También en la segunda entrega se modificó la clase que antes se encargaba de manejar el socket, dado que tuvimos que pasarnos a network stream, cambiando el nombre de este a NetworkStreamHandler, pero la logica detras de esta clase sigue siendo la misma que describimos en la primera documentación.

## Proyecto CommonModels



6

Definimos un modelo de un juego para poder ser utilizado en las controladoras, parseando un JSON a un objeto con properties predefinidas y luego que los servicios de GRPC en el ServerAdmin, también puedan acceder a sus properties para mandar un mensaje, Por ejemplo

```

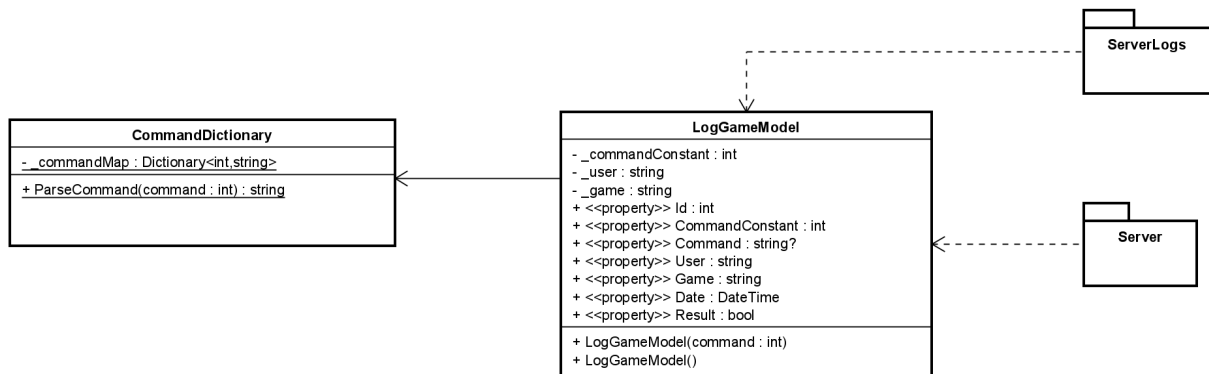
public async Task<string> AddGame(GameModel model)
{
    AddGameRequest request = new AddGameRequest()
    {
        Name = model.Name,
        Genre = model.Genre,
        Synopsis = model.Synopsis,
        OwnerUserName = model.OwnerUserName,
        PathToPhoto = model.PathToPhoto
    };
    var response : AddGameReply = await _client.AddGameAsync(request);
    return response.Response;
}

```

Se implementó en un proyecto separado, por si se desea implementar otra tecnología a futuro, se puede utilizar estos modelos de las entidades del dominio, para poder manipularlas sin tener que conocer el dominio del sistema.

<sup>6</sup> Diagrama encontrado en Diagrams/CommonModels.svg

## Proyecto LogsModels



7

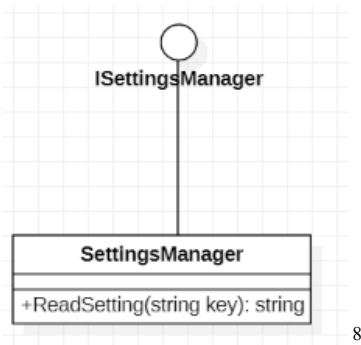
Este modelo se definió con el objetivo de poder manipular la información en el proyecto de ServerLog, agrupando la información que requerida fuera del manejo de la misma. Este proyecto cuenta con su clase principal, **LogGameModel**, donde cuenta con ciertas properties, cómo **User**, **Game** y **Date** que son posibles filtro de los logs guardado en el sistema, según lo solicite el cliente. También existe una clase auxiliar, **Command Dictionary**, para poder utilizar los comandos definidos en la primera entrega, para saber qué acción debe el servidor realizar. Para no poner únicamente un número, decidimos mapearlo a lo que corresponde ese **CommandConstant** a un mensaje del comando, así los logs, pueden ser autodescriptivos, de que acción se realizó en el server.

---

<sup>7</sup> Diagrama encontrado en Diagrams/LogModels.svg

## Proyecto Common

Se agregó un paquete Common, renombrando nuestro viejo Common a CommonProtocol, donde en este paquete se encuentran sólo dos clases, una interfaz, y SettingsManager. Este proyecto es muy estable, por lo que varios dependen de él para resolver este proceso de configuración desde un archivo XML en vez de hardcodear el valor.



Para esto en ServerAdmin se agregó también un archivo App.Config en formato XML para configurar los parámetros como el contexto y canal para utilizar GRPC entre ServerAdmin y Server.

Para ServerLog, se agregó un archivo similar, conteniendo la Key de Host, con valor localhost, en caso de que se quiere mudar a la nube en un futuro y solo se tenga que configurar el XML.

Para ConsoleClient, se decidió crear un archivo App.Config en formato XML donde se pueden configurar los parámetros de configuración con los que se ejecutan ambas aplicaciones.

---

<sup>8</sup> Diagrama encontrado en Diagrams/Common.svg



## Configuración

Estos son:

Server:

ServerIpAddress: La IP donde corre el server.

ServerPort: El puerto que escucha el server.

ServerPathToImageFolder: El directorio donde se guardaran las imágenes de todos los juegos.

Cliente:

ServerIpAddress: La IP del server a donde se conecta el cliente.

ServerPort: El puerto del server al que se conecta el cliente.

ClientPathToImageFolder: El directorio donde se guardaran las imágenes que recibe el cliente el servidor.

ServerLogs:

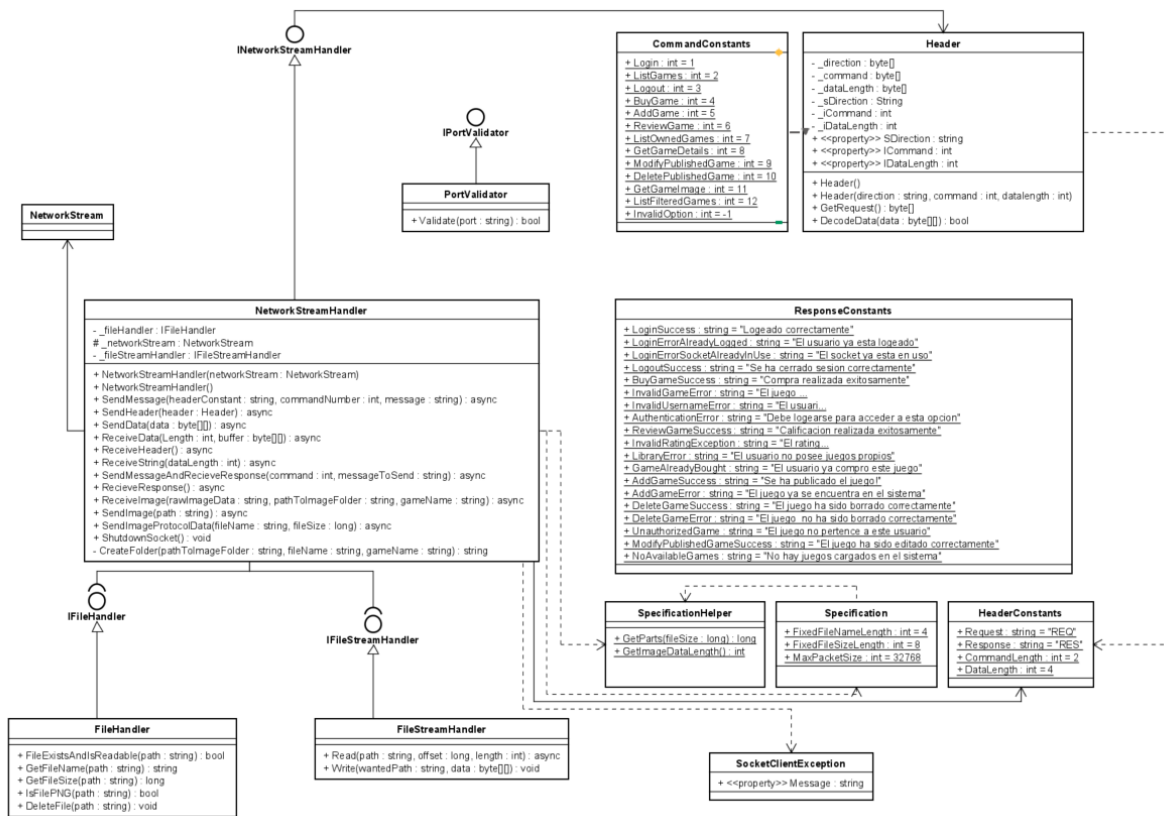
Host: Host en el cual reside la cola de mensajes a consumir.

Cliente:

ChannelPort: La IP del server a donde al servidor principal por GRPC, en este caso localhost:5004

AppContext: Que indica parámetros de configuración para utilizar esta tecnología.

# Proyecto CommonProtocol

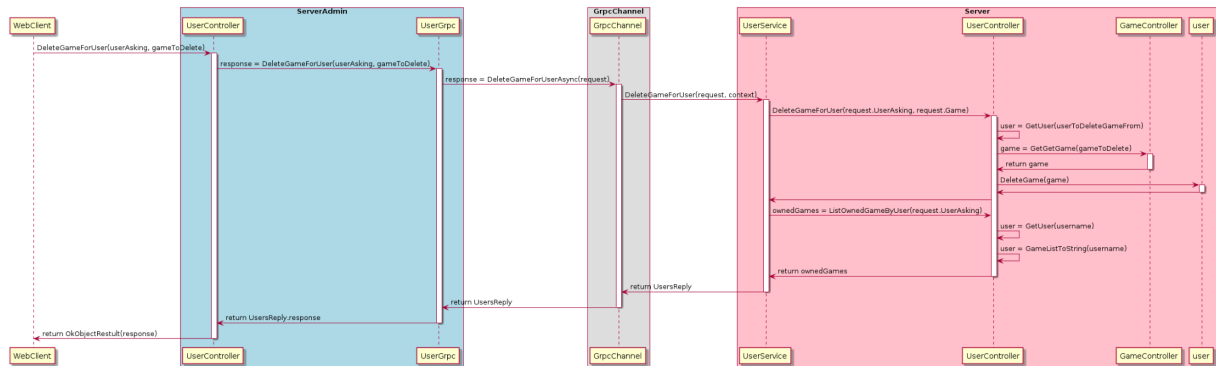


9

Se actualizó el proyecto anteriormente llamado Common a CommonProtocol, cambiando el uso de Sockets por NetworkStream, que fue implementado en la segunda entrega de este obligatorio.

<sup>9</sup> Diagrama encontrado en Diagrams/CommonProtocol.svg

## Diagramas de Interacción

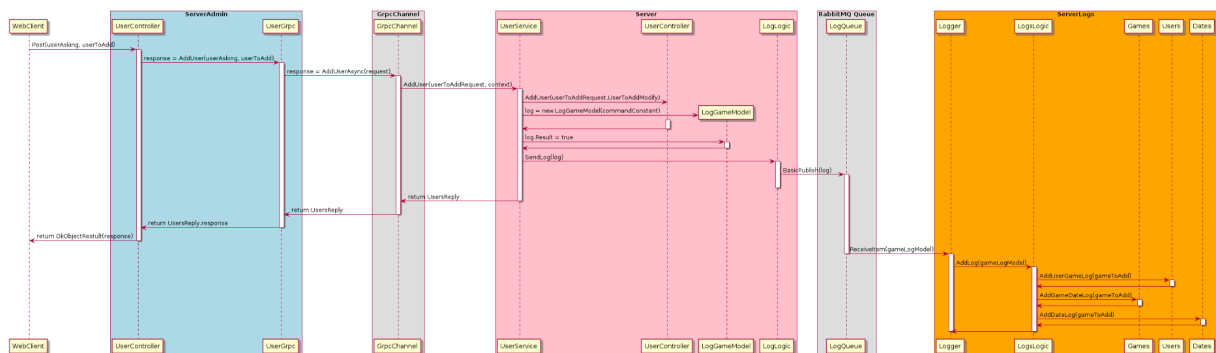


10

En este diagrama se busca reflejar las clases involucradas en el curso normal de eliminar un juego (sin el envío de logs, ya que mostraré en el siguiente) que un usuario había comprado previamente desde una request HTTP al server admin.

Para una mejor visualización del mismo se puede ver el archivo encontrado en la ubicación detallada en el pie de página.

Para mostrar cómo interactúan los proyectos involucrados en esta acción, se marcaron con diferentes colores.



11

En este diagrama se intenta mostrar cómo colaboran las clases para agregar un usuario desde el servidor administrativo, creando una request HTTP, luego comunicándose por GRPC con el servidor central y por último agregando el log a la cola de mensajes de forma asíncrona, para que después el servidor de logs pueda consumirla y guardarla.

<sup>10</sup> Diagrama encontrado en Diagrams/DeleteGameFromUser.svg

<sup>11</sup> Diagrama encontrado en Diagrams/AddUser.svg

## *Estándares de codificación*

Algunas de las decisiones de diseño ya fueron comentadas anteriormente, sin embargo queríamos resaltar otros puntos relevantes.

Al inicio del proyecto entendimos que era necesario establecer en grupo estándares de codificación para poder llegar a una solución clara y con buena calidad de código.

Algunos estándares establecidos fueron:

- El código y commits se escribe en inglés.
- La interfaz y mensajes a los usuarios se escriben en español.
- Los atributos privados de las clases empiezan con ‘\_’ Ej: `_nombreAtributo`.
- Los métodos, clases y atributos estáticos (sin importar su visibilidad) usan PascalCase.
- En el resto de casos se utiliza camelCase.
- No utilizar el “this.” para referirse a atributos al menos que sea:
  - Para evitar errores de compilación
  - Para ayudar a la comprensión del código
  - Para referirnos a una property adentro de una misma clase

## *Pruebas Postman*

Se realizaron pruebas exhaustivas en la herramienta Postman, probando todos los endpoints definidos en ServerAdmin relacionados a ABM de juegos y ABM de usuarios.

Se corrieron pruebas indicando errores conocidos y manejados, como también casos con resultado correcto, para luego en otra etapa poder corroborar con los Log generados.

También se agregó una colección con pruebas para los logs que registran todas las actividades realizadas por las pruebas mencionadas anteriormente en ServerAdmin. La idea es poder filtrar por distintos parámetros, como por ejemplo filtrar por un nombre de usuario, nombre de usuario y nombre de juego. Por fecha consideramos no filtrar dado que al no contar con una base de datos, los logs siempre pertenecen a un único día y no tiene mucho sentido utilizarlo de este modo. Sin embargo, queda extensible si se decide implementar una base de datos, y los logs quedan almacenados por mucho tiempo.

Las colecciones mencionadas se encuentran en la carpeta de Documentation/PostmanCollections.