Guía Técnica - Video Game Store API

1. Portada (Cover Page)

- **Título**: Video Game Store API - Guía Técnica

- **Autor(es)**: *Victor Daniel Suros Cortina*

- **Fecha**: 09/03/2025

2. Introducción

Este proyecto simula las transacciones de una tienda de videojuegos mediante una API REST desarrollada en **C# y .NET**. Permite gestionar **usuarios** y **videojuegos**, asegurando la seguridad mediante **JWT, encriptación AES y hashing SHA512**. Además, cada operación exitosa se registra en **Firebase Firestore**.

3. Resumen del Sistema

El sistema maneja dos entidades principales:

- Usuario: ID, Nombre de usuario, Email (encriptado), Contraseña (hasheada), Rol (Admin o Cliente).

C/C++

```
using Google.Cloud.Firestore;

[FirestoreData]
public class Game
{
    [FirestoreProperty]
    public int Id { get; set; }

    [FirestoreProperty]
    public string Name { get; set; } // Nombre

    [FirestoreProperty]
    public string Genre { get; set; } // Genero
```

```
    [FirestoreProperty]
    public double Price { get; set; } // Precio del Juego

    [FirestoreProperty]
    public int Stock { get; set; } // Cantidad disponible

    [FirestoreProperty]
    public List<DLC> Dlcs { get; set; } = new List<DLC>(); //
List of DLCs

    // Ensure a parameterless constructor
    public Game() { }
}

[FirestoreData]
public class DLC
{
    [FirestoreProperty]
    public string Name { get; set; } // Nombre del DLC

    [FirestoreProperty]
    public double Price { get; set; } // Precio del DLC

    // Ensure a parameterless
    public DLC() { }
}
```

- Juego: ID, Nombre, Género, Precio, Stock, DLCs (nombre, precio).

C/C++

```
using Google.Cloud.Firestore;

[FirestoreData]
public class User
{
    [FirestoreProperty]
    public int Id { get; set; }

    [FirestoreProperty]
    public string UserName { get; set; } // Nombre de Usuario

    [FirestoreProperty]
    public string Email { get; set; } // Email del Usuario

    [FirestoreProperty]
    public string Password { get; set; } // Contraseña

    [FirestoreProperty]
    public string Role { get; set; } // Rol del Usuario

    // Ensure a parameterless constructor
    public User() { }
}
```

Los usuarios pueden realizar acciones según su rol. Los administradores pueden gestionar juegos y usuarios, mientras que los clientes pueden comprar juegos y administrar su cuenta.

4. Arquitectura y Tecnologías

- .NET Web API para la lógica de negocio.
- JWT para autenticación y control de roles.
- AES para encriptación de emails.
- SHA512 para hashing de contraseñas.
- Firebase Firestore para almacenamiento de datos en la nube.

5. Endpoints y Funcionalidades del API

Endpoints de Usuario

- POST /add - Crea un nuevo usuario.
- POST - Verifica credenciales y devuelve un token JWT.
- GET /get - Obtiene todos los usuarios.
- PUT /{username} - Actualiza la información de un usuario.

- PUT /role/{id} - Cambia el rol de un usuario.
- DELETE /delete/admin/{username} - Un administrador elimina un usuario.
- POST /delete/client/{username} - Un cliente elimina su cuenta (requiere credenciales).

Endpoints de Juego

- POST /add - Agrega un nuevo juego.
- POST /add-multiple - Agrega varios juegos.
- GET /{id} - Obtiene un juego por ID.
- GET - Obtiene todos los juegos.
- PUT /{id} - Actualiza un juego.
- PUT /update-multiple - Actualiza varios juegos.
- DELETE /{id} - Elimina un juego.
- DELETE /delete-multiple - Elimina varios juegos.
- PUT /buy/{id} - Simula la compra de un juego, reduciendo su stock.

6. Autenticación y Autorización

- Se usa JWT para la autenticación.
- Se implementa control de acceso por roles:
    - Admin: Puede administrar usuarios y juegos.
    - Cliente: Puede comprar juegos y gestionar su cuenta.

- Medidas de seguridad:
    - Hashing SHA512 para contraseñas.

C/C++

```
public string HashPassword(string password){
        using (SHA512 SHA512 = SHA512.Create()){
            return GetHash(SHA512, password);
        }

    }
    // AllCaps Required
    public bool VerifyPassword(string password, string
password2){
        using (SHA512 SHA512 = SHA512.Create()){
            return VerifyHash(SHA512, password, password2);
        }
    } // Llamada a los metodos de el ejercicio de SHA
```

    - Encriptación AES para emails.

C/C++

```
private void InitAes() {
        using (Aes aes = Aes.Create()) {
            key = aes.Key;
            iv = aes.IV;
        }
    }
public byte[] EncryptEmail(string email) {
        return EncryptStringToBytes_Aes(email, key, iv);
    }
```

```
    public string DecryptEmail(byte[] encryptedEmail) {
        return DecryptStringFromBytes_Aes(encryptedEmail, key,
iv);
    } // Llamada a los metodos de el ejercicio de AES
```

7. Integración con Firestore

Cada operación exitosa en la API también se almacena en Firebase Firestore para mantener la consistencia de los datos.

C/C++

```
public async Task<List<User>> GetUsers()
        {
            CollectionReference usersRef =
_firestoreDb.Collection("Users");
            QuerySnapshot snapshot = await
usersRef.GetSnapshotAsync();
            List<User> users = new List<User>();

            foreach (DocumentSnapshot doc in snapshot.Documents)
            {
                if (doc.Exists)
                {
                    users.Add(doc.ConvertTo<User>());
                }
            }
            return users;
        }

        public async Task<User> GetUser(string username)
        {
```

```
            DocumentReference usersRef =
_firestoreDb.Collection("Users").Document(username);
            DocumentSnapshot snapshot = await
usersRef.GetSnapshotAsync();

            if (snapshot.Exists)
            {
                return snapshot.ConvertTo<User>();
            }
            return null;
        }

        public async Task AddUser(User user)
        {
            DocumentReference userRef =
_firestoreDb.Collection("Users").Document(user.Id.ToString());
            await userRef.SetAsync(user);
        }

        public async Task UpdateUser(string username,
Dictionary<string, object> updatedFields)
        {
            var userRef = _firestoreDb.Collection("User");
            var query = userRef.WhereEqualTo("UserName",
username);

            // Execute the query and get the first matching
document (if it exists)
            var snapshot = await
query.Limit(1).GetSnapshotAsync();

            // Check if any document was found
            if (snapshot.Documents.Count > 0)
            {
```

```
                var docRef =
snapshot.Documents.First().Reference;
                await docRef.UpdateAsync(updatedFields); //
Return the first document found
            }


        }

        public async Task DeleteUser(string username)
        {
            var userRef = _firestoreDb.Collection("User");
            var query = userRef.WhereEqualTo("UserName",
username);

            // Execute the query and get the first matching
document (if it exists)
            var snapshot = await
query.Limit(1).GetSnapshotAsync();

            // Check if any document was found
            if (snapshot.Documents.Count > 0)
            {
                var docRef =
snapshot.Documents.First().Reference;
                await docRef.DeleteAsync(); // Return the first
document found
            }


        }

        public async Task<List<Game>> GetGames()
        {
            CollectionReference gamesRef =
_firestoreDb.Collection("Games");
```

```
            QuerySnapshot snapshot = await
gamesRef.GetSnapshotAsync();
            List<Game> games = new List<Game>();

            foreach (DocumentSnapshot doc in snapshot.Documents)
            {
                if (doc.Exists)
                {
                    games.Add(doc.ConvertTo<Game>());
                }
            }
            return games;
        }

        public async Task<Game> GetGame(string gameId)
        {
            DocumentReference gamesRef =
_firestoreDb.Collection("Games").Document(gameId);
            DocumentSnapshot snapshot = await
gamesRef.GetSnapshotAsync();

            if (snapshot.Exists)
            {
                return snapshot.ConvertTo<Game>();
            }
            return null;
        }

        public async Task AddGame(Game game)
        {
            DocumentReference gameRef =
_firestoreDb.Collection("Games").Document(game.Id.ToString());
            await gameRef.SetAsync(game);
        }
```

```
        public async Task AddMultipleGames(List<Game> games)
        {

            var batch = _firestoreDb.StartBatch();

            foreach (var game in games)
            {
                var gameRef =
_firestoreDb.Collection("Games").Document(game.Id.ToString());
                batch.Set(gameRef, game);
            }

            await batch.CommitAsync();
        }

        public async Task DeleteGame(string gameId)
        {

            var gameRef =
_firestoreDb.Collection("Games").Document(gameId);
            await gameRef.DeleteAsync();
        }

        public async Task DeleteMultipleGames(List<int> gameIds)
        {

            var batch = _firestoreDb.StartBatch();

            foreach (var gameId in gameIds)
            {
                var gameRef =
_firestoreDb.Collection("Games").Document(gameId.ToString());
                batch.Delete(gameRef);
            }
```

```
            await batch.CommitAsync();
        }

        public async Task UpdateGame(string gameId,
Dictionary<string, object> updates)
        {

            var gameRef =
_firestoreDb.Collection("Games").Document(gameId);
            await gameRef.UpdateAsync(updates);
        }
```

Cuando el método de la API es exitoso, se llama al método correspondiente en esta clase para pasarlo a Firebase.

8. Estructura del Proyecto

Carpeta: ProyectoAPI

- Controllers/ - Contiene los controladores de la API (UserController, GameController).

- Datasets/ - Archivos JSON de inicialización para Usuarios y Juegos.

- FireBase/ - Credenciales para Firestore.

- Models/ - Definición de los modelos (User, Game).

- Services/ - Contiene:
    - GameService y UserService (lógica de negocio y gestión de IDs).
    - JwtService (manejo de tokens JWT).
    - FireBaseService (sincronización con Firestore).

Carpeta: ProyectoClientes

- Aplicación C# que actúa como cliente en un modelo Cliente-Servidor.

9. Conclusiones

- El sistema permite una gestión eficiente de una tienda de videojuegos.
- Se implementaron medidas de seguridad robustas.
- Firestore permite mantener la integridad de los datos en la nube.

10. Deudas Técnicas

- No hay deudas presentes respecto al documento entregado, aunque hay ciertas cosas que podría cambiar:

    - Actualmente uso Put para todas las operaciones de actualización omitiendo las entradas de datos innecesarias donde corresponda en vez de utilizar Patch.