

OBLIGATORIO III Programación de Redes 2021

Universidad ORT Uruguay
Profesor: Luis Barragué Martínez

Iñaki Etchegaray - 241072
Matías González - 219329

Índice

Abstract	3
Descripción del Sistema	4
Server de Logs	5
Overview	5
Diseño de Paquetes	6
Business	7
DataAccess	7
Database	8
Models y Domain	8
Server	9
RabbitMQ Service	10
Configuration y Factory	11
Diseño de la Web API	11
Server Admin	13
Overview	13
Diseño de Paquetes	13
Archivos Proto	14
Web API	14
API Model	14
Configuration	15
Diseño de la Web API	15
Cambios a Server Principal	16
Cambios al Protocolo	16
Cambios al Sistema	16
Cambios por Server de Logs	16
Cambios por Server Administrador	17
Anexo	18
Anexo 1: Especificaciones de APIs	18
API Servidor Logs	18
API ServerAdmin:	22
Game GRPC:	22
User GRPC:	27

Abstract

El objetivo de esta entrega es demostrar el conocimiento y uso de tecnologías de envío de información a través de una mejora del sistema de juegos VAPOR que se desarrolló en la entrega pasada.

Esta mejora se compone de dos sistemas nuevos que trabajaran en conjunto con el antiguo utilizando tres tecnologías nuevas: gRPC, MOM y Web API. La asignación de estas tecnologías a ciertas responsabilidades se explicará y fundamentará posteriormente.

En este documento, primero se detalla una descripción del sistema a grandes rasgos y luego iremos por sub sistema explicando las decisiones tomadas, el esquema general del diseño de cada una de ellas y finalmente cerraremos con el servidor principal, como estos nuevos sistemas impactaron en su diseño y funcionamiento.

Descripción del Sistema

Como se mencionó anteriormente, la solución ahora posee dos sistemas nuevos, un servidor de logs y un servidor administrador.

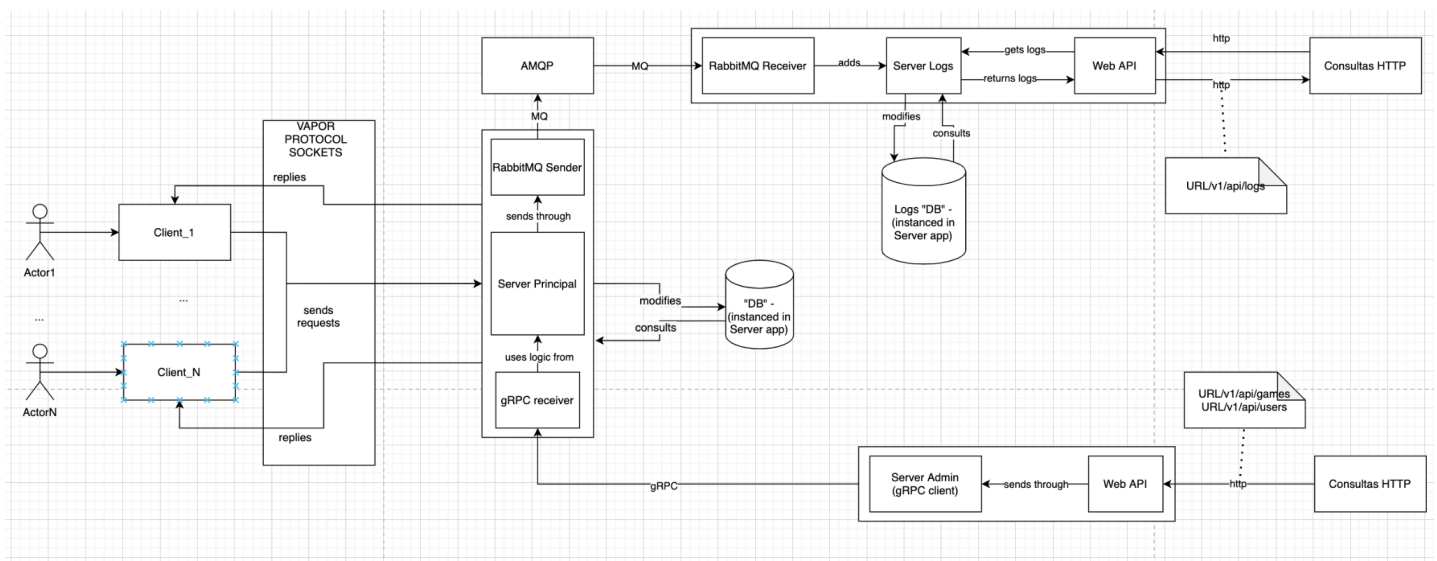
El primer sistema nuevo debe ser un servidor de logs, del cual el servidor antiguo es altamente participe. Toda acción de impacto sobre la base de datos, de errores, de login y logout por parte del servidor antiguo debe ser logueada por el servidor de logs. Para ello, decidimos utilizar la tecnología MOM de RabbitMQ para la comunicación entre el servidor principal y el servidor de logs. La justificación de esto se encuentra en la sección del servidor de logs.

El segundo sistema nuevo es el del servidor administrador. Este actúa como una especie de 'super-cliente' al poder realizar todas las funcionalidades de un cliente y aún más. Esto incluye el agregado de juegos, su manejo en la base de datos, el manejo de la vinculación de adquisición entre usuarios y juegos e incluso el agregado de usuarios, su modificación y su borrado. A diferencia de los clientes que solo podían crear sus usuarios y solo podrán adquirir juegos para ellos mismos.

En este segundo sistema decidimos usar gRPC para su comunicación con el servidor principal. La justificación de esto se encuentra en la sección del servidor de administración.

Finalmente, la tercera tecnología, Web API, estará presente en ambos sistemas como medio de interacción con los usuarios de esos servidores.

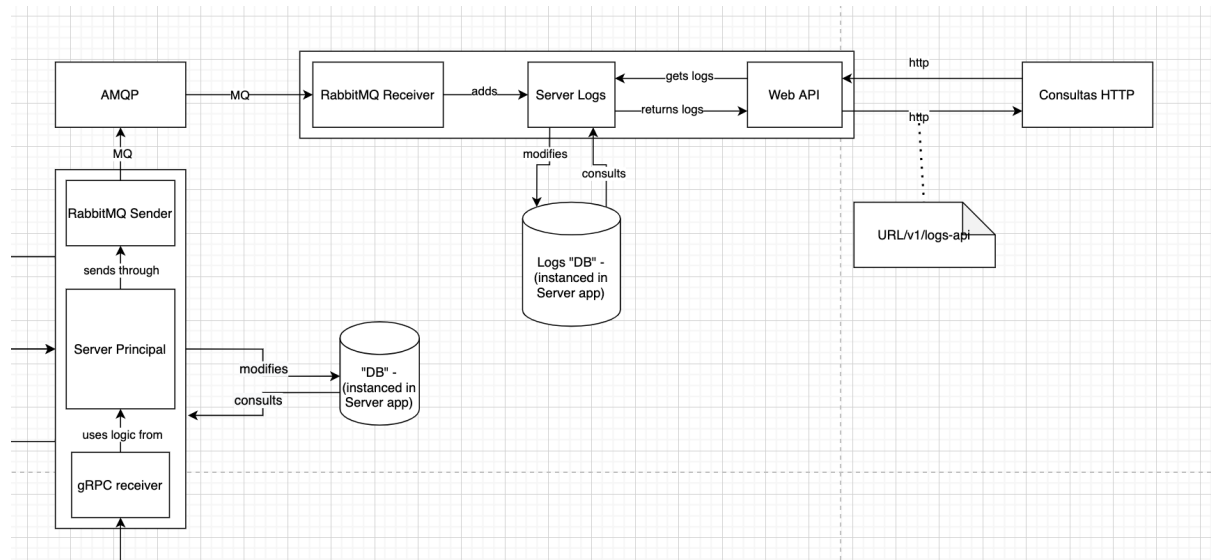
A continuación, se presenta el diagrama de la entrega anterior con los sistemas nuevos:



Server de Logs

Overview

Para poder hablar más fácilmente del sistema del servidor de logs, miremos a la sección del mismo en el diagrama presentado anteriormente:



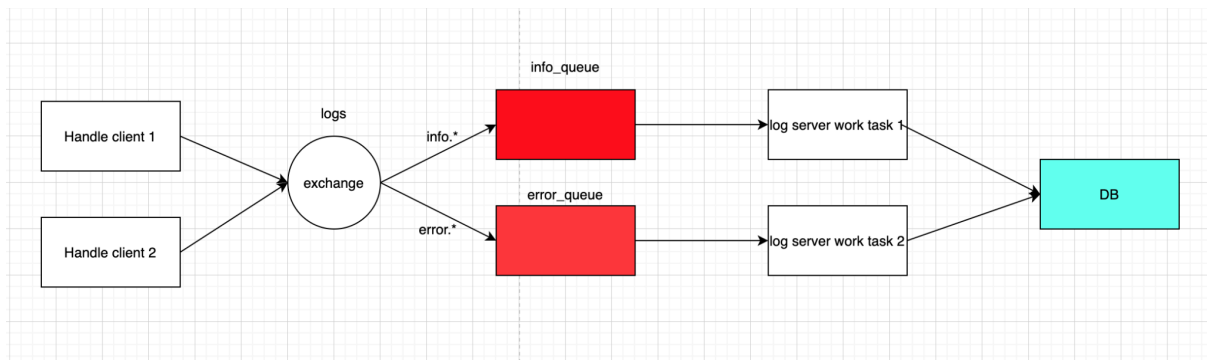
El servidor principal genera logs a partir de sus acciones, estas sean modificaciones en la base de datos (cualquier modificación), cualquier error que haya surgido en el servidor o cualquier login y logout que hagan los clientes.

Luego esos logs son enviados por el MOM RabbitMQ. Nosotros decidimos utilizar esta tecnología para los logs debido a que encaja mucho mejor para esta funcionalidad que una API o gRPC. Esto se debe a una de las principales ventajas de un MOM el cual es la asincronía total. Si el servidor de logs se cae, el servidor principal puede seguir actuando y agregando logs a la cola. Básicamente, el servidor principal no se debe preocupar por si el servidor de logs está activo, el solo envía información y se olvida. Luego, la próxima vez que se inicie todos los logs almacenados en la cola serán agregados en el orden adecuado. Esto encaja perfecto en un sistema de logs en donde la principal prioridad es la consistencia de las acciones sobre el tiempo, y que también debe de estar activo todo el tiempo. En contraste, un sistema gRPC o una API implican la necesidad de una sincronía que creemos innecesaria en este caso.

Al recibir los logs, estos son derivados a la lógica del servidor de logs, el cual los agrega a una base de datos en memoria.

Luego, usuarios externos pueden consultar a la base de datos de logs con una Web API. La misma trae los logs filtrados de acuerdo al pedido realizado, más sobre esto en la sección del diseño de la API. La API encaja bien en el consultado de logs ya que se requiere de una sincronía (descarta MOM), y debe ser flexible e intuitiva de usar para cualquier cliente y extensible para cualquier aplicación (descarta gRPC).

El modelado de las partes del programa en relación al MOM fue el siguiente:

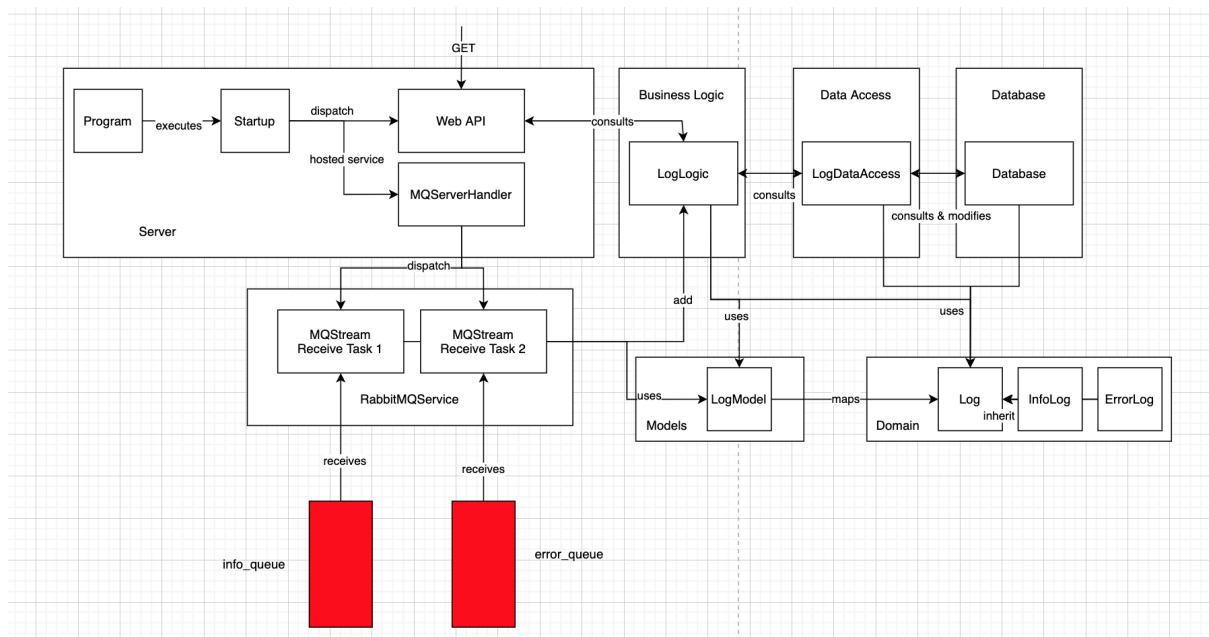


Se define un exchange que redirige a dos colas: info o error. La cola info está destinada a logs y mensajes que otorgan información. Estos incluyen cualquier tipo de modificación a la base datos, logueos o deslogueos de usuarios. La cola error está destinada a logs y mensajes que describen errores del servidor o de conexión. El servidor de logs maneja la entrada de información de cada una de estas colas con procesos asíncronos separados.

A continuación desplegamos el diseño de la aplicación.

Diseño de Paquetes

Para el diseño de la lógica del servidor de logs, decidimos utilizar un modelo de diseño en capas. Se definen cuatro capas: el servidor, la lógica de negocio, el data access y finalmente la base de datos, sumado de una capa paralela de servicios que posee el servicio de RabbitMQ. Finalmente, se encuentran los paquetes de Domain y Models, los cuales manejan los objetos del dominio utilizados por estos paquetes anteriormente mencionados. A continuación mostramos un diagrama que modela a grandes rasgos el flujo de la información en el servidor:



Business

El paquete Business se encarga de la lógica requerida para implementar las funcionalidades del servidor con respecto a la consulta y agregado de logs. Business expone la siguiente interfaz:

```
public interface ILogLogic
{
    bool Add(LogModel log);

    List<LogModel> Get(string username="", string gamename="", DateTime? date=null);
}
```

La interfaz ILogLogic actúa sobre la base de datos con el add, permitiendo el agregado de logs al sistema.

Luego el método Get se encarga de traer la lista de logs, estos vienen filtrados por nombre de usuario, nombre de juego y desde cierta fecha provista. Si no se incluye ningún parámetro, esto será lo mismo que traer todos los logs en el sistema. Los logs vienen ordenados por ID, es decir, por orden de inserción en el sistema.

DataAccess

El paquete de Data Access se encarga de consultar e impactar sobre la base de datos. Su impacto y consulta sobre ella debe ser sin aplicar lógica extra que no sea relacionada a agregar o a conseguir. La interfaz expuesta sobre el recurso de logs es la siguiente:

```
8 references | You, 2 weeks ago | 1 author (You)
public interface ILogDataAccess<T> where T : Log
{
    2 references
    bool Add(Log log);
    2 references
    List<Log> GetAll();
    2 references
    List<Log> Get(string username);
    4 references
    List<Log> Get(string username, string gamename);
}
```

La función add agrega un log a la base de datos tal cual es recibido. La función GetAll trae todos los logs del sistema pertinentes.

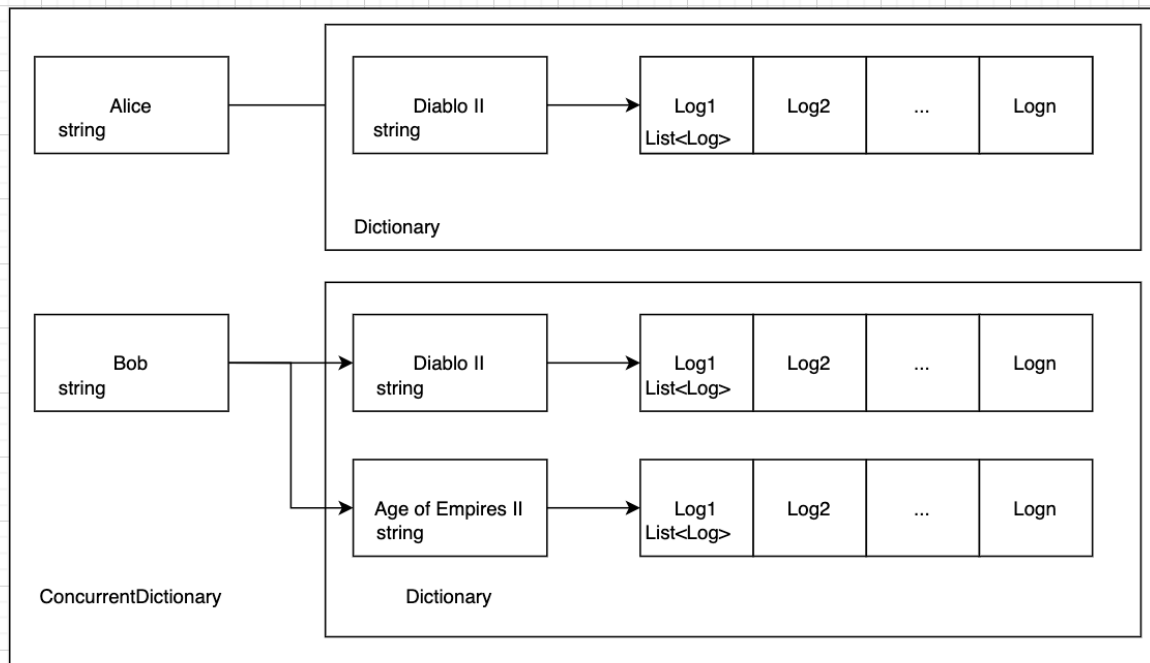
La función get tiene dos firmas posibles. El Get con nombre de usuario te trae aquellos logs pertinentes a ese usuario. El Get con nombre de usuario y nombre de juego te trae aquellos logs pertinentes a ese usuario junto con ese juego. Si en el segundo Get, omitimos el nombre de usuario, este nos traerá todos los logs pertinentes a ese juego.

Entendemos que estas implementaciones de Get podrían encajar más en Business, pero decidimos no poner esa lógica allí ya que esto implicaría que la business logic sería consciente de cómo la base de datos guarda la información. Para evitar esto, decidimos pasar esa funcionalidad a la capa de Data Access.

Database

Este paquete se encarga de implementar la base de datos en memoria no persistente. Su responsabilidad es simple, al solo ser un contenedor de información en modo singleton.

Una decisión grande tomada en la realización de este paquete es como decidimos guardar los logs. Para facilitarnos el trabajo de filtrar los logs en base a juegos y usuarios, nosotros decidimos utilizar dos diccionarios, uno destinado a logs de información y otro destinado a logs de errores. Estos diccionarios se componen de la siguiente manera:



El diccionario exterior es un **ConcurrentDictionary**, este es el mecanismo de concurrencia que decidimos utilizar y que nos mantiene la mutua exclusión de la base de datos entera. El mismo se maneja como llave el nombre de usuario. Luego el diccionario interior maneja de llave el nombre del juego y que lleva a una lista de logs que incluyen todos los logs que involucren a ese usuario interactuando con ese juego.

Tener un diccionario que compone listas separadas para cada usuario y para cada juego significa que cada usuario va a poder acceder a sus logs o a los logs del juego pertinente (la operación más frecuente y más cara en otro caso) de manera mucho más eficiente y rápida. Además, facilita la codificación de obtención de logs pertinentes a cierto juego y usuario o solo cierto usuario.

Sin embargo, esto también significa un manejo más complejo de agregado a la base de datos y una obtención más compleja cuando queremos obtener los logs de un juego solo o todos los logs del sistema.

Models y Domain

El paquete **Domain** se encarga de definir el objeto principal del dominio de la aplicación: los **Logs**.


```

public class Log
{
    6 references
    public int Id {get;set;}
    6 references
    public string Username {get; set;}
    12 references
    public string Gamename {get;set;}
    2 references
    public string Description {get;set;}
    3 references
    public DateTime Date {get;set;}
}

```

Además, los objetos LogError y LogInfo ambos heredan de Log. Esta distinción es lo que los diferencia entre ser un log de error y un log de info obviamente.

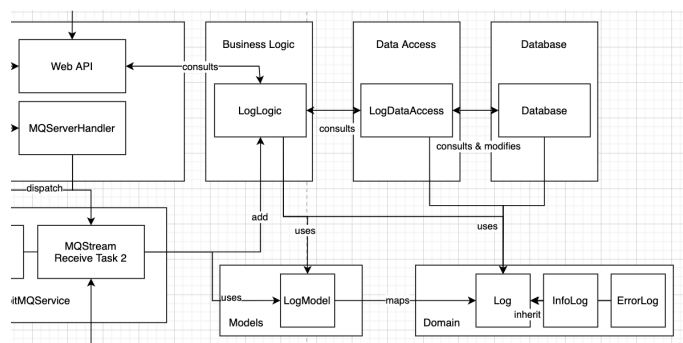
Luego el paquete de modelos se encarga de proveer un nivel de indirección entre el dominio y los objetos que interactúan con los puntos de acceso de la aplicación (es decir, la WebAPI o RabbitMQ). Por lo tanto existe la clase LogModel:

```

public class LogModel
{
    3 references
    public int Id {get;set;}
    6 references
    public LogType LogType {get;set;}
    3 references
    public string Username {get; set;}
    3 references
    public string Gamename {get;set;}
    3 references
    public string Description {get;set;}
    3 references
    public DateTime Date {get;set;}
}

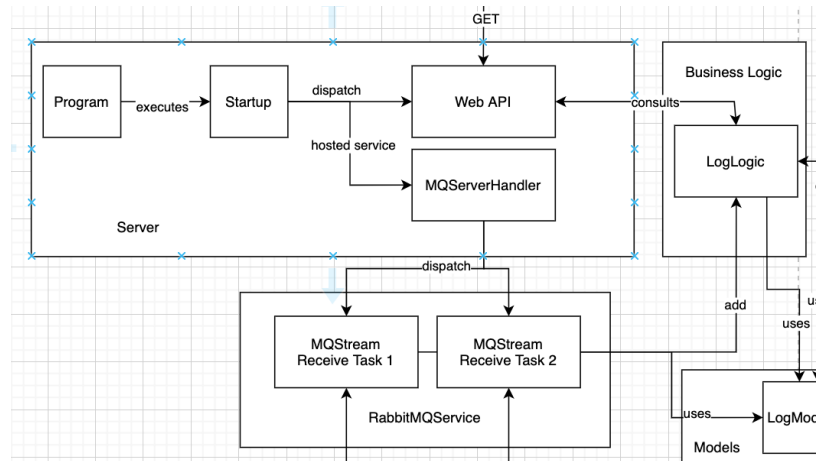
```

Luego estos son utilizados por distintos paquetes, dependiendo de qué tan cerca están a la base de datos:



Server

El paquete de servidor es el paquete que actúa como punto de acceso a la lógica de negocio. Es el paquete que inicia la aplicación y que setea todos los procesos para que funcione el sistema. Además, es el paquete que integra los controladores de la Web API, recibiendo las llamadas y las delega a la lógica de negocio.



La clase Program inicia por main a la clase Startup, esta realiza todas las configuraciones necesarias para que funcione la API y realiza la inyección de dependencias del resto de las funcionalidades. Por último, inicia MQServerHandler como servicio hosteado.

Web API representa a todos los controladores iniciados que se encuentran en la escucha constante por llamadas HTTP. Esta API es muy chica ya que compone un solo controlador con un solo endpoint, más sobre esto en la sección de diseño de Web API. El controlador recibe las llamadas HTTP y delega a la lógica de negocio.

MQServerHandler inicializa el servicio de RabbitMQ instanciando las clases. Luego inicia las tareas asíncronas de escucha de RabbitMQService. Además, es el que define qué acción debe de hacer la recepción de rabbit, en donde este delega a la lógica de negocio en agregar los logs:

```

await _streamControl.ReceiveAsync<LogModel>(
    _configurationHandler.GetField(ConfigurationConstants.INFO_QUEUE_NAME_KEY),
    _configurationHandler.GetField(ConfigurationConstants.INFO_ROUTING_KEY_KEY),
    x => Task.Run(() => { _logLogic.Add(x); }, stoppingToken)
);

await _streamControl.ReceiveAsync<LogModel>(
    _configurationHandler.GetField(ConfigurationConstants.ERROR_QUEUE_NAME_KEY),
    _configurationHandler.GetField(ConfigurationConstants.ERROR_ROUTING_KEY_KEY),
    x => Task.Run(() => { _logLogic.Add(x); }, stoppingToken)
);

```

RabbitMQ Service

Este paquete es el primero en ser compartido por dos soluciones distintas. RabbitMQ Service se encarga de implementar la lógica relacionada a la comunicación con las colas de RabbitMQ. Este define la siguiente interfaz:

```

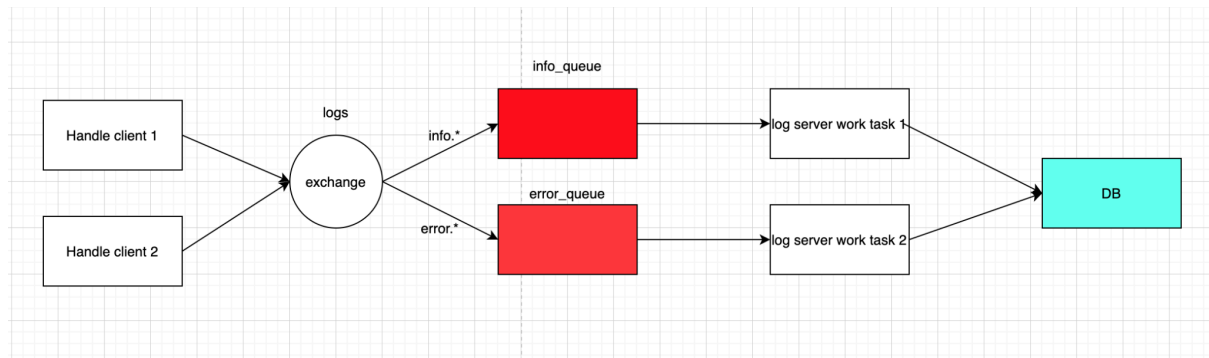
public interface IMQStream
{
    // Matias González, 2 weeks ago • LogsServer base implemented. Added existing Co
    0 references
    Task SendAsync<T>(string queueName, string routingKey, T toSend);

    2 references
    Task ReceiveAsync<T>(string queueName, string routingKey, Action<T> onReceive);
}

```

La interfaz implementa el envío y recepción asincrónica de mensajes con rabbit. Por lo tanto, al componer la funcionalidad de envío y recepción, decidimos hacer que este paquete fuera compartido por ambos el servidor de logs y el servidor principal.

Los métodos de la clase concreta a IMQStream definen el esquema mencionado anteriormente:



La función ReceiveAsync se encarga de setear las colas, el exchange y bindear estas colas a sus routing keys correspondientes (info.* y error.*). Luego realiza la suscripción al evento de recepción a una función que usa el Action on receive y deserializa la información.

Luego la función SendAsync solo se encarga de enviar información por un exchange declarado y una cola declarada.

El servidor de logs, naturalmente, hace uso de ReceiveAsync, mientras que el servidor principal hace uso de SendAsync.

Configuration y Factory

Los últimos paquetes por mencionar son Configuration y Factory. Los dos no aparecen en el diagrama ya que no nos pareció pertinente a su composición.

Configuration es el paquete compartido del servidor principal que le permite al servidor de logs poder acceder a su app.config.

Factory es el paquete encargado de realizar la inyección de dependencias de las clases del sistema. No posee otra responsabilidad que esa.

Diseño de la Web API

La Web API de este servidor es una API pequeña, está compuesta de un solo endpoint que compone funcionalidad que posee varios resultados posibles.

Decidimos utilizar de raíz de endpoint el siguiente string:
"v1/api/logs"

Creemos que este string sigue los principios de un buen diseño de api al componer una raíz "v1" en caso de que surjan versiones futuras. Luego definiendo api para hacerse

entender qué tipo de recurso es de HTTP (no es una página necesariamente). Finalmente, logs representa el recurso de la api accedida, utilizando sustantivos plurales y evitando verbos.

Utilizamos Postman para las pruebas a la API y además para realizar la documentación de la misma.

En el [Anexo 1](#) se encuentra la documentación exhaustiva de la API del servidor de Logs.

Server Admin

Overview

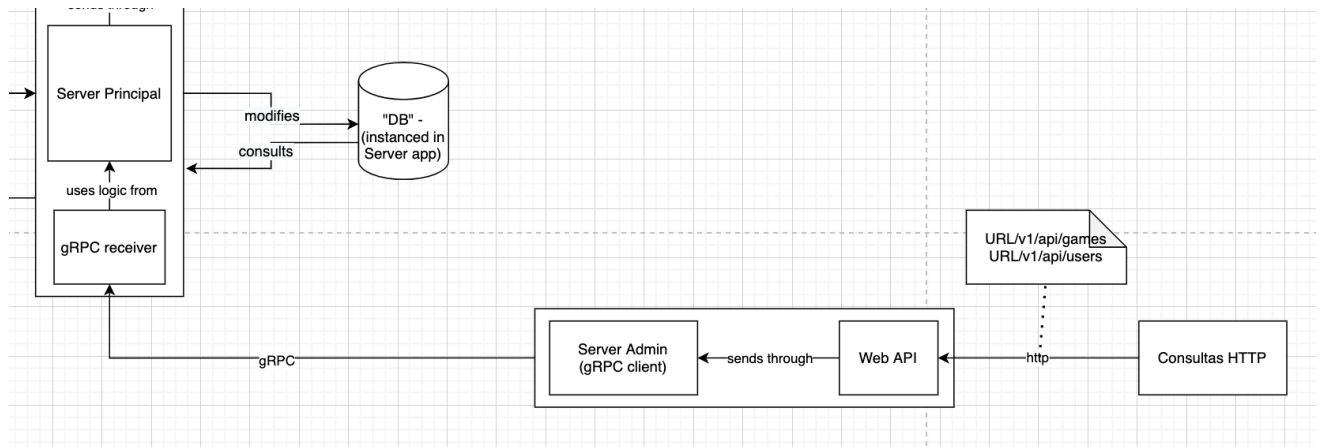
Para este server decidimos utilizar la tecnologia de GRPC ya que su facilidad a la hora de llamar funciones síncronas desde un server hacia el otro como si fuesen un metodo dentro de la misma solución nos resultó muy fácil de utilizar.

Elegimos gRPC por sobre MOM ya que nos parecía ventajoso el tener una conexion sincronica entre los servidores ya que el server admin debe ser consciente de sí en el servidor principal ocurre un error. Por otro lado, descartamos WebAPI ya que creemos estar en un ambiente cerrado donde ambos servidores se conocen y no existe la necesidad de exponer una interfaz de público acceso.

Por como fue diseñado nuestro servidor principal en un principio el añadir estos nuevos comandos a través de gRPC fue sumamente sencillo.

Este server funciona similar a un cliente ya que le envía comandos al servidor principal, los cuales este luego ejecuta impactando sobre la base de datos. A diferencia de un cliente, este server administrativo recibe el comando a ejecutar y toda la información necesaria mediante consultas a una WebAPI, luego esto se envía al servidor principal a traves de gRPC y este se encarga de impactar esos cambios en la DB utilizando la business Logic al igual que ocurre con los clientes.

Esto se puede ver claramente en el siguiente diagrama:



Las funciones principales de este server son poder agregar, modificar y eliminar tanto juegos como usuarios del sistema, además de poder manejar la adquisición de juegos de los usuarios, pudiendo agregar o quitar juegos de los pertenecientes a cada usuario.

Diseño de Paquetes

El diseño de paquetes en el servidor administrativo resultó ser más simple que en los servidores anteriores ya que el mismo simplemente necesitaba los endpoints para la webAPI y luego la conexión gRPC a al servidor principal con lo que esto implica. Esto entonces llevó a una división en tres paquetes, primero un paquete principal WebAPI y luego dos paquetes de soporte, API Model y Configuration.

Archivos Proto

Para la comunicación con el servidor principal gRPC utiliza archivos .proto. Ya que estos son llamados desde los endpoints al recibir la consulta en la WebAPI creímos correcto tenerlos en ese paquete y no en un paquete aparte. En estos archivos se define la estructura de la información a ser enviada y lo que se espera recibir a través de los mensajeros.

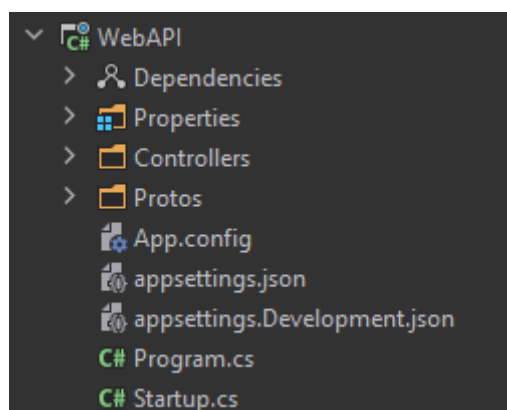
Implementamos entonces a través de estos archivos dos servicios GameMessenger y UserMessenger los cuales contienen las llamadas sincrónicas que llegan luego al servidor principal donde luego se ejecutan las funcionalidades más adelante explicadas en el mismo.

```
service GameMessenger {  
    rpc AddGame (AddGameRequest) returns (GameReply);  
    rpc ModifyGame (ModifyGameRequest) returns (GameReply);  
    rpc DeleteGame (DeleteGameRequest) returns (GameReply);  
    rpc LinkUserGame (LinkUserGameRequest) returns (GameReply);  
    rpc UnlinkUserGame (UnlinkUserGameRequest) returns (GameReply);  
}
```

```
service UserMessenger {  
    rpc AddUser (AddUserRequest) returns (UserReply);  
    rpc ModifyUser (ModifyUserRequest) returns (UserReply);  
    rpc DeleteUser (DeleteUserRequest) returns (UserReply);  
}
```

Web API






Además de contener los archivos del protocolo buffer este paquete contiene los controllers para los endpoints de la WebAPI. Por último un App.config donde se guarda la ip del servidor principal.



API Model

API Model contiene las estructuras de los datos que se esperan recibir de las consultas a la API. Este paquete existe para separar los modelos de objetos del código principal, en este caso se necesitaron 3 modelos. Los modelos de User y Game son copias de los objetos del Dominio del servidor principal, el modelo UserGameLink en cambio es un

modelo de ayuda para manejar la adquisición de juegos. En este caso particular UserModel y UserGameLink contienen los mismos atributos pero se los separó en dos modelos de todas formas ya que está igualdad es una mera casualidad debido a la poca cantidad de atributos en los usuarios del sistema actual pero en caso de aumentar la cantidad de información en un usuario esto ya no sería igual.

```
public class GameModel
{
     2 usages
    public string Gamename {get; set;}
     2 usages
    public string Genre {get; set;}
     2 usages
    public string ESRB {get; set;}
     2 usages
    public string Synopsis {get; set;}
     2 usages
    public string PathToImage {get; set;}
}
```

Configuration

El paquete configuration es simplemente una referencia al paquete Configuration dentro del servidor principal. El mismo se utiliza simplemente para acceder al app.Config y leer los paths a las imágenes.

Diseño de la Web API

Como se mencionó anteriormente este servidor administrativo constaba de tres funcionalidades. Primero poder agregar, modificar y eliminar usuarios, juegos, y poder tanto asociar como desasociar juegos de un usuario. Entonces decidimos dividir la WebAPI en dos endpoints, /games y /users, ambos usando la raíz 'v1/api'. La función de asociación se la asignó al endpoint de games ya que sabiendo como funciona la lógica del servidor principal y que quien se encarga de asociar los juegos es la GameLogic creimos mas coherente mantener esta responsabilidad en los juegos. Para estos casos de ruta se definió /user-acquire/gameid.

En el [Anexo 1](#) se encuentra la documentación exhaustiva de la API del servidor Admin.

Cambios a Server Principal

Cambios al Protocolo

No hubo ningún cambio considerable al protocolo entre el Servidor principal y los clientes.

Cambios al Sistema

Los cambios realizados en el servidor principal radican por la inclusión de los nuevos sistemas. A continuación mencionamos los cambios radicando de cada sistema en particular.

Cambios por Server de Logs

Se agregaron dos nuevos paquetes: RabbitMQService y LogsCommunicator. RabbitMQService es el mismo paquete mencionado en la sección del servidor de logs. LogsCommunicator es el paquete encargado de confeccionar logs y enviarlos utilizando el LogsCommunicator.

LogsCommunicator expone la interfaz de envío de logs:

```
31 references | You, a week ago | 1 author (You)
public interface ILogSender
{
    11 references
    Task SendLog(LogModel log);
}
```

Nota: LogModel es un LogModel distinto al del servidor de logs. Representan lo mismo, pero son dos clases distintas en paquetes distintos.

SendLog simplemente hace uso del SendAsync mencionado en RabbitMQService y envía la información a la cola en el broker de AMQP.

Luego el envío de logs se implementó en los lugares en donde hayan errores de servidor (no de negocio) y en la ejecución de cada uno de los comandos si corresponde. A modo de ejemplo acá hay una pieza del código en el comando adquirir juego:

```
gameLogic.AcquireGame(query);

string logMessage = $"The user {query.Username} has acquired the game {gameLogic.GetGame(query.Gameid).Title}.";
SendLog(query.Username, query.Gameid, logMessage);
```

Otro cambio realizado es la separación de la funcionalidad de configuración del paquete de Common a su paquete único separado. Esto fue para poder utilizar la funcionalidad en otras soluciones sin tener que traer otros paquetes no necesarios.

Cambios por Server Administrador

Por el lado del servidor administrador, el servidor principal se afectó en un gran cambio principal: la inclusión del receptor de llamadas gRPC.

Para implementar este cambio, decidimos hacer un proyecto nuevo con esquema de uso gRPC y pasar todo el código del paquete Server a este paquete nuevo. Luego nos deshicimos de la consola tradicional del servidor principal antiguo, ya que nos parecía innecesaria en esta versión nueva de la aplicación.

Luego el paquete de Server implementa en la carpeta de Services las clases de los servicios generadas por la ejecución de los archivos proto mencionados anteriormente. En estos servicios, se implementa en cada una de las funciones el código que se ejecuta al hacer la llamada a la función RPC en el servidor administrador.

En base a esto, en estas llamadas RPC se implementan las funcionalidades esperadas de la especificación. Para ellas, reutilizamos la lógica implementada en el obligatorio anterior en el paquete de Business Logic.

Sin embargo, no todas las funcionalidades ya estaban implementadas, algunas eran nuevas. En específico: la modificación de usuarios, el borrado de usuarios y el de adquirir de un juego a un usuario. La primera y segunda fueron implementadas en la clase UserLogic y la última en la clase GameLogic, ambas localizadas en el paquete Business Logic.

Anexo

Anexo 1: Especificaciones de APIs

API Servidor Logs

GET

GET System Logs

Open Request →

{{Logs-URL}}/v1/api/logs?username=ExampleUser&gamename=ExampleGame&date=2000-09-15

Summary

Gets the main servers system logs.

Result:

Brings logs filtered by user involved, game involved and from a specified date. Returns all logs if no parameters given.

Parameters

Username: The user's name involved in the logs.
Gamename: The game's name involved in the logs.
Date: All logs received will be from this date.

Responses:

200: If everything went right.

Request Params

username	ExampleUser
gamename	ExampleGame
date	2000-09-15

Example

No inputs example ▾

Request

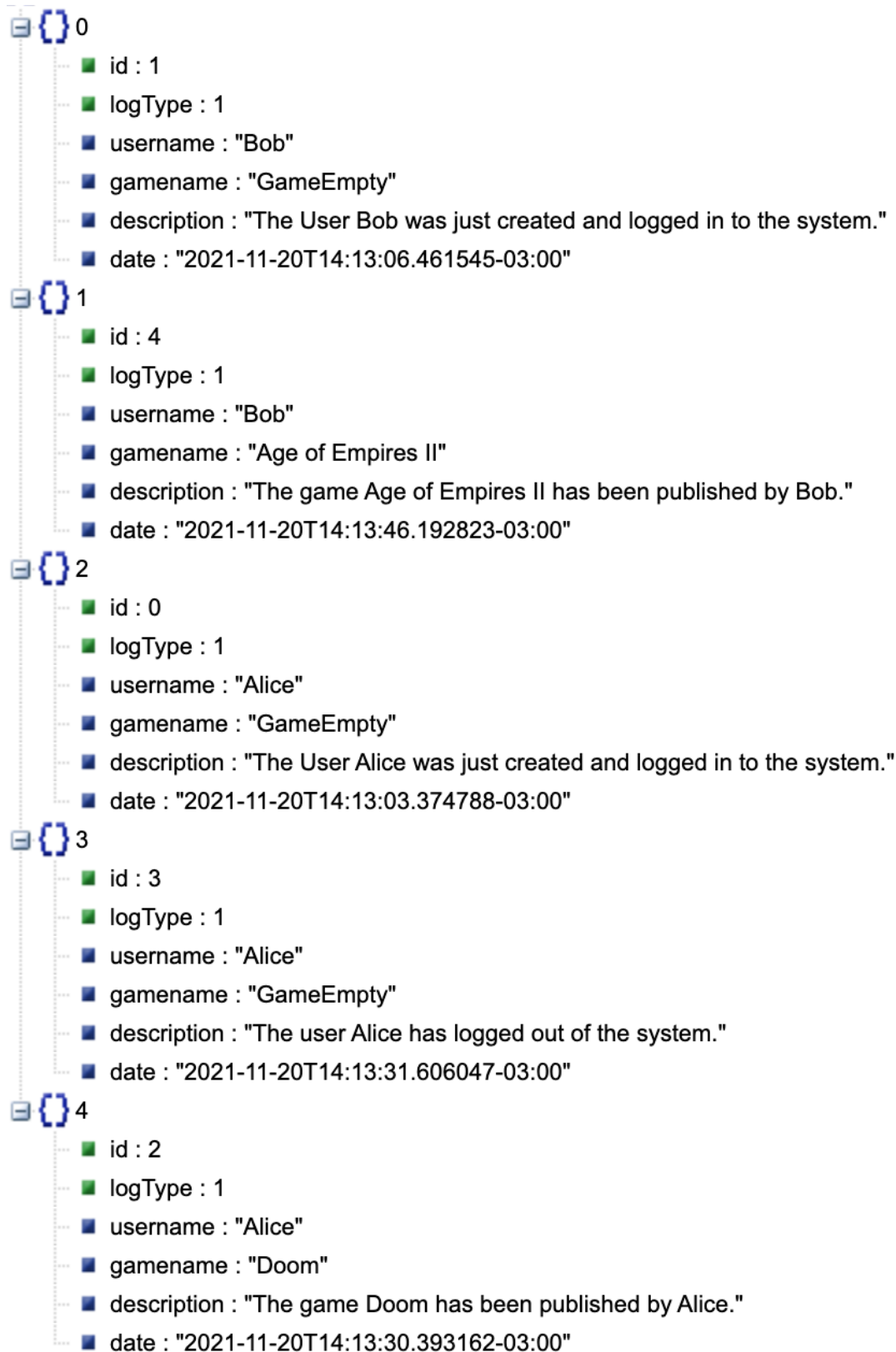
cURL

curl --location -g --request GET '{{Logs-URL}}/v1/api/logs'

Response

Body

Headers



Example

Username input example ▾

Request

cURL



```
curl --location -g --request GET '{{Logs-URL}}/v1/api/logs?username=Bob'
```

Response

Body Headers

json



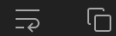
```
[
  {
    "id": 1,
    "logType": 1,
    "username": "Bob",
    "gamename": "GameEmpty",
    "description": "The User Bob was just created and logged in to the system.",
    "date": "2021-11-20T14:13:06.461545-03:00"
  },
  {
    "id": 4,
    "logType": 1,
    "username": "Bob",
    "gamename": "Age of Empires II",
    "description": "The game Age of Empires II has been published by Bob.",
    "date": "2021-11-20T14:13:46.192823-03:00"
  }
]
```

Example

Gamename input example ▾

Request

cURL



```
curl --location -g --request GET '{{Logs-URL}}/v1/api/logs?gamename=Doom'
```

Response

Body Headers

json



```
[
  {
    "id": 2,
    "logType": 1,
    "username": "Alice",
    "gamename": "Doom",
    "description": "The game Doom has been published by Alice.",
    "date": "2021-11-20T14:13:30.393162-03:00"
  }
]
```

Example

All inputs example ▾

Request

HTTP



GET /v1/api/logs?gamename=Age of Empires II&username=Bob&Date=2022-11-20T12:00:00 HTTP/1.1
Host: {{Logs-URL}}

[View more](#)

Response

Body Headers

json



```
[ ]
```

API ServerAdmin:

Game GRPC:

POST **POST Game** [Open Request →](#)

`{{AdminServer-URL}}/v1/api/games`

Summary:
Adds new game to the main server's database.

Result:
Brings response with status code and log message informing if the action was done successfully or not.

Body raw (json)

json

```
{
  "gamename": "Doom",
  "genre": "Shooter",
  "esrb": "M+",
  "synopsis": "Become Doomguy as you slaughter hordes of demons with you shotgun and chainsaw",
  "PathToImage": "/Users/sleepy/Desktop/Screen Shot 2021-10-31 at 10.09.30 AM.png"
}
```

View more

Example

POST Game correct ▾

Request

cURL



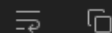
```
curl --location -g --request POST '{{AdminServer-URL}}/v1/api/games' \
--data-raw '{
  "gamename": "Doom",
  "genre": "Shooter",
  "esrb": "M+",
  "synopsis": "Become Doomguy as you slaughter hordes of demons with you shotgun and cha:
  "PathToImage": "/Users/sleepy/Desktop/Screen Shot 2021-10-31 at 10.09.30 AM.png"
}'
```

View more

Response

Body Headers

json



```
{
  "statusCode": 20,
  "message": "The game Doom has been published by ADMIN."
}
```

POST POST Acquire Game

Open Request →

{{AdminServer-URL}}/v1/api/games/user-acquire/0

Summary:

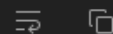
Adds a game to the acquired games list of a the user specified.

Result:

Brings response with status code and log message informing if the action was done successfully or not.

Body raw (json)

json



```
{
  "username": "Alice"
}
```

Example

POST Alice acquire Doom

Request

cURL



```
curl --location -g --request POST '{{AdminServer-URL}}/v1/api/games/user-acquire/0' \
--data-raw '{
  "username": "Alice"
}'
```

Response

Body Headers

json



```
{
  "statusCode": 20,
  "message": "The user Alice has acquired the game Doom. Granted by ADMIN."
}
```

PUT PUT Game

[Open Request](#) →

{{AdminServer-URL}}/v1/api/games/0

Summary:

Modifies an existing game in the main server's database.

Result:

Brings response with status code and log message informing if the action was done successfully or not.

Body raw (json)

json



```
{
  "Gamename": "Diablo",
  "Genre": "Casual",
  "ESRB": "",
  "Synopsis": "Diablo",
  "PathToImage": "/Users/sleepy/Desktop/Screen Shot 2021-10-31 at 10.09.30 AM.png"
}
```


Example

PUT Doom -> Diablo

Request

cURL



```
curl --location -g --request PUT '{{AdminServer-URL}}/v1/api/games/0' \
--data-raw '{
  "Gamename": "Diablo",
  "Genre": "Casual",
  "ESRB": "",
  "Synopsis": "Diablo",
  "PathToImage": "/Users/sleepy/Desktop/Screen Shot 2021-10-31 at 10.09.30 AM.png"
}'
```

Response

Body Headers

json



```
{
  "statusCode": 20,
  "message": "The game Diablo has been modified by ADMIN."
}
```

DEL DELETE Acquire Game

[Open Request](#) →

{{AdminServer-URL}}/v1/api/games/user-acquire/0

Summary:

Removes a game from the acquired games list in a user.

Result:

Brings response with status code and log message informing if the action was done successfully or not.

Body raw (json)

json



```
{
  "username": "Alice"
}
```

Example

DELETE Alice unacquire Diablo

Request

cURL



```
curl --location -g --request DELETE '{{AdminServer-URL}}/v1/api/games/user-acquire/0' \
--data-raw '{
  "username": "Alice"
}'
```

Response

Body Headers

json



```
{
  "statusCode": 20,
  "message": "The user Alice no longer owns Diablo, removed by ADMIN."
}
```

DEL DELETE Game

[Open Request](#) →

{{AdminServer-URL}}/v1/api/games/0

Summary:

Deletes a game from the main server's database.

Result:

Brings response with status code and log message informing if the action was done successfully or not.

ExampleDELETE Doom

Request

cURL



```
curl --location -g --request DELETE '{{AdminServer-URL}}/v1/api/games/0'
```

Response

BodyHeaders

json



```
{  
  "statusCode": 20,  
  "message": "The game Doom has been deleted by ADMIN."  
}
```

User GRPC:

POST POST UserOpen Request →



```
{{AdminServer-URL}}/v1/api/users
```

Summary:
Adds a new user to the main server's database.

Result:
Brings response with status code and log message informing if the action was done successfully or not.

Body raw (json)

json



```
{  
  "Username": "Alice"  
}
```

Example

POST Alice ▾

Request

cURL



```
curl --location -g --request POST '{{AdminServer-URL}}/v1/api/users' \
--data-raw '{
  "Username": "Alice"
}'
```

Response

Body Headers

json



```
{
  "statusCode": 20,
  "message": "The user Alice has been added."
}
```

PUT PUT User

Open Request →

{{AdminServer-URL}}/v1/api/users/0

Summary:

Modifies an existing user in the main server's database.

Result:

Brings response with status code and log message informing if the action was done successfully or not.

Body raw (json)

json



```
{
  "Username": "Miguel"
}
```

Example

PUT Alice -> Miguel

Request

cURL



```
curl --location -g --request PUT '{{AdminServer-URL}}/v1/api/users/0' \
--data-raw '{
  "Username": "Miguel"
}'
```

Response

Body Headers

json



```
{
  "statusCode": 20,
  "message": "The user with id 0 has been modified to Miguel."
}
```

DEL DELETE User

[Open Request](#) →

{{AdminServer-URL}}/v1/api/users/0

Summary:

Deletes a user in the main server's database.

Result:

Brings response with status code and log message informing if the action was done successfully or not.

Example

DELETE Miguel

Request

cURL



```
curl --location -g --request DELETE '{{AdminServer-URL}}/v1/api/users/0'
```

Response

Body Headers

json



```
{
  "statusCode": 20,
  "message": "The user Miguel has been deleted by ADMIN."
}
```