Universidad ORT Uruguay
Facultad de Ingeniería

Programación de redes

Sistema Vapor

Nicolás Gibbs - 227347
Germán Konopka - 238880
Martin Robatto - 240935

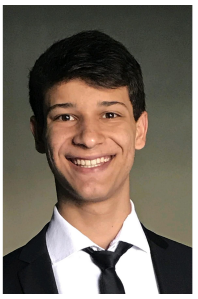
Tutores: Luis Barragué y Roberto Assandri

2021

Autoría

Nro. Estudiante	240935
Nombre:	Martín
Apellido:	Robatto
Grupo / Turno:	M6A

Nro. Estudiante	238880
Nombre:	German
Apellido:	Konopka
Grupo / Turno:	M6A



Nro. Estudiante	227347
Nombre:	Nicolás
Apellido:	Gibbs
Grupo / Turno:	M6A

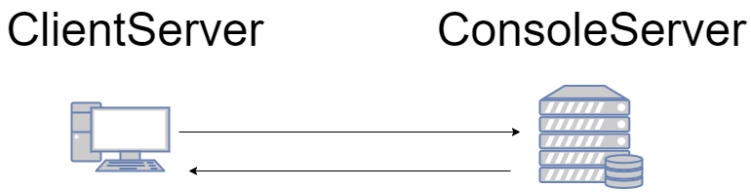
Índice

Descripción General

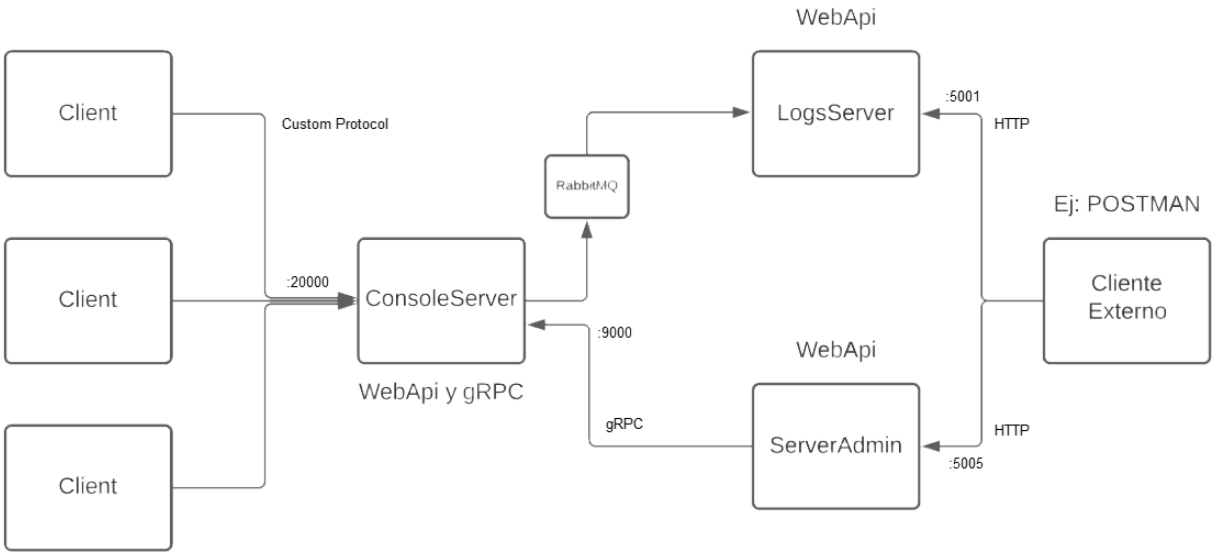
Nuestro trabajo consistió en la creación de un sistema basado en C# que consta de dos aplicaciones, cliente y servidor, que presentan funciones tales como gestión completa de juegos, ver catálogo y adquirir juegos, publicar reseñas, entre otras. La aplicación cliente se encarga de las interacciones de los usuarios con el sistema, mientras el servidor es quien atiende las peticiones de los clientes y administra los recursos del sistema relacionados a los servicios brindados.

Cabe destacar que en esta oportunidad no se cuenta con persistencia de datos mediante el uso de una base de datos, sino que se recurre al almacenamiento en memoria para los datos ingresados al sistema. En otras palabras, se busca comunicar que una vez cerrada la aplicación servidor, los datos ingresados se eliminaran.

Como se implementó en la primera entrega, se trata de una arquitectura cliente- servidor, donde el cliente realiza las peticiones que le correspondan y el servidor atiende las peticiones para los recursos que administra.



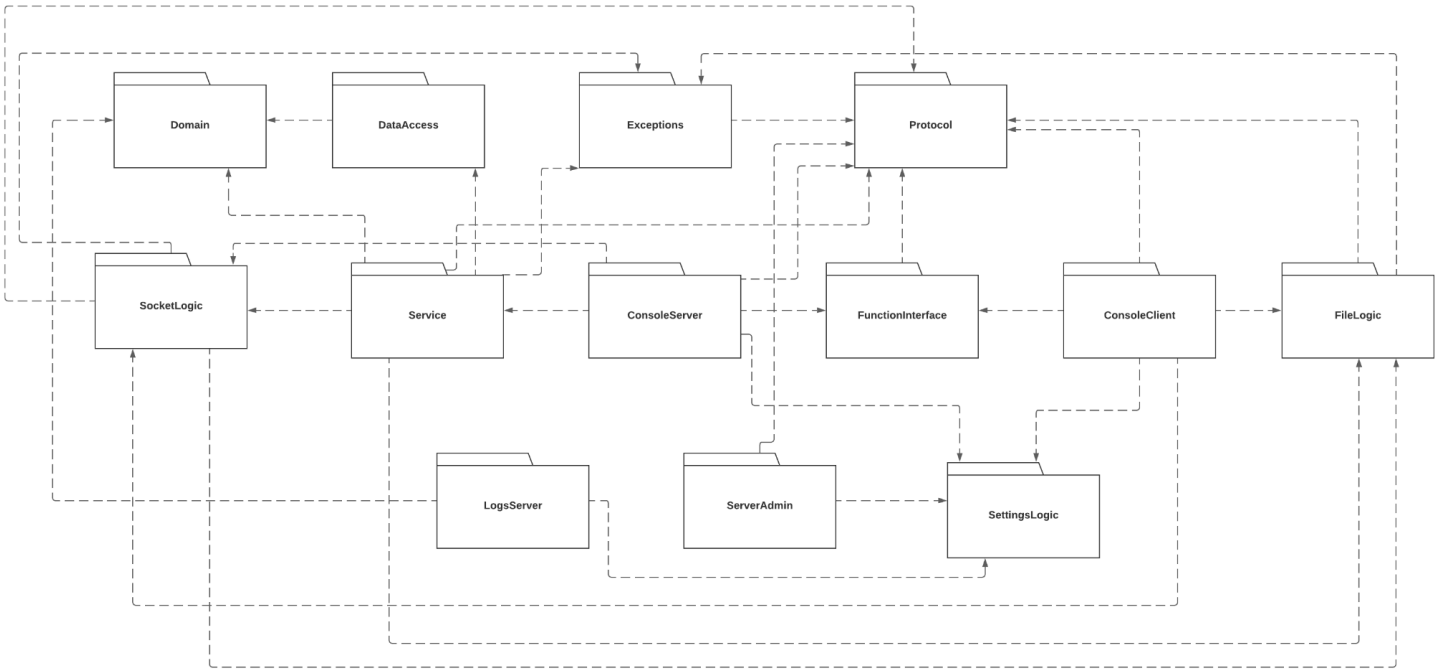
Además de la aplicación cliente-servidor, se implementaron dos nuevas tecnologías, como son MOM: Message Oriented Middlewares, mediante el uso de RabbitMQ, que permite la independencia de la comunicación con respecto al Tiempo para el registro de logs del sistema y gRPC para la interacción entre las aplicaciones Admin Server y Console Server que expone algunos servicios en específico.



Se establecieron dos web API (Application Programming Interface) como interfaz para la implementación de las nuevas tecnologías, las cuales, proporcionan un conjunto de endpoints que proveen acceso a las funcionalidades. Para definir estas interfaces e indicar la ejecución de operaciones sobre los datos, se utilizó el conjunto de principios de arquitectura REST (Representational State Transfer).

Diagrama de Paquetes

A continuación, se muestra la distribución general de los módulos que componen al sistema. Representa la vista general de la estructura del sistema y de toda la lógica para la interacción cliente-servidor. Cada uno de ellos representa un archivo .dll que estará en el directorio del sistema.



Breve descripción de responsabilidades

Nuestra solución está compuesta por varios proyectos que contienen un conjunto de clases organizadas según las responsabilidades que conlleven. Se describen todos los paquetes según su responsabilidad y las funcionalidades que presentan.

ServerAdmin: Contiene las clases Program y Startup, que son el punto de entrada de nuestra aplicación, donde configuramos el servidor web; y donde configuramos los componentes de la aplicación respectivamente.

ServerAdmin.Models: Contiene clases que actúan como modelos de entrada y salida (Data Transfer Objects) para la transferencia de objetos entre el Cliente y la Api.

ServerAdmin.Controllers: Contiene clases (Controllers) que definen endpoints que reciben las consultas HTTP implementadas (POST, DELETE y UPDATE) y delegar la resolución de la operación solicitada.

LogsServer: Contiene la WebApi que expone los servicios para el acceso y filtrado de logs del sistema. Almacena las clases Program y Startup, que son el punto de entrada de nuestra aplicación, donde configuramos el servidor web; y donde configuramos los componentes de la aplicación respectivamente. Tiene el rol de consumidor de mensajes en RabbitMQ.

LogsServer.Controllers: Contiene clases (Controllers) que definen endpoints que reciben las consultas HTTP implementadas (GET).

ConsoleClient: Contiene lo relacionado con la aplicación cliente. Tiene el display para imprimir en consola, el administrador de cliente ("ClientHandler") y por último el program con el método main.

ConsoleServer: Contiene lo relacionado con la aplicación servidor. Tiene el display para imprimir en consola, el administrador del servidor("ServerHandler") y por último el program con el método main. Para esta nueva versión, se le adjunto todas las dependencias necesarias para que exponga servicios gRPC y aunque conceptualmente no sea una WebApi dado que no presenta controllers que exponen endpoints, técnicamente es una WebApi para exponer los servicios gRPC.

DataAccess: Contiene las clases responsables de almacenar los datos del sistema. En vistas de una solución más simple, se optó por crear un repositorio por cada entidad de dominio y no utilizar un único repositorio genérico. Se destaca la utilización del patrón "Singleton" aplicado a todos los repositorios existentes, el cual se explica más adelante en el documento.

Domain: Responsable de contener las entidades de dominio que el sistema necesita como base.

Exceptions: Contiene clases que representan excepciones personalizadas. En caso de encontrar algún error en la validación de los modelos de entrada, se lanza su correspondiente excepción personalizada que contiene un mensaje detallando lo ocurrido, con el fin de mejorar el feedback hacia el cliente a la hora de lanzar una excepción.

FileLogic: Paquete que define los métodos de lectura y escritura de archivos. A su vez, contiene un manager para obtener información y realizar validaciones de archivos.

FunctionInterface: Contiene las interfaces que definen el comportamiento que las funcionalidades del cliente y servidor deben implementar. Se generó principalmente para no violar el principio de Open/Close, en caso de que se necesite implementar nuevas funcionalidades en un futuro.

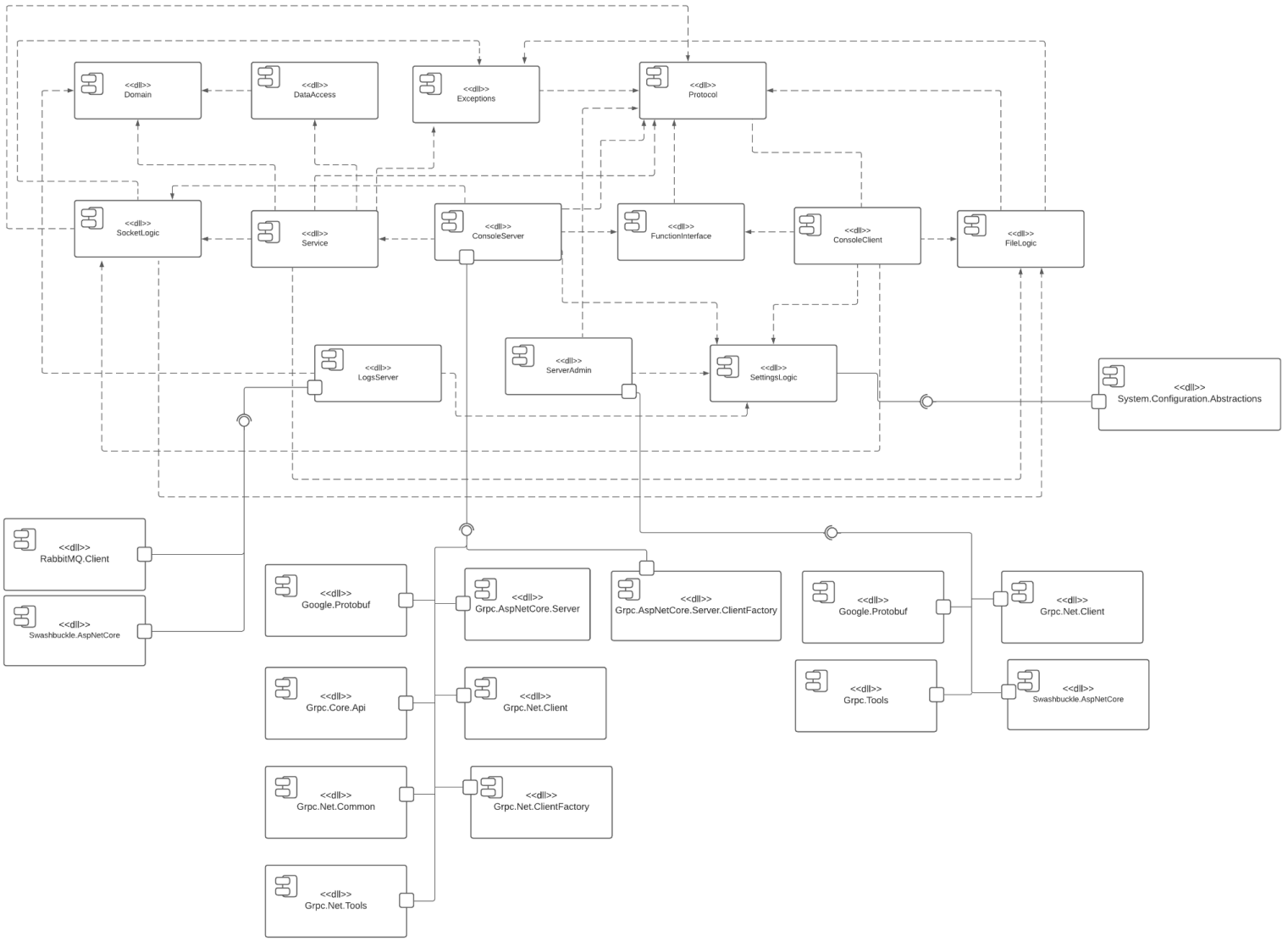
Protocol: Aquí es donde se define el protocolo necesario para que las aplicaciones cliente y servidor intercambien información mutuamente. Contiene el formato para los "headers", las constantes para las funciones, es decir los comandos para cada funcionalidad del sistema, las constantes para el "header" y los status code creados. La clase "DataPacket" es una fabricación pura que representa el contenido de un paquete en el intercambio de información, a diferencia de la clase "ResponseData" que se desarrolló como contenedor temporal para almacenar las respuestas del servidor.

Service: Contiene implementaciones de lógica de negocio para los recursos expuestos. Se utilizan para procesar las consultas que le son delegadas. Dentro, cada funcionalidad tiene sus propias validaciones, así como métodos encargados de colaborar con otras entidades, para brindar el servicio correspondiente.

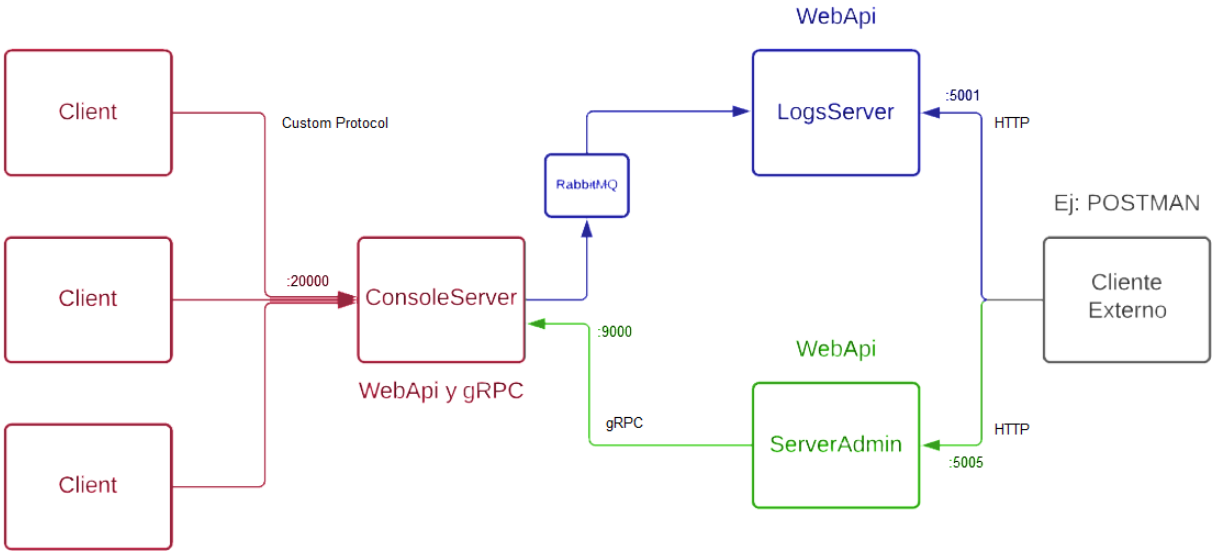
SettingsLogic: Paquete que contiene la lógica dirigida a leer el archivo “appsettings”. Cuenta con la interfaz “ISettingsManager” que se encarga de definir los métodos implementados por la clase SettingsManager.

SocketLogic: Paquete que define los métodos de envío y recepción de bytes entre diferentes sockets. Es el nivel más bajo al cual llegaremos en nuestro sistema.

Diagrama de Implementación (Componentes)

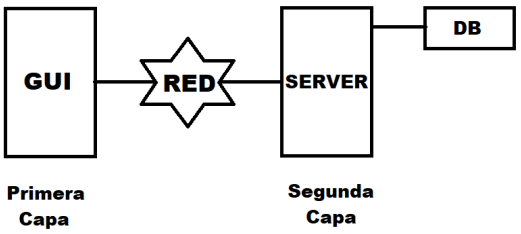


Justificación del Diseño



Console Server (Rojo)

A modo de introducción, comenzando por la arquitectura de alto nivel, el proyecto se basa en un modelo de dos capas: GUI en consola y servidor con la lógica y base de datos en memoria.



Nuevamente cabe destacar que, en esta oportunidad, no se cuenta con persistencia de datos mediante el uso de una base de datos, sino que se recurre al almacenamiento en memoria para los datos ingresados al sistema.

Al momento de desarrollo, la capa de Server se encuentra igualmente dividida en subcapas, siendo estas: Console Server, Service y DataAccess. Este enfoque, que consiste en segmentar la capa según sus responsabilidades, cumple con el principio de responsabilidad única (SRP de SOLID) y ayuda a mantener organizado el código para que se pueda encontrar fácilmente dónde se implementa una función determinada.

La comunicación entre el cliente y el servidor se realiza mediante Sockets. Al momento de inicializar el servidor se ejecuta de forma asincrónica el método CreateHostBuilder(args)... para recibir las llamadas gRPC, el método Connect() de la clase LogSender para crear la conexión con la cola de logs de RabbitMQ y el método ListenForConnections() que emplea la clase Socket para comenzar a escuchar conexiones con los sockets de clientes. La ejecución asincrónica se realiza utilizando la clase Task de la librería System.Threading.Tasks

Ver Diagrama de Clase: ConsoleClient
Ver Diagrama de Clase: ConsoleServer

Utilizamos el término async para convertir un método en asincrónico principalmente en operaciones de Entrada y Salida (I/O) en la red y el stream para los archivos. Conjuntamente con async, utilizamos el término await que envía la ejecución a segundo plano y nos devuelve el control al método que invoca. Finalmente, se destaca que seguimos la recomendación de agregar el sufijo Async al nombre del método.

El protocolo generado para nuestro sistema es utilizado por ambas partes en el intercambio de información. Se envían headers que almacenan la información correspondiente al tipo de envío, indicando si se trata de una petición o respuesta, seguido por el comando específico de la funcionalidad deseada y el largo de la información que se enviará luego del header.

Ver Diagrama de Clase: Protocol

El formato general de la trama se representa en la clase DataPacket y contiene:

Nombre Del Campo	HEADER	CMD	LARGO	DATOS	STATUS CODE
Valores	RES/REQ	0-99	Entero	Variable	200/201/400/404/500
Largo	3	2	4	Variable	3

El receptor de la información crea un buffer donde reserva el espacio necesario en bytes para almacenar la información y se espera por su llegada en la correspondiente segmentación de paquetes.
Se utilizó la técnica Data Transfer Object (DTO) que consiste en limitar la información que se recibe y devuelve utilizando modelos en formato de cadena de caracteres, separando el contenido con el carácter “#”, o en ocasiones se utiliza también “&”, que tienen un sub-set de determinada entidad del dominio. Estos modelos son usados principalmente para omitir ciertas properties al momento de intercambiar información con la finalidad de reducir tamaño de las entidades y esconder información, simplificando el problema para los clientes.

Una de las características esenciales que identifica a la arquitectura cliente-servidor es que permite la conexión de varios clientes simultáneamente con el servidor. Para lograr esta concurrencia se utilizan Tasks, donde cada cliente conectado tiene su ejecución asíncrona para realizar operaciones, lo que genera un problema de lectura sucia cuando múltiples clientes intentan acceder al mismo recurso (listas de memoria compartida) mientras se actualiza su contenido. Para ello, se soluciona implementando un control de acceso a los recursos (listas de memoria compartida) utilizando locks que los bloqueen cada vez que un cliente quiera realizar operaciones que utilicen su contenido.

A su vez, se implementó el patrón creacional Singleton para los repositorios en DataAccess y las lógicas en Service, como mecanismo para asegurar que estas clases tienen una única instancia disponible, controlando el acceso al recurso compartido, y se les proporcione un único punto de acceso global para dicha instancia.

Ver Diagrama de Clase: DataAccess

Se implementó el patrón creacional Factory Method para desarrollar las funciones disponibles en el proyecto. Para lograrlo, se proporcionan las interfaces IClientFunction e IServerFunction, contenidas en el paquete FunctionInterface, que exponen el método común Execute() para ser implementado de forma diferente por función. El método fábrica está contenido en las clases FunctionDictionary para cliente y servidor, el cual retorna un diccionario con clave de ID de Función (especificadas como constantes en la clase FunctionConstants) y valor de Instancia de Clase que implementa la Interface para una función particular.

El código que utiliza el método fábrica no encuentra diferencias entre las funciones devueltas y las trata como la Interface IClientFunction o IServerFunction. El cliente sabe que todas las funciones deben tener el método Execute() implementado, pero no necesita saber cómo funciona exactamente.
Al poder incorporar nuevos tipos de funciones en el programa sin descomponer el código existente se cumple el principio de abierto/cerrado.

Ver Diagrama de Clase: FunctionInterface

Se conoce que Factory Method puede funcionar conjuntamente con el patrón de comportamiento Template Method. Optamos por utilizar Template Method que defina un esqueleto de un algoritmo en el método Execute() de la superclase FunctionTemplate, pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura. Los pasos pueden ser abstractos, o ser virtual contando con una implementación por defecto.

Para el cliente, los pasos definidos en el método Execute() son:
BuildRequest() → SendRequest() → ReceiveResponse() → ProcessResponse()

Para el servidor, los pasos definidos en el método Execute() son:
ReceiveRequest() → ProcessRequest() → BuildResponse() → SendResponse()

Los pasos relacionados con Build y Process son declarados como abstractos, forzando las subclases a proporcionar sus propias implementaciones para estos métodos. Ahora, los pasos relacionados con Send y Receive están implementados dentro de la superclase para deshacerse de código duplicado, puesto que no tiene sentido proporcionar implementaciones en la totalidad de subclases, y son declarados como virtual, para poder sobreescribirse en funciones particulares como las encargadas de transferir archivos.

Por ejemplo, para añadir un nuevo tipo de función a la aplicación, sólo se debe crear una nueva subclase de FunctionTemplate, implementar los métodos Build() y Process() y finalmente agregarla al diccionario de FunctionDictionary con clave de ID de Función (especificadas como constantes en la clase FunctionConstants) y valor de Instancia de Clase.

Ver Diagrama de Clase: ConsoleClient
Ver Diagrama de Clase: ConsoleServer

Llegados a este punto, queremos aclarar que se efectuó un supuesto para el requerimiento funcional: CRF4. Búsqueda de juegos. Se deberá poder buscar un juego por categoría, título y rating.
Optamos por implementar el requisito en tres funcionalidades independientes de búsqueda, por lo que el usuario no podrá filtrar con múltiples criterios. Creemos fuertemente que al ser de esta manera, se genera un mayor entendimiento acerca de la búsqueda de juegos que el sistema ofrece.

Para verificar la correctitud de los modelos de entrada, se implementaron clases validadoras en el paquete Service. Entre las validaciones implementadas, la principal y comúnmente utilizada trata de que los datos de entrada no estén vacíos.

Ver Diagrama de Clase: Service

En caso de encontrar un defecto en los modelos de entrada, se implementaron un conjunto de excepciones personalizadas, que son lanzadas según el contexto particular, para tener mayor conocimiento de lo ocurrido. Los casos contemplados hasta la fecha son:

- Not Found: No se encuentra una entidad solicitada.
- Invalid Input: La información entrante contiene datos erróneos.
- Already Exists: La entidad entrante ya existe en el sistema.
- No Readable File: Un archivo no se puede leer.

Estas excepciones personalizadas heredan de una excepción generalizada igualmente personalizada que se encarga de asociar el mensaje de las hijas con un código de estado. Esta excepción generalizada (AppException) se utiliza para centralizar todas las excepciones personalizadas y es atrapada por el filtro de excepciones, generando un bloque try-catch más reducido.

Ver Diagrama de Clase: Exceptions

Se utiliza un conjunto reducido de códigos de estado, comúnmente conocidos por los desarrolladores, permitiendo simplificar el significado de retorno.

- 200 - Ok:
  Se utiliza como respuesta a solicitudes GET correctas.
- 201 - Created:
  Se utiliza como respuesta a solicitudes POST correctas, indicando que la petición ha sido completada y ha resultado en la creación de un nuevo recurso.
- 400 - Bad Request:
  Se utiliza como respuesta a solicitudes erróneas (error de cliente), cuando la solicitud contiene sintaxis errónea, indicando que no procederá la solicitud en el servidor.
- 404 - Not Found:
  Se utiliza como respuesta a solicitudes erróneas (error de cliente), cuando no se encuentra el recurso solicitado.
- 500 - Internal Server Error:
  Se utiliza como respuesta a solicitudes erróneas (error de servidor), cuando ocurre un problema interno.

Cabe destacar que estos códigos se retornan acompañados de un mensaje significativo de lo ocurrido.

Por último, es fundamental tener carpetas en las siguientes ubicaciones para el correcto funcionamiento de las funcionalidades de envío y recepción de carátulas de juegos:
C:\VAPOR\SERVER
C:\VAPOR\CLIENT

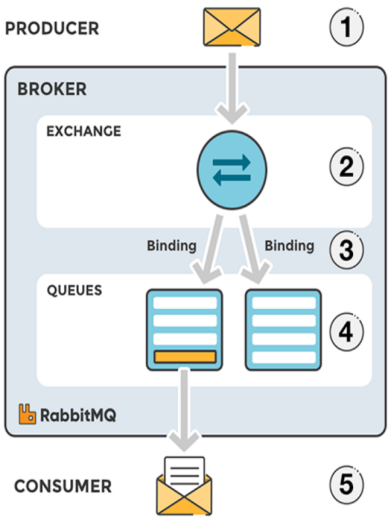
Logs Server (Azul)

Otro patrón de diseño identificado es el patrón Fachada. Se utiliza esencialmente en las API, dado que es un tipo de interfaz que expone un conjunto de funciones e información a desarrolladores independientes, cumpliendo con la intención del patrón. La responsabilidad de la fachada es saber a quién dirigir las peticiones, no debe tener lógica alguna.

El paquete LogsServer actúa como fachada al sistema completo, ya que contiene a los controladores que son el punto de entrada para los clientes.

Nuevamente cabe destacar que, en esta oportunidad, no se cuenta con persistencia de datos mediante el uso de una base de datos, sino que se recurre al almacenamiento en memoria para los logs.
Se implementó el patrón creacional Singleton para el repositorio de logs, como mecanismo para asegurar que tenga una única instancia disponible, controlando el acceso al recurso compartido, y se les proporcione un único punto de acceso global para dicha instancia.
Se genera un problema de lectura sucia cuando múltiples clientes intentan acceder al mismo recurso (lista de memoria compartida) mientras se actualiza su contenido. Para ello, se soluciona implementando un control de acceso al recurso (lista de memoria compartida) utilizando lock que los bloquee cada vez que un cliente quiera realizar operaciones que utilicen su contenido.

MOM



Utilizamos la tecnología MOM: Message Oriented Middlewares, mediante el uso de RabbitMQ, para el registro de logs del sistema dado que permite la independencia de la comunicación con respecto al Tiempo al interconectar el Console Server con LogsServer, inclusive estando offline el extremo consumidor, sin afectar al que produce el mensaje. Esto es que el productor Console Server puede producir mensajes (Logs) y encolarlos en “log_queue” incluso cuando el consumidor LogsServer se encuentre offline, de modo que los mensajes (Logs) quedan encolados hasta ser consumidos.

El uso de está tecnología nos permite almacenar Logs de forma sincrónica cuando ambos servidores están online o de forma asincrónica cuando únicamente Console Server se encuentre online.

Se utilizó la tecnología de RabbitMQ como implementación de MOM.

Para ello, el ConsoleServer contiene internamente el namespace ConsoleServer.LogsLogic que incluye la clase LogSender para enviar logs, conectarse y generar el canal con RabbitMQ. Se añadió un nuevo paso al FunctionTemplate que implica ejecutar el método abstracto SendLog(), que es implementado por cada función que herede de esta clase. La implementación implica generar el log correspondiente a la función y utilizar el método SendLog() de LogSender para encolarlo.

Luego, el LogsServer contiene internamente el namespace LogsServer.LogsLogic que incluye la clase LogReceiver para consumir logs, conectarse y generar el canal con RabbitMQ. Al consumir los logs, los descerializa y almacena en el repositorio.

Admin Server (Verde)

De igual manera que LogsServer, el paquete ServerAdmin actúa como fachada al sistema completo, ya que contiene a los controladores que son el punto de entrada para los clientes.

Se utilizó la técnica Data Transfer Object (DTO), como buena práctica para el desarrollo de la API, esta consiste en limitar la información que recibe y devuelve utilizando modelos que tienen un sub-set de determinada entidad del dominio. En este nuevo namespace Models, se encuentran modelos de salida (ModelOut) y modelos de entrada (ModelIn). La realización de modelos es una implementación del patrón Adapter.

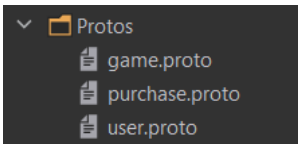
Estos modelos son usados principalmente para omitir algunas properties con la finalidad de reducir tamaño de las entidades y esconder información, simplificando el problema para los clientes. A su vez, contar con modelos ayuda a minimizar el impacto de cambio en caso de que las entidades cambien su estado interno. En conclusión, la API solamente conoce modelos y la lógica de negocio conoce las entidades.

Servicios gRPC

gRPC es un proyecto de open-source desarrollado por Google que nos permite llamar desde un cliente, un método o servicio como si fuese de forma local. Funcionan a través de comunicación binaria, basado en contratos, es independiente del sistema, soporta streaming y es seguro. Su principal atractivo es la mejora en la performance. Cabe destacar que trabaja sobre Http/2.

Funcionan a través de la definición de Protocol buffers, que son un mecanismo automatizado, eficiente, flexible para serializar data estructurada, parecido a XML pero más pequeño, rápido y sencillo. Nosotros somos quienes definimos cómo se van a estructurar los datos. Especificamos como se quiere que se serialize la información definiendo mensajes de protocol buffer en archivos de formato .proto. Cada mensaje de protocol buffer es una pequeña entrada de información que contiene la información en pares clave – valor.

En nuestro sistema, estos archivos .proto los podemos encontrar tanto en el server que ofrece los servicos gRPC (ConsoleServer) como también en el cliente que consume esos servicios (ServerAdmin). Los organizamos dentro de directorios llamados Protos con el fin de una mejor organización desde el punto de vista de las responsabilidades.



Cada uno de estos archivos tiene la estructura de las funciones que se ejecutan remotamente y el formato de los mensajes, con cada una de sus variables indexadas.

Ejemplo de game.proto:

```
syntax = "proto3";

option csharp_namespace = "ServerAdmin";

package game;

service GameManager {
  rpc PostGame (GameParam) returns (GameReply);
  rpc DeleteGame (GameParam) returns (GameReply);
  rpc PutGame (GameParam) returns (GameReply);
}

message GameReply {
  int32 status_code = 1;
}

message GameParam {
  string line = 1;
}
```

Las funcionalidades ofrecidas por la WebApi ServerAdmin, son ABM tanto para juegos, usuarios y compras. Para ejecutar dichos servicios debemos realizar las peticiones correspondientes. Cuando llega una petición a cualquier controlador de ServerAdmin, desde allí se invocará al servicio correspondiente perteneciente a los servicios gRPC. Cabe destacar que, además de ejecutar la lógica de la operación, luego se generará el log correspondiente para ser almacenado.
Siguiendo con el ejemplo de juegos, se detalla el código perteneciente al controlador para una mejor observación:

```
namespace ServerAdmin.Controllers
{
    [ApiController]
    [Route(template: "games")]
    public class GameController : ControllerBase
    {
        private readonly ISettingsManager _settingsManager = new SettingsManager();
        private readonly GrpcChannel _channel;

        public GameController()
        {
            var protocol:string = _settingsManager.ReadSetting(ServerConfig.ProtocolConfigKey);
            var ip:string = _settingsManager.ReadSetting(ServerConfig.IPAddressConfigKey);
            var port:string = _settingsManager.ReadSetting(ServerConfig.GRPCPortConfigKey);
            _channel = GrpcChannel.ForAddress($"{protocol}://{ip}:{port}", new GrpcChannelOptions() {
                HttpClient = new System.Net.Http.HttpClient() {
                    DefaultRequestVersion = new Version(major: 2, minor: 0)
                }
            });
        }
```

```
[HttpPost]
public async Task<IActionResult> Post([FromBody] GameModelIn model)
{
    var client = new GameManager.GameManagerClient(_channel);
    var gameLine = new GameParam() {Line = model.ParseToPostFormat()};
    var response:GameReply = await client.PostGameAsync(gameLine);
    GameModelOut modelOut = new GameModelOut() {Title = model.Title};
    return response.StatusCode == StatusCodeConstants.CREATED ? Created(uri:string.Empty, modelOut) : StatusCode(response.StatusCode);
}
```

API REST

Uniform Interface

La API está basada en recursos que pueden ser accedidos por el cliente. Como ejemplo, las entidades de dominio que son persistentes. Cada recurso de nuestra API tiene un identificador (URI) bien definido y concreto, para que sea más fácil de comprender su funcionamiento, complementado con buenas prácticas como son utilizar sustantivos, en plural, con letra minúscula y con niveles de extensión cortos.

Stateless

La API no maneja estado, cada consulta HTTP es independiente dado que contiene toda la información requerida para ser interpretada por el servidor.

Ejemplo: Por cada consulta que requiera de información, como sea un identificador, deberá ser enviada sin importar que una consulta anterior haya utilizado esa misma información.

Cacheable

Actualmente la API no define las respuestas del servidor como cacheables.
Un beneficio que se obtiene al tener respuestas cacheables es mejorar la performance del servidor. A pesar de esto, consideramos que el servidor no sufrirá problemas de performance en esta etapa ya que no será muy elevado el número de consultas.

Client-Server

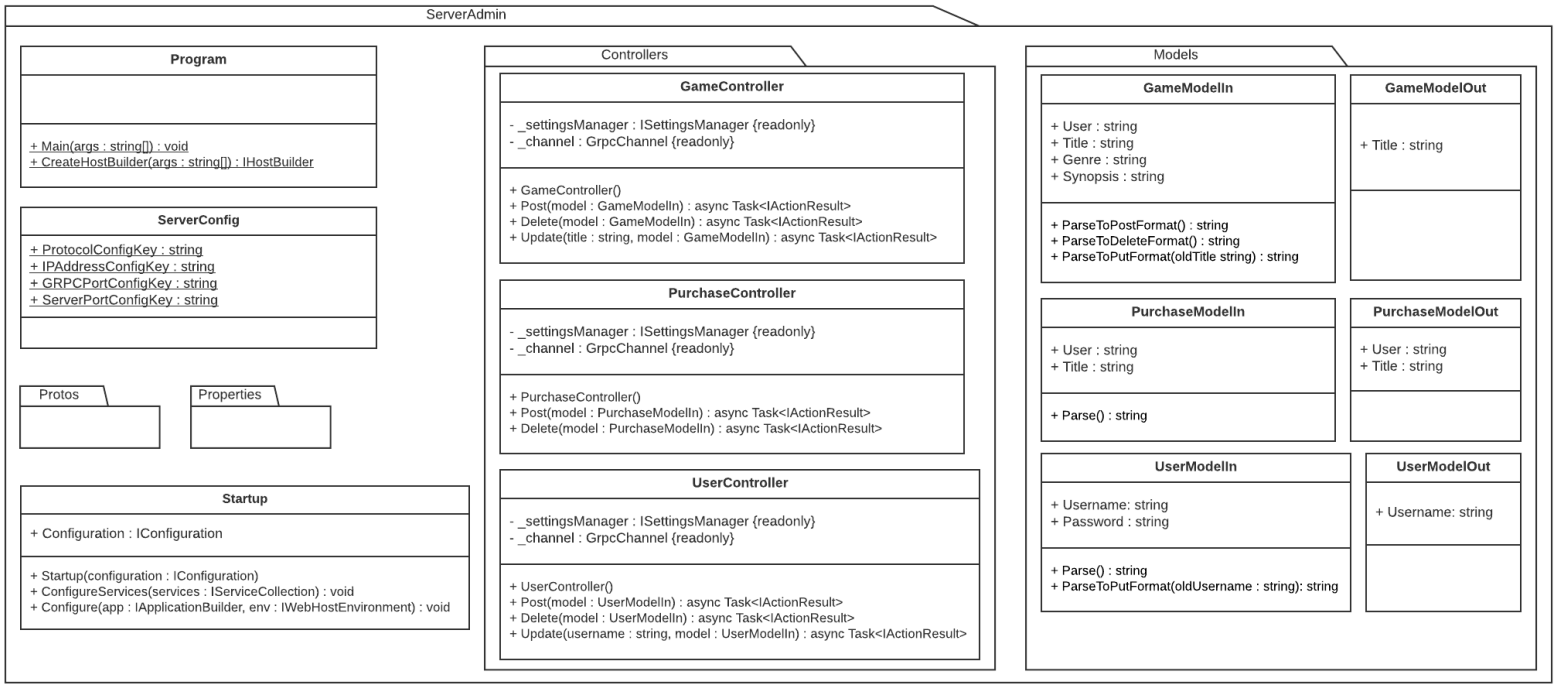
La API es cliente-servidor debido al criterio de interfaz uniforme que permite separar clientes de servidores. Esto implica que los clientes no se preocupan de cómo los datos son almacenados y el servidor no se preocupa por manejar la interfaz o el estado de usuario. Mientras la interfaz no se altere, los clientes y servidores pueden ser reemplazados y desarrollados independientemente.

Layered System

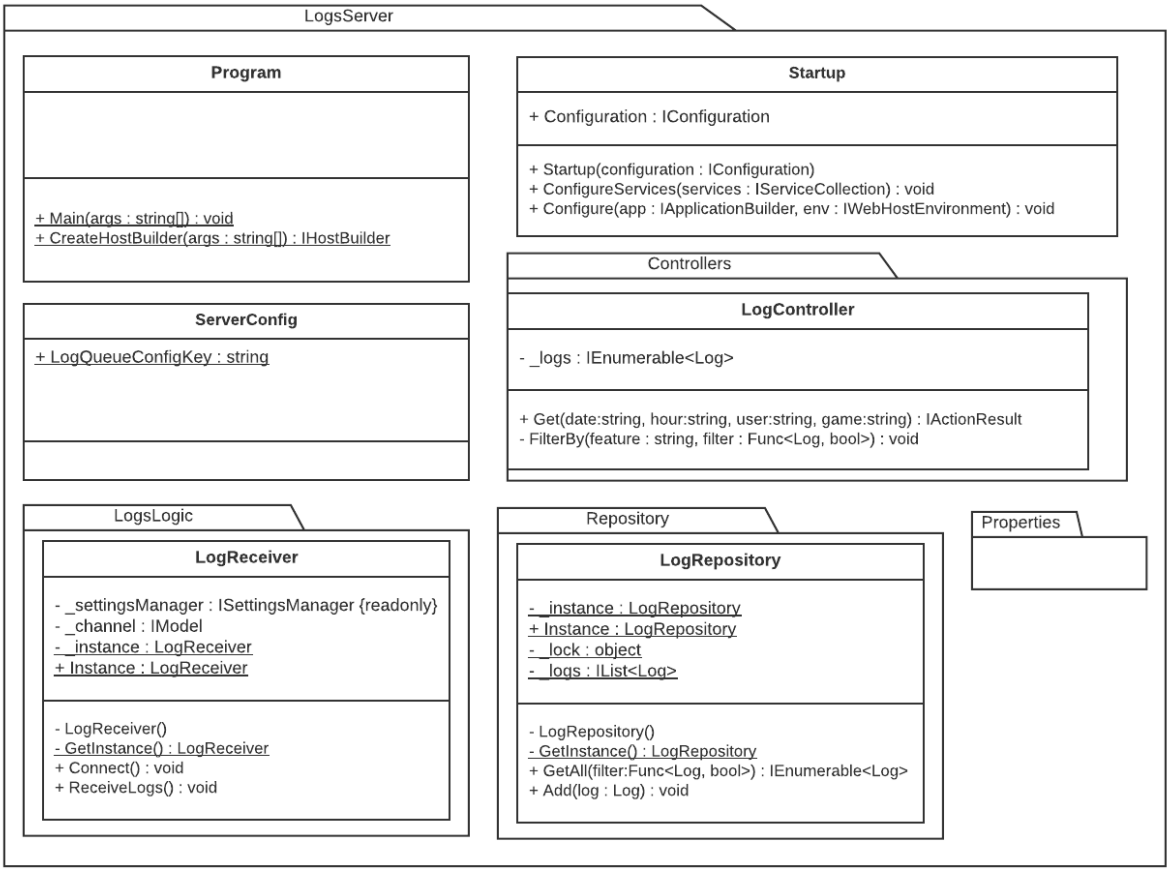
La API se basa en una arquitectura en capas distribuidas jerárquicamente.

Anexo: Diagramas de Clases

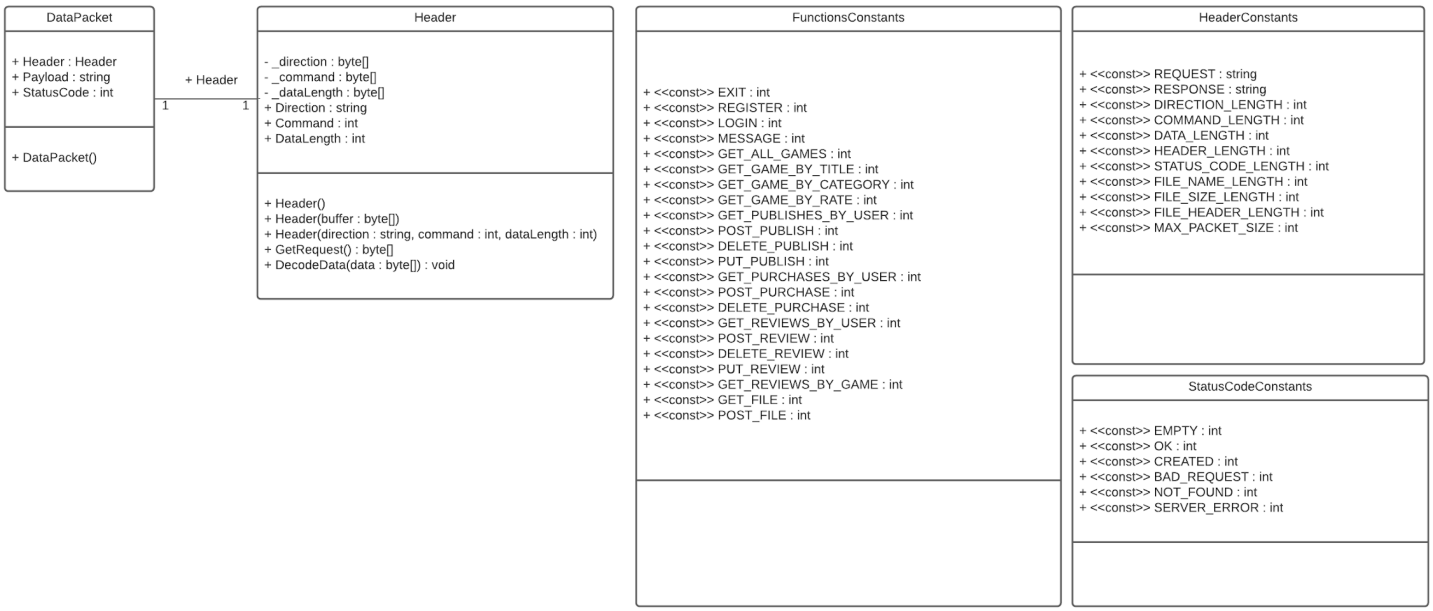
ServerAdmin



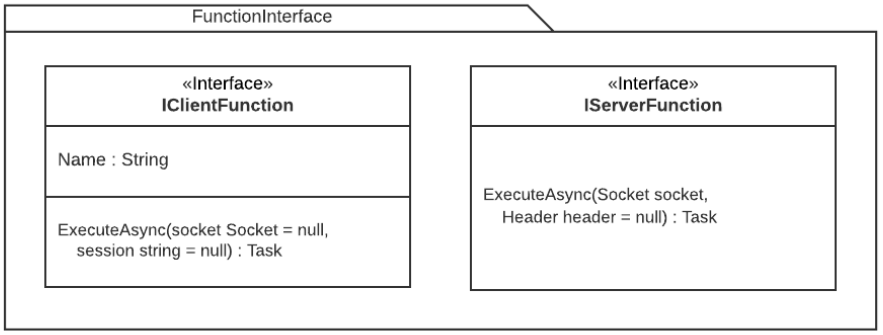
LogsServer


LogsServer
Program
+ Main(args : string[]) : void
+ CreateHostBuilder(args : string[]) : IHostBuilder
Startup
+ Configuration : IConfiguration
+ Startup(configuration : IConfiguration)
+ ConfigureServices(services : IServiceCollection) : void
+ Configure(app : IApplicationBuilder, env : IWebHostEnvironment) : void
ServerConfig
+ LogQueueConfigKey : string
Controllers
LogController
- _logs : IEnumerable<Log>
+ Get(date:string, hour:string, user:string, game:string) : IActionResult
- FilterBy(feature : string, filter : Func<Log, bool>) : void
LogsLogic
LogReceiver
- _settingsManager : ISettingsManager {readonly}
- _channel : IModel
- _instance : LogReceiver
+ Instance : LogReceiver
- LogReceiver()
- GetInstance() : LogReceiver
+ Connect() : void
+ ReceiveLogs() : void
Repository
LogRepository
- _instance : LogRepository
+ Instance : LogRepository
- _lock : object
- _logs : IList<Log>
- LogRepository()
- GetInstance() : LogRepository
+ GetAll(filter:Func<Log, bool>) : IEnumerable<Log>
+ Add(log : Log) : void
Properties

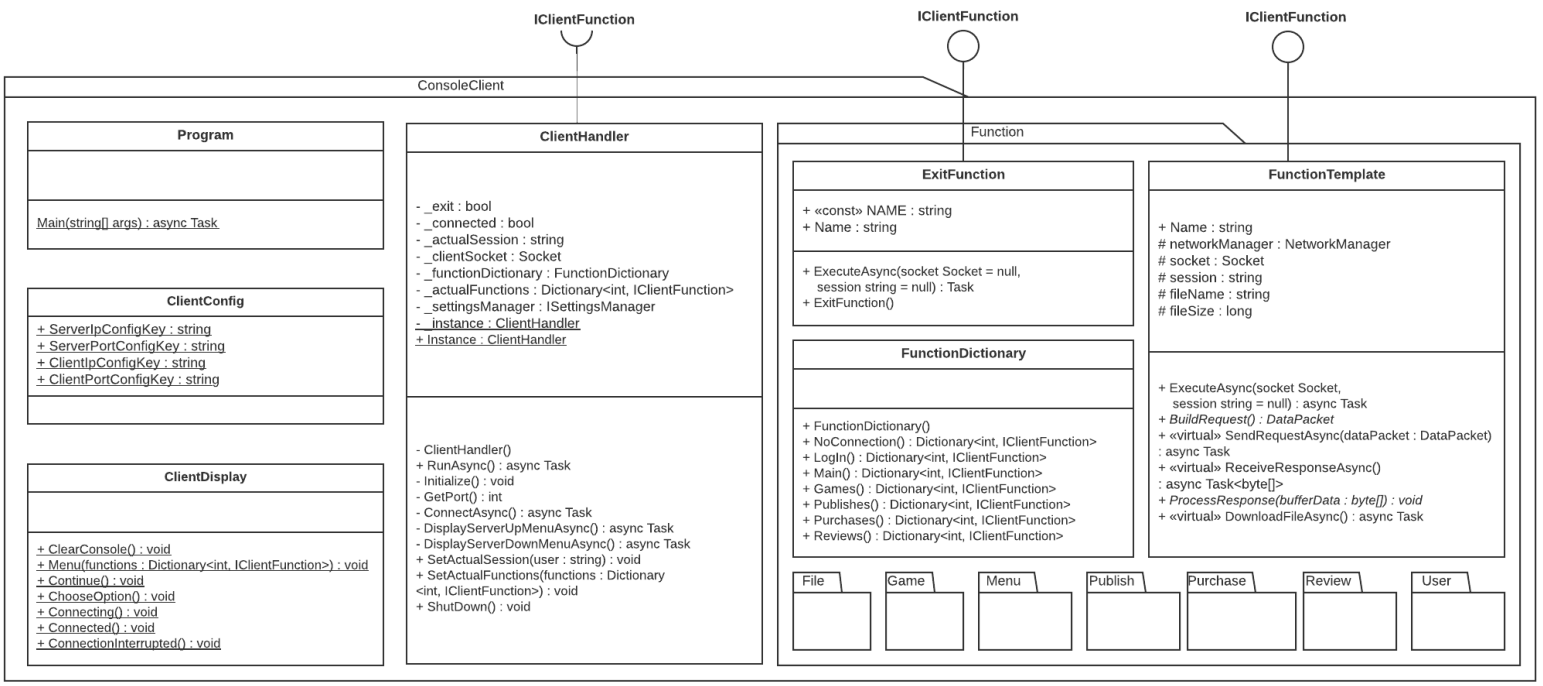
Protocol



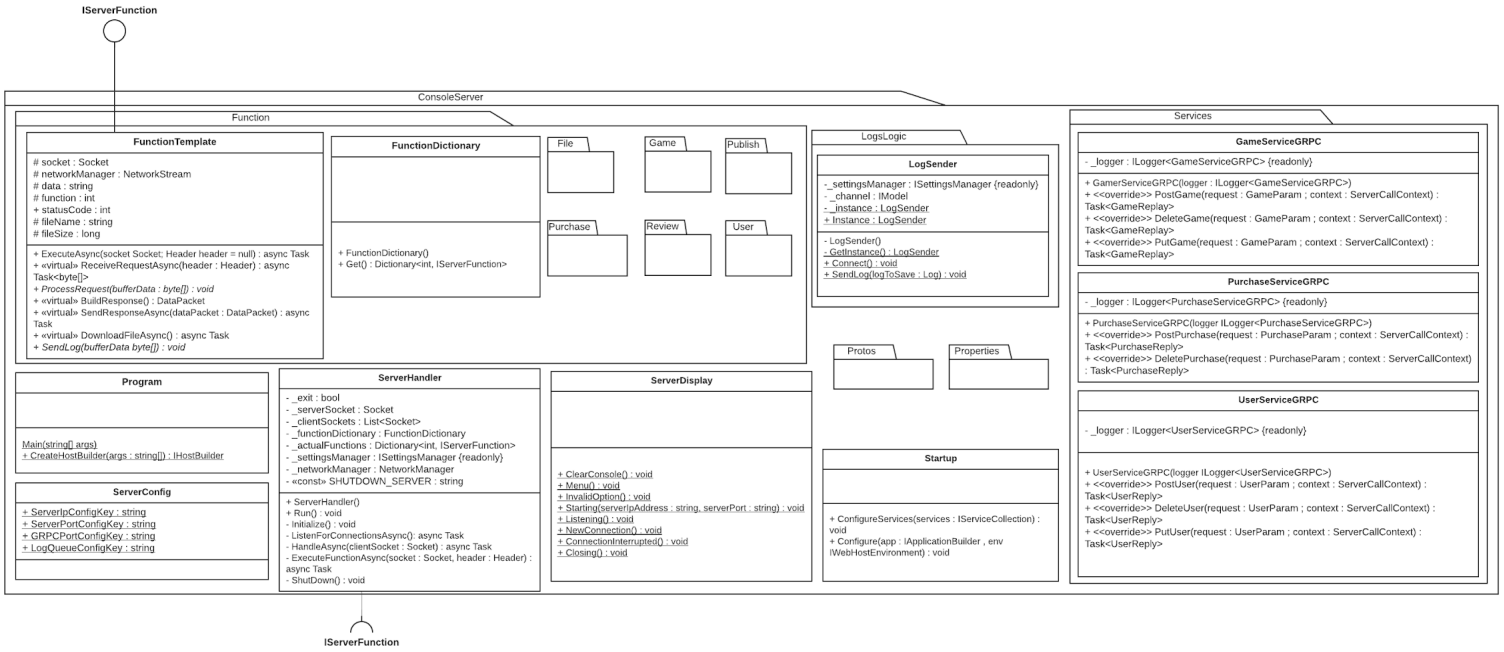
FunctionInterface



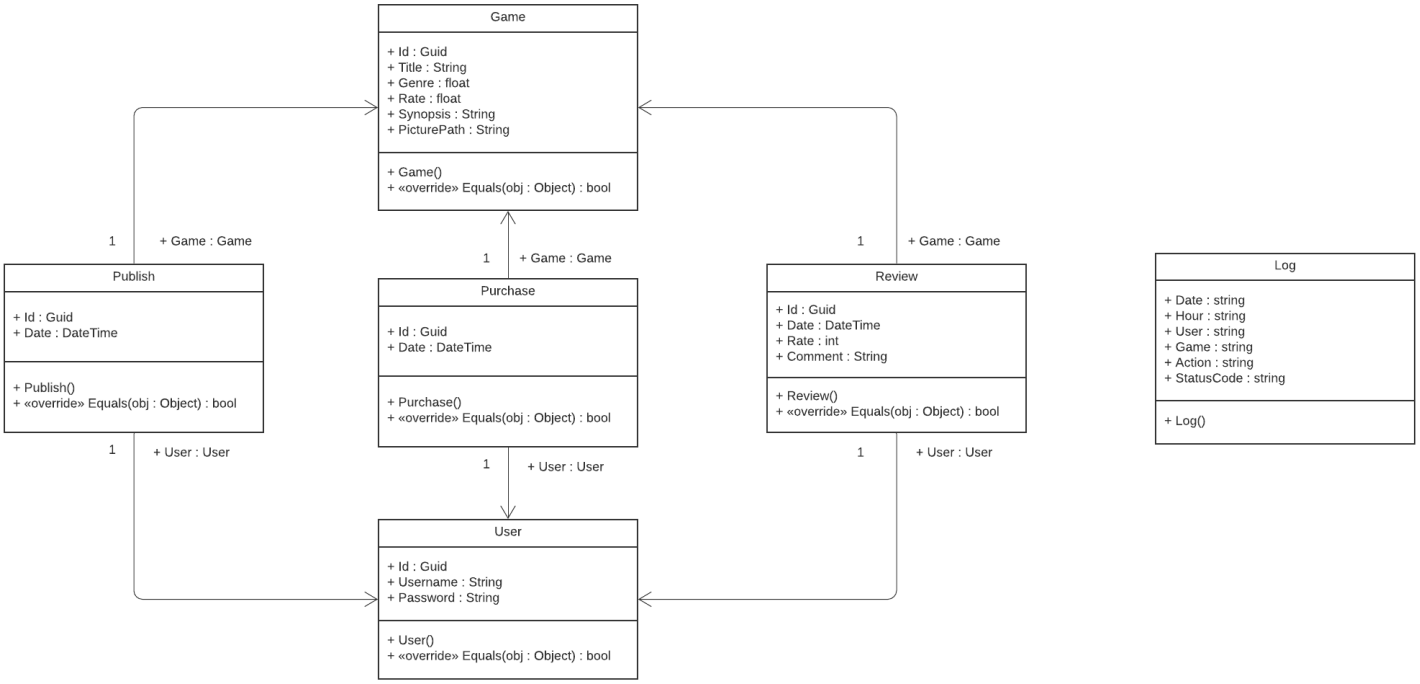
ConsoleClient



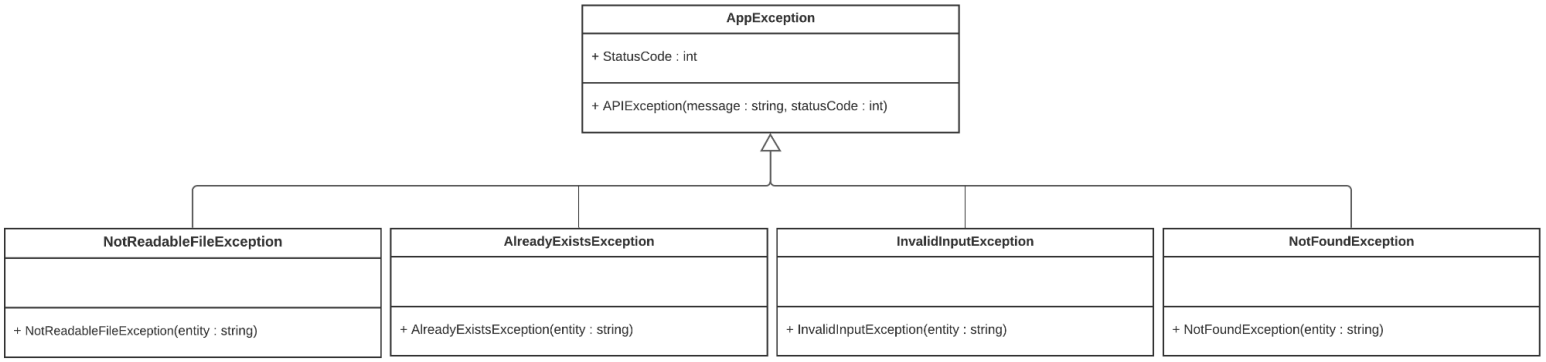
ConsoleServer



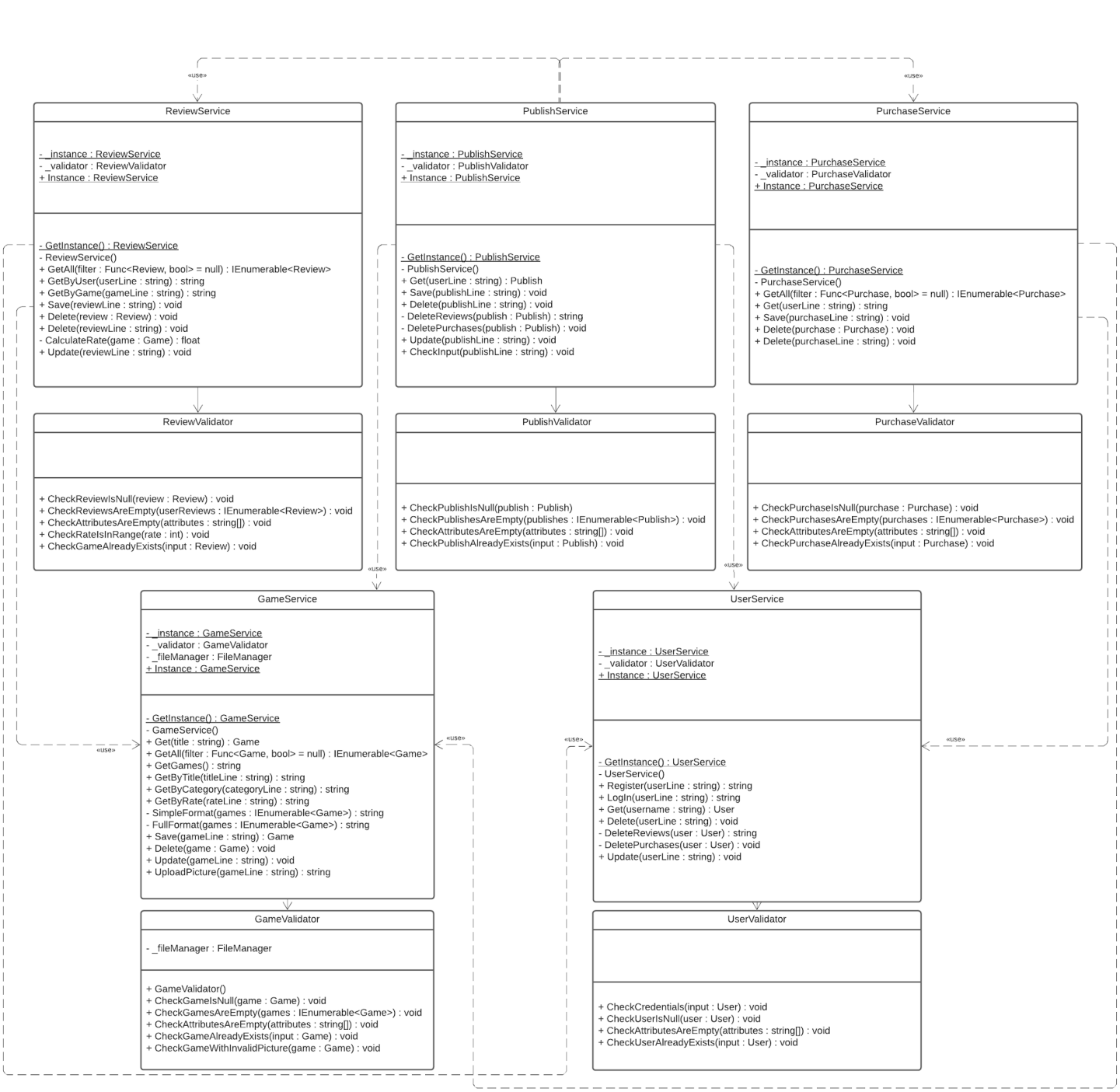
Domain



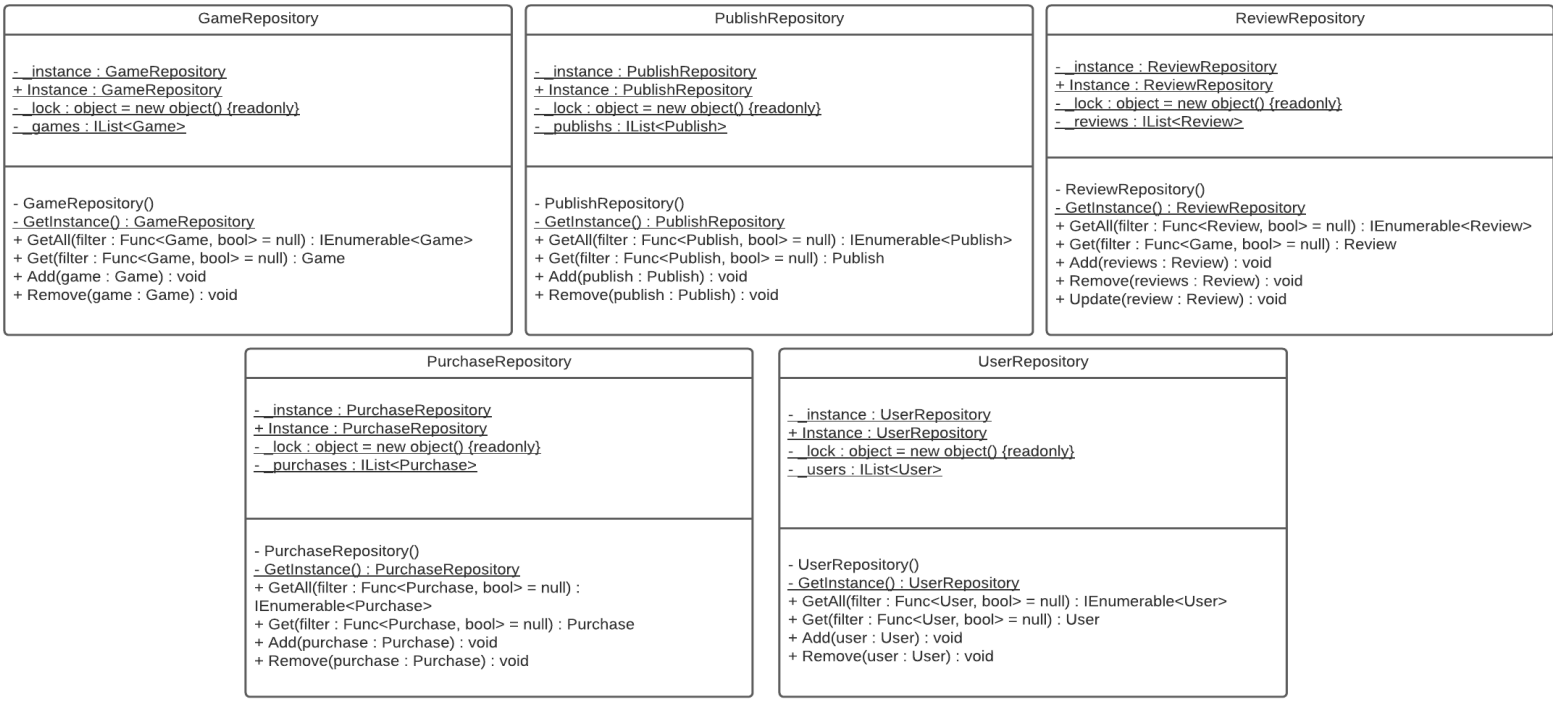
Exceptions



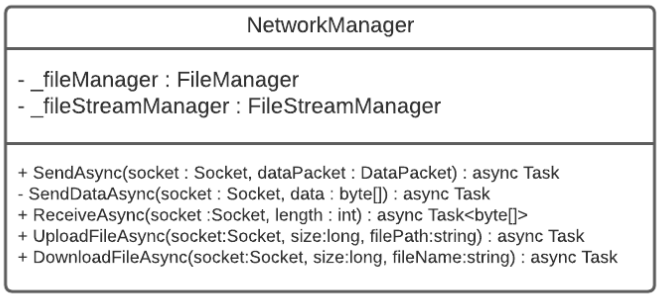
Service


IServerFunct
«use»
«use»
ReviewService
- _instance : ReviewService
- _validator : ReviewValidator
+ Instance : ReviewService
- GetInstance() : ReviewService
- ReviewService()
+ GetAll(filter : Func<Review, bool> = null) : IEnumerable<Review>
+ GetByUser(userLine : string) : string
+ GetByGame(gameLine : string) : string
+ Save(reviewLine : string) : void
+ Delete(review : Review) : void
+ Delete(reviewLine : string) : void
- CalculateRate(game : Game) : float
+ Update(reviewLine : string) : void
PublishService
- _instance : PublishService
- _validator : PublishValidator
+ Instance : PublishService
- GetInstance() : PublishService
- PublishService()
+ Get(userLine : string) : Publish
+ Save(publishLine : string) : void
+ Delete(publishLine : string) : void
- DeleteReviews(publish : Publish) : string
- DeletePurchases(publish : Publish) : void
+ Update(publishLine : string) : void
+ CheckInput(publishLine : string) : void
PurchaseService
- _instance : PurchaseService
- _validator : PurchaseValidator
+ Instance : PurchaseService
- GetInstance() : PurchaseService
- PurchaseService()
+ GetAll(filter : Func<Purchase, bool> = null) : IEnumerable<Purchase>
+ Get(userLine : string) : string
+ Save(purchaseLine : string) : void
+ Delete(purchase : Purchase) : void
+ Delete(purchaseLine : string) : void
ReviewValidator
+ CheckReviewIsNull(review : Review) : void
+ CheckReviewsAreEmpty(userReviews : IEnumerable<Review>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckRateIsInRange(rate : int) : void
+ CheckGameAlreadyExists(input : Review) : void
PublishValidator
+ CheckPublishIsNull(publish : Publish)
+ CheckPublishesAreEmpty(publishes : IEnumerable<Publish>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckPublishAlreadyExists(input : Publish) : void
PurchaseValidator
+ CheckPurchaseIsNull(purchase : Purchase) : void
+ CheckPurchasesAreEmpty(purchases : IEnumerable<Purchase>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckPurchaseAlreadyExists(input : Purchase) : void
«use»
«use»
GameService
- _instance : GameService
- _validator : GameValidator
- _fileManager : FileManager
+ Instance : GameService
- GetInstance() : GameService
- GameService()
+ Get(title : string) : Game
+ GetAll(filter : Func<Game, bool> = null) : IEnumerable<Game>
+ GetGames() : string
+ GetByTitle(titleLine : string) : string
+ GetByCategory(categoryLine : string) : string
+ GetByRate(rateLine : string) : string
- SimpleFormat(games : IEnumerable<Game>) : string
- FullFormat(games : IEnumerable<Game>) : string
+ Save(gameLine : string) : Game
+ Delete(game : Game) : void
+ Update(gameLine : string) : void
+ UploadPicture(gameLine : string) : string
UserService
- _instance : UserService
- _validator : UserValidator
+ Instance : UserService
- GetInstance() : UserService
- UserService()
+ Register(userLine : string) : string
+ LogIn(userLine : string) : string
+ Get(username : string) : User
+ Delete(userLine : string) : void
- DeleteReviews(user : User) : string
- DeletePurchases(user : User) : void
+ Update(userLine : string) : void
«use»
«use»
«use»
«use»
GameValidator
- _fileManager : FileManager
+ GameValidator()
+ CheckGameIsNull(game : Game) : void
+ CheckGamesAreEmpty(games : IEnumerable<Game>) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckGameAlreadyExists(input : Game) : void
+ CheckGameWithInvalidPicture(game : Game) : void
UserValidator
+ CheckCredentials(input : User) : void
+ CheckUserIsNull(user : User) : void
+ CheckAttributesAreEmpty(attributes : string[]) : void
+ CheckUserAlreadyExists(input : User) : void

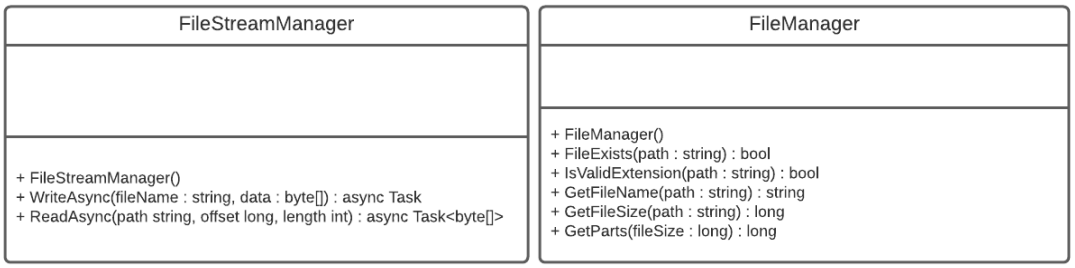
DataAccess


GameRepository
- _instance : GameRepository
+ Instance : GameRepository
- _lock : object = new object() {readonly}
- _games : IList<Game>
- GameRepository()
- GetInstance() : GameRepository
+ GetAll(filter : Func<Game, bool> = null) : IEnumerable<Game>
+ Get(filter : Func<Game, bool> = null) : Game
+ Add(game : Game) : void
+ Remove(game : Game) : void
PublishRepository
- _instance : PublishRepository
+ Instance : PublishRepository
- _lock : object = new object() {readonly}
- _publishs : IList<Publish>
- PublishRepository()
- GetInstance() : PublishRepository
+ GetAll(filter : Func<Publish, bool> = null) : IEnumerable<Publish>
+ Get(filter : Func<Publish, bool> = null) : Publish
+ Add(publish : Publish) : void
+ Remove(publish : Publish) : void
ReviewRepository
- _instance : ReviewRepository
+ Instance : ReviewRepository
- _lock : object = new object() {readonly}
- _reviews : IList<Review>
- ReviewRepository()
- GetInstance() : ReviewRepository
+ GetAll(filter : Func<Review, bool> = null) : IEnumerable<Review>
+ Get(filter : Func<Game, bool> = null) : Review
+ Add(reviews : Review) : void
+ Remove(reviews : Review) : void
+ Update(review : Review) : void
PurchaseRepository
- _instance : PurchaseRepository
+ Instance : PurchaseRepository
- _lock : object = new object() {readonly}
- _purchases : IList<Purchase>
- PurchaseRepository()
- GetInstance() : PurchaseRepository
+ GetAll(filter : Func<Purchase, bool> = null) :
IEnumerable<Purchase>
+ Get(filter : Func<Purchase, bool> = null) : Purchase
+ Add(purchase : Purchase) : void
+ Remove(purchase : Purchase) : void
UserRepository
- _instance : UserRepository
+ Instance : UserRepository
- _lock : object = new object() {readonly}
- _users : IList<User>
- UserRepository()
- GetInstance() : UserRepository
+ GetAll(filter : Func<User, bool> = null) : IEnumerable<User>
+ Get(filter : Func<User, bool> = null) : User
+ Add(user : User) : void
+ Remove(user : User) : void

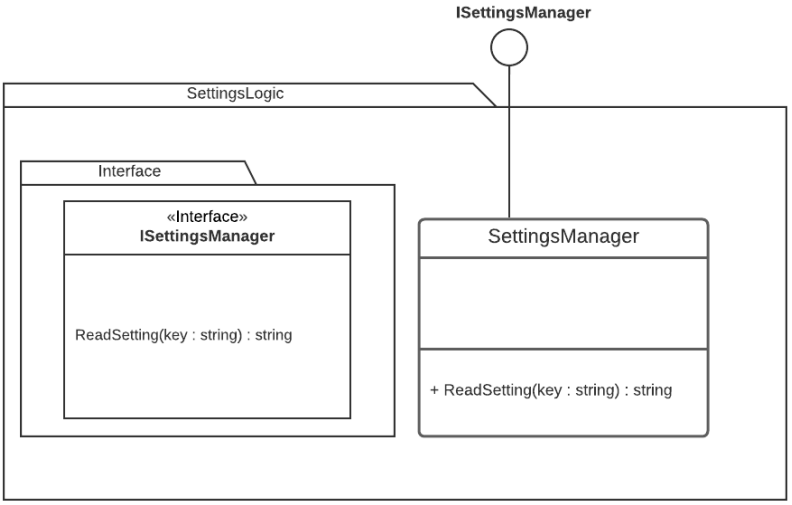
SocketLogic



FileLogic



SettingsLogic



Anexo: Listado de Funciones

ID	Funcion	Resource	Description	Responses
0	EXIT	User	Cierra la conexión	200
1	POST_USER	User	Crea un usuario	201, 400, 500
2	LOGIN	User	Logeo de usuario	200, 404, 500
3	DELETE_USER	User	Elimina el usuario correspondiente al nombre ingresado	200, 404, 500
4	PUT_USER	User	Modifica el usuario correspondiente al nombre ingresado	200, 400, 404, 500
5	GET_ALL_GAMES	Game	Retorna todos los juegos	200, 404, 500
6	GET_GAME_BY_TITLE	Game	Retorna el juego correspondiente al nombre ingresado	200, 404, 500
7	GET_GAME_BY_CATEGORY	Game	Retorna los juegos con igual categoría a la ingresada	200, 404, 500
8	GET_GAME_BY_RATE	Game	Retorna los juegos con igual calificación a la ingresada	200, 404, 500
10	GET_PUBLISHES_BY_USER	Publish	Retorna todas las publicaciones del usuario ingresado	200, 404, 500
11	POST_PUBLISH	Publish	Crea una publicación	201, 400, 500
12	DELETE_PUBLISH	Publish	Elimina la publicación correspondiente al nombre ingresado	200, 404, 500
13	PUT_PUBLISH	Publish	Modifica la publicación correspondiente al nombre del juego ingresado	200, 400, 404, 500
20	GET_PURCHASES_BY_USER	Purchase	Retorna todas las compras del usuario ingresado	200, 404, 500
21	POST_PURCHASE	Purchase	Crea una compra	201, 400, 500
22	DELETE_PURCHASE	Purchase	Elimina la compra correspondiente al nombre ingresado	200, 404, 500
30	GET_REVIEWS_BY_USER	Review	Retorna todas las reseñas del usuario ingresado	200, 404, 500
31	POST_REVIEW	Review	Crea una reseña	201, 400, 500
32	DELETE_REVIEW	Review	Elimina la reseña correspondiente al nombre ingresado	200, 404, 500
33	PUT_REVIEW	Review	Modifica la reseña correspondiente al nombre del juego ingresado	200, 400, 404, 500
36	GET_REVIEWS_BY_GAME	Review	Retorna todas las reseñas del juego ingresado	200, 404, 500
40	GET_FILE	Game	Retorna una imagen de un juego	200, 404, 500
41	POST_FILE	Publish	Carga una imagen en una publicación	201, 400, 500

Referencias bibliográficas

Apuntes e información de las prestaciones correspondientes a Programación de Redes, en Universidad ORT Uruguay.

Apuntes e información de las prestaciones correspondientes a Diseño de Aplicaciones 1 y 2, en Universidad ORT Uruguay.

Patron Singleton

Patron Factory Method

Patron Template Method