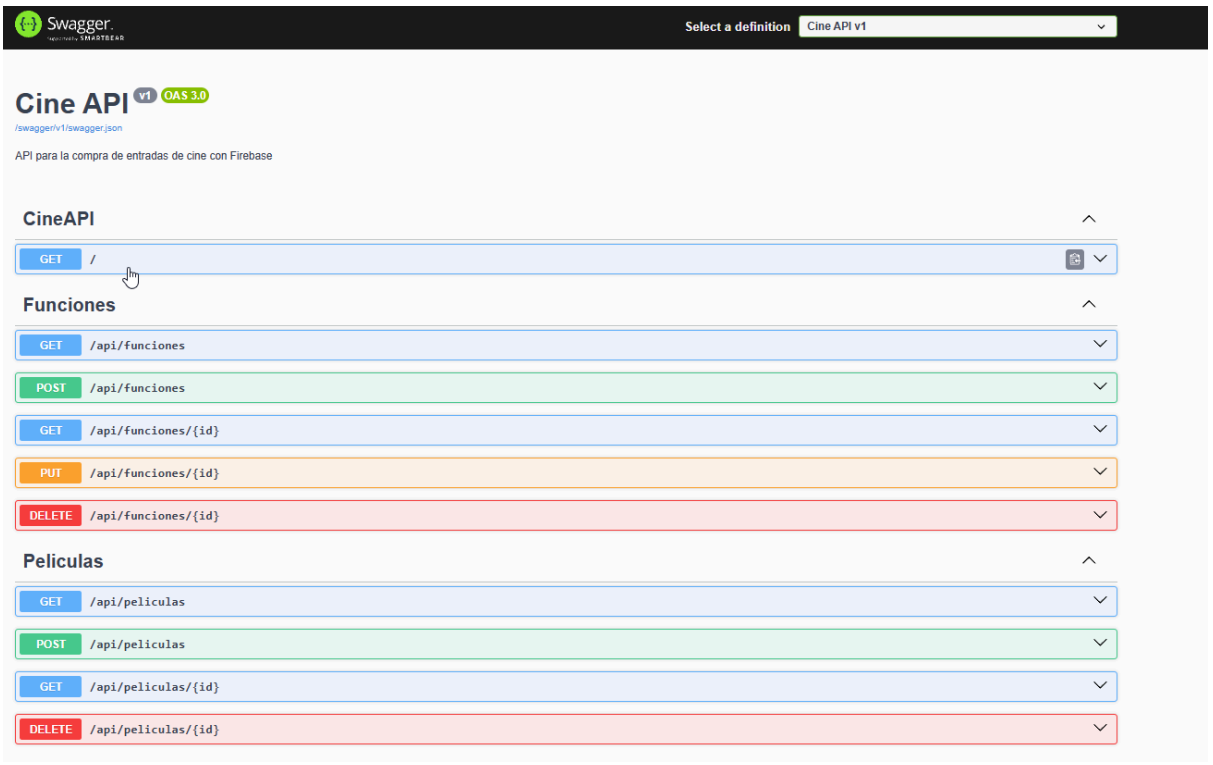
Sistema de Reservas de Cine



Sistema backend para gestionar reservas de asientos en un cine. Permite a los usuarios registrarse, ver funciones de cine, realizar reservas y gestionar el estado de estas reservas. El sistema utiliza Firebase Firestore como base de datos.

Fermin Osinaga

DAM 2ºB

Índice

1. Desarrollo

Introducción

Este proyecto es un sistema de reservas de cine con una API RESTful. En la que se puede crear un usuario, consultar las películas que hay disponibles, comprobar sus asientos y hacer reservas de los mismos.

2. Arquitectura del Proyecto

Frontend

El proyecto es solo backend, pero puede ser usado por aplicaciones web o móviles desarrolladas con frameworks como Angular, React o Blazor.

Backend

El backend está desarrollado con **ASP.NET Core Web API**, encargándose de la gestión de usuarios, funciones, reservas y autenticación. Los endpoints permiten realizar operaciones CRUD sobre la base de datos

Base de datos

Se utiliza **Firebase Firestore**, una base de datos NoSQL en la nube. La información se organiza en colecciones como:

- **Usuarios**: Contiene datos de los usuarios registrados.
- **Funciones**: Contiene información sobre las películas y sus horarios.
- **Reservas**: Almacena las reservas realizadas por los usuarios.
- **Asientos**: Contiene información sobre los asientos disponibles y no disponibles.
- **Salas**: Contiene la información sobre las salas en las que se muestran las funciones.
- **Películas**: Contiene la información relacionada con cada película.

Flujo de Datos

1. Los clientes envían solicitudes HTTP (GET, POST, PUT, DELETE) a la API.
2. La API valida los datos y realiza operaciones en Firebase Firestore.
3. La API responde con la información solicitada o con un mensaje de error si la solicitud no es válida.

3. Tecnologías Utilizadas

- **ASP.NET Core:** El framework de desarrollo para crear la API RESTful.
- **Firebase Firestore:** Base de datos NoSQL que se usa para almacenar los datos del sistema en la nube.
- **Swagger:** Herramienta para la documentación de la API, accesible desde `/swagger` cuando se ejecuta la API.
- **BCrypt**: Encriptación segura de contraseñas.

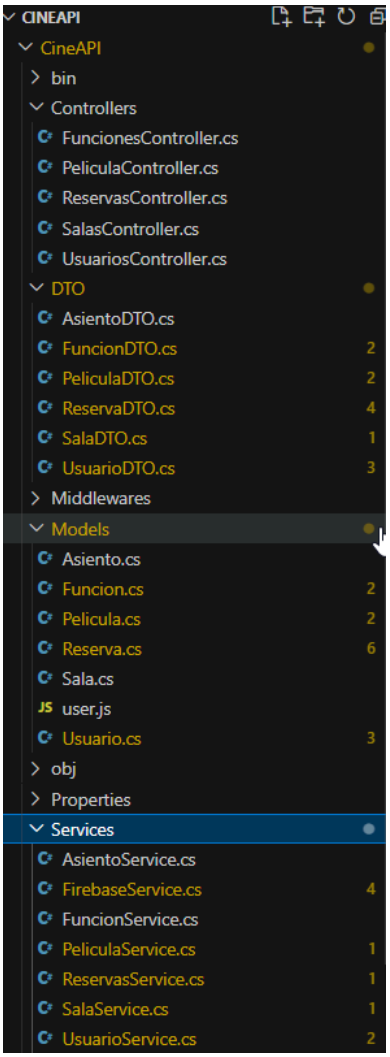
4. Estructura del Código

Descripción General

- **Controllers**: Manejan las solicitudes HTTP.
- **Services**: Contienen la lógica de negocio.
- **Model**: Es la información que se almacena de cada clase.
- **DTO**: Contiene la información que se usa en la API. (Data transfer object.)

Operaciones CRUD

- **Crear (POST)**: Creación de usuarios, reservas…
- **Leer (GET)**: Consulta de películas, usuarios, reservas…
- **Actualizar (PUT)**: Modificación de información de usuarios, reservas, funciones…
- **Eliminar (DELETE)**: Eliminación de reservas, usuarios, películas...

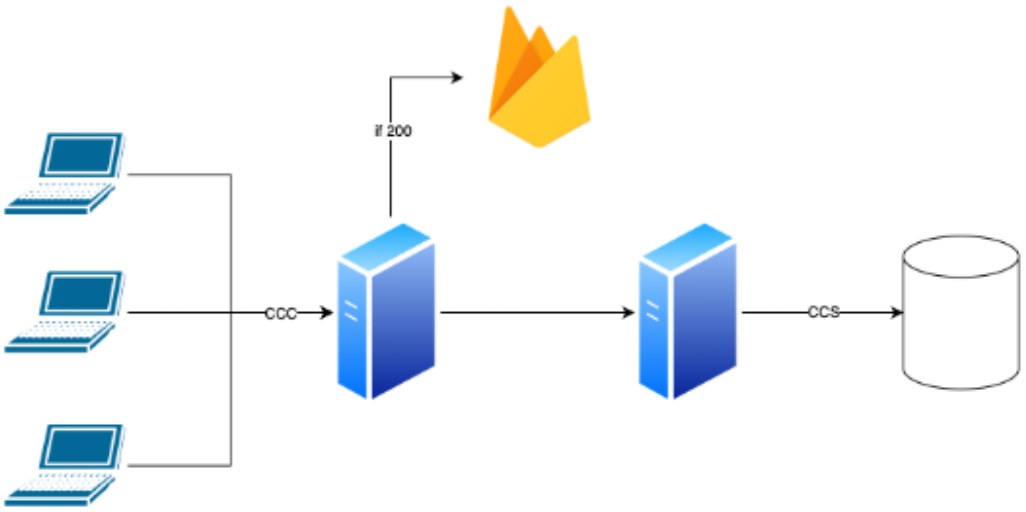


5. Pruebas y Validaciones

Pruebas de Integración

Se probaron los endpoints con **Swagger** y **Postman**, validando la correcta comunicación entre la API y Firebase Firestore.

6. Arquitectura del Proyecto



1. **Clientes (Frontend)**: Los clientes (una instancia o más del visual studio) hacen **consultas HTTP** (normalmente usando el protocolo REST) hacia el servidor que este se conecta con la (API) para realizar operaciones como crear, leer, actualizar o eliminar datos.

2. **Servidor (Backend)**: El servidor se comunica con la **API** que es responsable de procesar las solicitudes que provienen de los clientes. La API contiene los **endpoints** que exponen las funcionalidades que los clientes pueden usar (por ejemplo, registrar usuarios, hacer reservas, obtener información sobre las películas, etc.). La API también puede **validar** y **autenticar** las solicitudes antes de enviar las consultas a la base de datos.
3. **Base de Datos (Firebase Firestore)**: La base de datos es donde se almacena toda la información (como usuarios, reservas, salas, etc.). Firebase Firestore es una base de datos NoSQL basada en documentos, y la API hace consultas a esta base de datos cuando necesita almacenar o recuperar datos.

Resumidamente:

1. El cliente **envía** una solicitud HTTP (por ejemplo, una solicitud POST para crear un usuario).
2. El servidor **recibe** y procesa la solicitud.
3. El servidor valida la solicitud asegurándose de que los datos sean correctos.
4. El servidor realiza una consulta **a la base de datos Firestore** (por ejemplo, para almacenar un nuevo usuario o consultar datos existentes).
5. El servidor **devuelve una respuesta** al cliente, que puede ser una confirmación de que la operación fue exitosa o un mensaje de error si hubo algún error.

7. Conclusiones

Este proyecto ha sido un reto ya que he tenido que implementar diferentes sistemas y diferentes asignaturas, pero he conseguido hacer que sirva para una gestión de reservas de cines. Al integrar Firebase Firestore y no tener una base de datos local, hace que sea todo más fácil.

He aprendido a generar una API desde 0, conectar Firebase al proyecto y hacer uso de él, y también a encriptar contraseñas con BCrypt.

8. Encriptación

Socket

El cliente se conecta al servidor utilizando un socket TCP, lo que establece un canal de comunicación.

Se establece una conexión asincrónica entre el cliente y el servidor.

Intercambio de Claves:

El servidor genera una clave pública **RSA** que se utiliza para encriptar la clave simétrica **AES**.
El cliente recibe la clave pública **RSA** del servidor.
El cliente genera una clave simétrica **AES** (de 16 bytes para AES-128), la encripta usando la clave pública RSA del servidor y la envía de vuelta al servidor.

Cifrado y Descifrado de Mensajes:

Todos los mensajes entre el cliente y el servidor son cifrados utilizando AES.
El Initialization Vector (IV) se genera aleatoriamente para cada mensaje y se incluye en el mensaje cifrado.
El mensaje cifrado se transmite como un bloque que incluye tanto el IV como los datos cifrados.
Desencriptación de Mensajes:

El servidor utiliza la clave AES recibida para descifrar los mensajes.
El cliente, al recibir un mensaje cifrado, utiliza la misma clave AES para descifrarlo.

9. Deudas Técnicas

Algunas mejoras pendientes incluyen:

- Implementación de WebSockets para notificaciones en tiempo real.
- Optimización del manejo de errores.
- Mejora en la autenticación y seguridad de la API (uso de JWT).

10. Demostración (Video)

Api
https://youtu.be/A9wP6eBmrGo

Cliente - Servidor
https://youtu.be/Gtm7PLKaVuY