



PROYECTO SGE

2ª EVALUACIÓN

CFGS Desarrollo de Aplicaciones Multiplataforma
Informática y Comunicaciones

APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

Año: 2025

Fecha de presentación: 10/2/2025

Nombre y Apellidos: *Hugo del Rey Holgueras*

Email: *hugo.reyhol@educa.jcyl.es*

Índice

1. Introducción.....	4
2. Estado del arte.....	4
2.1 Definición de arquitectura de microservicios.....	4
2.2 Definición de API.....	4
2.3 Estructura de una API.....	5
2.4 Formas de crear una API en python.....	5
3. Descripción general del proyecto.....	6
3.1 Objetivos.....	6
3.2 Entorno de trabajo.....	6
- Lenguajes de programación:.....	6
- Frameworks:.....	6
- IDEs:.....	7
- Bases de datos:.....	7
- Docker:.....	7
4. Documentación técnica.....	8
4.1 Análisis de sistema.....	8
- Aplicación:.....	8
- API:.....	8
- Autenticación:.....	9
- Manejo de errores:.....	9
- Pruebas unitarias:.....	9
4.2 Diseño de la base de datos.....	10
4.3 Implementación.....	11
4.4 Pruebas.....	20
4.5 Despliegue de la aplicación.....	20
5. Manuales.....	20
5.1 Manual de usuario.....	20

PROYECTO FINAL 2ª EVALUACIÓN – APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

Inicio de sesión.....	20
Menús.....	21
Menú amplio:.....	21
Menú compacto:.....	21
Pantalla de inicio.....	21
Pantalla de búsqueda.....	22
Pantalla de estadísticas o perfil.....	22
Pantalla de detalles.....	23
5.2 Manual de instalación.....	24
1º Instalar lenguajes necesarios:.....	24
2º Instalar aplicaciones para lanzar los programas:.....	24
3º Descargar el código:.....	24
4º Crear la base de datos PostgreSQL:.....	25
5º Lanzar la API:.....	25
6º Abrir la aplicación:.....	26
6. Conclusiones y posibles ampliaciones.....	26
7. Bibliografía.....	26

1. Introducción

En este proyecto se ha realizado una API en python usando FastAPI, es posible interactuar con esta API mediante una aplicación hecha con Flutter, la cual permite exportar a múltiples plataformas, de esta forma los datos de nuestra aplicación se centralizarán en un servidor y podremos acceder a ellos usando un cliente de escritorio, móvil o web.

2. Estado del arte

2.1 Definición de arquitectura de microservicios

La arquitectura de microservicios es un enfoque de desarrollo de software en el que una aplicación se construye como un conjunto de servicios pequeños, independientes y especializados. Cada uno de estos microservicios se encarga de una función específica dentro de la aplicación y se comunica con los demás mediante API, normalmente HTTP/REST, gRPC o mensajería.

Principales características de la arquitectura de microservicios:

1. **Independencia:** Cada microservicio opera de manera autónoma y puede ser desarrollado, desplegado y escalado de forma independiente.
2. **Desacoplamiento:** Al estar separados, los cambios en un microservicio no afectan directamente a los demás, siempre que se respeten las interfaces de comunicación.
3. **Escalabilidad:** Se pueden escalar solo los microservicios que lo necesiten en lugar de toda la aplicación.
4. **Desarrollo ágil:** Permite que diferentes equipos trabajen en distintos microservicios al mismo tiempo, lo que acelera el desarrollo.
5. **Tecnologías diversas:** Cada microservicio puede estar desarrollado en un lenguaje o tecnología distinta, según lo que mejor se adapte a su función.
6. **Tolerancia a fallos:** Un fallo en un microservicio no necesariamente afecta a toda la aplicación, lo que mejora la resiliencia del sistema.

2.2 Definición de API

Una API es un conjunto de reglas y definiciones que permite que dos sistemas de software se comuniquen entre sí. Actúa como un intermediario que facilita el intercambio de datos y funcionalidades entre aplicaciones, servicios o dispositivos.

Características principales de una API:

1. **Interoperabilidad:** Permite que diferentes sistemas, incluso si están desarrollados en lenguajes distintos, se comuniquen sin problemas.
2. **Abstracción:** Oculta los detalles internos de un sistema, exponiendo solo las funciones necesarias para la interacción.
3. **Estandarización:** Se basa en protocolos y formatos como HTTP, JSON, XML o gRPC.
4. **Seguridad:** Puede incluir autenticación y autorización para controlar el acceso a los datos y servicios.

2.3 Estructura de una API

La API se compone de las siguientes partes:

1. **Endpoint:** La URL a la que se envían las solicitudes para acceder a los recursos, es la parte que sigue al dominio, en el caso de *localhost:8000/docs* el endpoint sería *docs*.
2. **Métodos:** Son las acciones que se pueden utilizar sobre un recurso, los más comunes son: GET para obtener un recurso, POST para crear un recurso, PUT para modificar completamente un recurso, PATCH para modificar parcialmente un recurso, DELETE para eliminar un recurso.
3. **Parámetros:** Permiten personalizar las solicitudes a la API, se pueden colocar siguiendo el endpoint.
4. **Cabeceras o Headers:** Son datos enviados con la solicitud para proporcionar información adicional, como autenticación y formato de respuesta.
5. **Cuerpo o Body:** Datos extra que se pueden enviar al hacer una petición.
6. **Respuesta de la API:** Son los datos que devuelve la API como respuesta a nuestra petición.

2.4 Formas de crear una API en python

Las dos opciones más populares para crear una API son FastAPI y Flask, pero para el proyecto se ha elegido FastAPI ya que tiene las siguientes ventajas sobre Flask:

1. **Velocidad:** FastAPI es más rápida que Flask.
2. **Tipado:** FastAPI permite el uso de Pydantic y typing para realizar validaciones automáticas.
3. **Autodocumentación:** FastAPI nos da la posibilidad de generar documentación usando Swagger o Redoc.
4. **Asíncrona:** FastAPI es compatible con *async* y *await*, lo que permite realizar varias funciones simultáneamente.

FastAPI es el framework que se usa para construir la API la cual se ejecutará en un servidor Uvicorn.

La estructura básica de una API construida con FastAPI es la siguiente:

- **Routers:** Son los encargados de definir los endpoints de la API y la lógica que se ejecutará cuando se llame a estos.
- **Schemas:** Son las representaciones de los datos que se van a enviar o recibir desde la API.
- **Models:** Son las representaciones de las tablas de la base de datos.
- **JWT:** Es el encargado de proteger endpoints para que solo puedan usarlos usuarios en específico.
- **Test:** Son comprobaciones mediante las cuales nos aseguramos de que el código funcione como esperamos.

3. Descripción general del proyecto

3.1 Objetivos

El objetivo de este proyecto es conectarnos a una API con una aplicación que hemos desarrollado previamente, adquiriendo en el proceso conocimientos de backend, frontend y seguridad. La aplicación tiene que poder realizar todas las operaciones CRUD con la API de forma segura y funcional.

3.2 Entorno de trabajo

- Lenguajes de programación:

Python: Para programar la lógica de la API he usado python 3.12.3, gracias a su versatilidad y sencillez cuenta con una gran comunidad y una amplia variedad de bibliotecas, lo que facilita el desarrollo y aprendizaje de este lenguaje.

Dart: Para construir la aplicación que consume la API se ha usado dart 3.6.0, un lenguaje de programación desarrollado por Google optimizado para la creación de aplicaciones multiplataforma, tiene una sintaxis similar a Java pero modernizada, lo que lo hace un lenguaje sencillo de aprender.

- Frameworks:

FastAPI: Para construir la API se ha usado FastAPI 0.115.8, un framework web para construir APIs en python. Lo he seleccionado ya que es fácil de aprender, rápido y escalable.

Flutter: Para crear la aplicación he usado Flutter 3.27.1, un framework desarrollado por Google que se apoya sobre el lenguaje dart para crear crear aplicaciones multiplataforma a partir de un código único.

- IDEs:

PyCharm: Para la programación en python uso PyCharm 2024.3.1.1 Community Edition, un IDE de código abierto creado por JetBrains que ofrece muchas herramientas que facilitan el desarrollo en python.

Intelij IDEA: Para la programación en dart uso IntelliJ IDEA 2024.3.1.1 Community Edition, un IDE de código abierto creado por JetBrains diseñado para el desarrollo en java, para hacerlo compatible con dart y Flutter he instalado dos plugins llamados "Dart" y "Flutter", desarrollados por JetBrains y Google respectivamente, estos añaden todas las herramientas necesarias para crear una aplicación desde cero.

Android Studio: Para poder testear la aplicación en un entorno móvil he utilizado Android Studio Ladybug 2024.2.1 Patch 3, un IDE desarrollado por Google con base en IntelliJ IDEA para la creación de aplicaciones Android. De este IDE he usado principalmente su emulador Android ya que al abrir un proyecto hecho en Flutter con este IDE puedes compilarlo e instalarlo en el emulador automáticamente.

- Bases de datos:

PostgreSQL: La base de datos en la que almaceno la información que utiliza la API es PostgreSQL, un sistema de gestión de bases de datos relacional de código abierto, destaca por su robustez y escalabilidad.

pgAdmin: Para administrar gráficamente la base de datos PostgreSQL he usado pgAdmin, la cual permite gestionar la base de datos y ejecutar consultas SQL.

- Docker:

Docker: Para desplegar la base de datos y su gestor web he usado Docker, una plataforma de software que permite desarrollar, empaquetar y ejecutar aplicaciones dentro de contenedores, entornos ligeros, portátiles y consistentes que encapsulan todo lo que una aplicación necesita para funcionar, incluyendo el código, las bibliotecas, las dependencias y el sistema operativo, de manera aislada del resto del sistema, esto nos permite ejecutar una aplicación en una gran variedad de sistemas operativos y equipos sin preocuparnos por las distintas alteraciones que estos puedan causar en nuestra aplicación.

Docker Desktop: Para manejar los contenedores he usado Docker Desktop, una herramienta que permite definir y gestionar aplicaciones compuestas por múltiples contenedores.

4. Documentación técnica

4.1 Análisis de sistema

- Aplicación:

La aplicación que he desarrollado es un gestor de colecciones de videojuegos, permite a los distintos usuarios de la aplicación agregar a su colección personal varios juegos pudiendo organizarlos en distintas categorías como "pendiente" o "completado", asignarles una nota y tiempo total jugado, a parte de esto también se puede visualizar información sobre el juego como una descripción de este o su desarrollador.

La aplicación permite registrarse o iniciar sesión en ella, una vez dentro se divide en tres pestañas principales, el inicio para ver nuestra colección actual, la búsqueda para encontrar algún juego que queramos añadir o sobre el que queramos ver información y las estadísticas u opciones en la que se mostrará distinta información sobre el usuario como de cuantos juegos dispone en su colección.

Para que la aplicación funcione correctamente tiene que poder realizar peticiones y procesar las respuestas de la API para obtener la lista de juegos, insertar y autenticar usuarios y hacer un CRUD completo sobre la tabla que relaciona los juegos con los usuarios.

Como ya le incorpore una función de cifrado a la aplicación la contraseña se cifra antes de ser enviada a la API.

- API:

La API está dividida en tres partes:

- **Usuarios (/user):** Es la parte encargada de procesar los inicios de sesión y los registros de nuevos usuario, el login (post: /login) permite obtener un token con una duración de 30 minutos a partir de un formulario de inicio con el nombre y contraseña previamente cifrada del usuario y el registro (post: /insert) crea un nuevo usuario a partir de un JSON y devuelve el token de ese usuario para poder iniciar sesión automáticamente después de registrarse.
- **Juegos (get /game):** La aplicación recibe la lista de juegos existentes de aquí, devuelve una lista de JSON con la información de los distintos juegos ordenados por su título.
- **Juegos de usuario (/user_game):** Es la encargada de crear (post: /) las relaciones entre las tablas, actualizarlas (patch /{idGame}), borrarlas (delete /{idGame}) y leerlas (get /), en este

PROYECTO FINAL 2ª EVALUACIÓN - APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

último caso devuelve una lista de JSON con la información de cada juego de los usuarios. Todos estos métodos requieren autenticación con el token que se genera al iniciar sesión y solo interactúan con los juegos del usuario al que pertenece el token.

- Autenticación:

Para autenticar al usuario se usan las librerías *python-jose[cryptography]* y *python-multipart*, al iniciar sesión o registrarse se genera un token que se devuelve y almacena en la aplicación, cuando se requiere acceder a una ruta protegida este token se agrega como un header a la petición a la API y una vez en ella se comprueba que sea correcto y usa su información para saber que usuario a realizado cada petición, si la comprobación no es correcta la API lanza el error 401 para proteger los datos de usuarios de otros no autorizados.

- Manejo de errores:

Cuando la API no es capaz de responder a una petición devuelve una respuesta con un código distinto a 200 el cual identifica el error encontrado, los *services* de la aplicación, encargados de comunicarse con la API, se encargan de procesar estos errores para que la aplicación funcione correctamente.

Los códigos que he usado han sido:

- **200:** Para indicar que la petición se ha resuelto adecuadamente.
- **401:** Cuando se intenta acceder a un endpoint protegido sin la autenticación necesariamente
- **404:** Cuando se intenta acceder a un recurso que no existe

La aplicación está preparada para lidiar con estos casos comunes y manda información en un formato preestablecido en la API para evitar otros errores que podrían suceder al no poder interpretar adecuadamente la información.

- Pruebas unitarias:

Para comprobar que la API y la aplicación funcionasen como esperaba he creado varios test automáticos.

En la aplicación he comprobado que se obtengan adecuadamente los juegos desde el service.

```
void main() {  
    test("Get games", () async {  
        final games = await GameService.getGames();  
  
        print(games.length);  
  
        for (var game in games) {  
            print(game.title);  
        }  
    });  
}
```

En la API he usado las librerías *pytest* y *httpx* para las comprobaciones. La primera lee todos los juegos y comprueba que la respuesta sea 200 y que el total de juegos sean 3, la cantidad que he introducido en la base de datos para probarla. La segunda hace login con un usuario predefinido y verifica que se devuelva un token con un formato correcto. La tercera intenta borrar el juego de un usuario sin autenticarse y espera recibir un código 401 para asegurarnos de que la protección esté funcionando correctamente.

```
def test_read_games():  # Hugo
    response = client.get("/game/")
    assert response.status_code == 200
    assert len(response.json()) == 3

def test_login():  # Hugo
    data = {
        "grant_type": "password",
        "username": "test",
        "password": "test",
        "scope": "",
        "client_id": "string",
        "client_secret": "string"
    }
    response = client.post(url="/user/login", data=data)
    assert response.status_code == 200
    assert str(response.json()["token"]).startswith("Bearer")

def test_user_games():  # Hugo
    response = client.delete("/user_game/1")
    assert response.status_code == 401
```

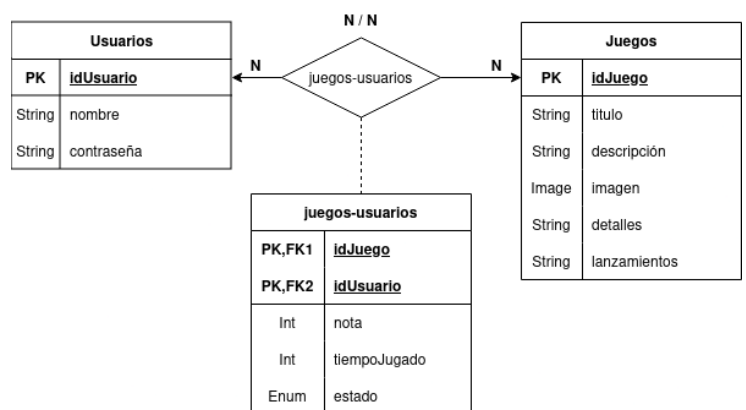
4.2 Diseño de la base de datos

La base de datos consta de tres tablas:

Usuarios (Users): Representa los usuario que utilizan la aplicación con un nombre único que los representa y su contraseña cifrada usando SHA256.

Juegos (Games): Guarda toda la información de cada juego, los campos principales son su título para ordenarlos alfabéticamente al hacer las peticiones y guardan una imagen en formato binario.

Juegos_Usuarios (Users_Games): Es una tabla que tiene como clave primaria las claves primarias de las anteriores tablas, es la encargada de representar la relación entre las otras tablas, además guarda información extra como la nota que le ha dado el usuario o en que categoría se encuentra.



4.3 Implementación

La API se divide en varias partes como se puede ver en la imagen de la derecha, la API comienza cuando se ejecuta **main.py**, en este archivo se llama a todas las otras funciones que se encargan de crear la base de datos con

```
from fastapi import FastAPI
import uvicorn
from app.db.db_initialize import initialize_database
from app.routers import user, game, user_game
from app.db.database import Base, engine

def create_tables(): 1 usage 1 Hugo
    Base.metadata.create_all(bind=engine)

create_tables()
initialize_database()

app: FastAPI = FastAPI()
app.include_router(user.router)
app.include_router(game.router)
app.include_router(user_game.router)

if __name__ == '__main__':
    uvicorn.run(app="main:app", port=8000, reload=True)
```

create_tables() y **initialize_database()** después se instancia **FastAPI** y se le configura los distintos *routers*, los endpoints que la API a los que se puede conectar un usuario, esta API se ejecuta en un servidor **uvicorn** en el que podemos definir el puerto en el que escuchará las peticiones. Las librerías necesarias para esta primera parte son *FastAPI* y *Uvicorn*.

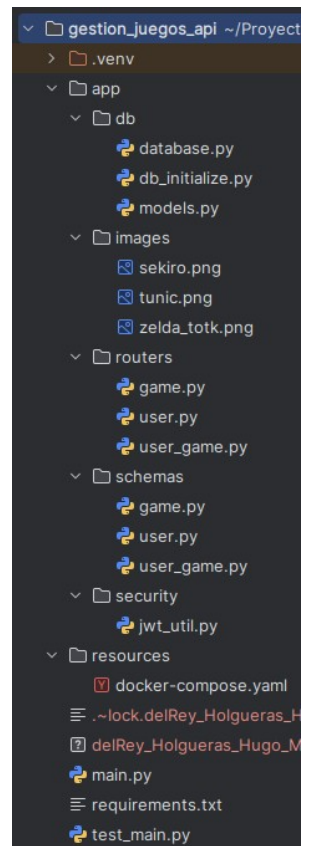
Lo siguiente es la creación automática de la base de datos, para ello se utiliza **database.py**, en este archivo se define la URL donde se encuentra la base de datos y se crea la conexión con esta, también define la función **get_db()** la cual devuelve una instancia de la base de datos y la cierra cuando se deja de usar. La librería que hace esto posible es *SQLAlchemy*.

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = "postgresql://dbuser:dbpass@localhost:5432/gestion_juegos"

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
Base = declarative_base()

def get_db(): 12 usages 1 Hugo
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```



PROYECTO FINAL 2ª EVALUACIÓN – APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

Lo siguiente es definir los modelos que se van a utilizar, estos representan las tablas que se crearán en la base de datos cuando se instancia, para esto también usamos *SQLAlchemy*.

Los modelos tienen una propiedad `__tablename__` con el nombre que tendrá la tabla en la base de datos cuando se cree, las variables que definamos tienen que ser instancias de **Column** que representan las columnas de las tablas y una tiene que tener **primary_key=True** para asignarla como clave primaria. En el modelo `UserGame` las claves foráneas se definen dentro de las `Column` con su comportamiento, en este caso al borrar un elemento de las tablas referenciadas también se borrarán los elementos de `UserGame` que tuviesen una relación con él.

```
from app.db.database import Base
from sqlalchemy import Column, Integer, String, LargeBinary, DateTime
from sqlalchemy.schema import ForeignKey
from sqlalchemy.orm import relationship

class User(Base): 19 usages  ⬆️ Hugo
    __tablename__ = "Users"
    idUser = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, unique=True)
    password = Column(String)
    userGame = relationship( argument: "UserGame", backref="Users", cascade="delete,merge")

class Game(Base): 9 usages  ⬆️ Hugo
    __tablename__ = "Games"
    idGame = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    description = Column(String)
    image = Column(LargeBinary)
    details = Column(String)
    releases = Column(String)
    userGame = relationship( argument: "UserGame", backref="Games", cascade="delete,merge")

class UserGame(Base): 10 usages  ⬆️ Hugo
    __tablename__ = "User_Games"
    idGame = Column(Integer, ForeignKey( column: "Games.idGame", ondelete="CASCADE"), primary_key=True)
    idUser = Column(Integer, ForeignKey( column: "Users.idUser", ondelete="CASCADE"), primary_key=True)
    score = Column(Integer, nullable=True)
    timePlayed = Column(Integer)
    gameState = Column(String)
    ⚡ lastChange = Column(DateTime)
```

PROYECTO FINAL 2ª EVALUACIÓN - APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

El siguiente archivo es **db_initialize.py**, este se encarga de introducir los datos predefinidos en la base de datos cuando esta se crea, tiene definida una lista de juegos y un usuario para realizar los test, luego comprueba si no existen y en caso de no hacerlo los inserta.

Para cargar las imágenes hay que abrirlas como un archivo y leerlas para obtener los bytes.

El usuario test se crea sin cifrar su contraseña, lo que hace imposible iniciar sesión como él desde la aplicación porque la contraseña introducida se cifraría antes de enviarse y se detectará como una distinta.

Después de introducir todos los datos se hace un **commit** para confirmar los cambios y se cierra la conexión con la base de datos.

```
def initialize_database():
    db = SessionLocal()
    try:
        predefined_games = [
            Game(
                title="The Legend of Zelda: Tears of the Kingdom",
                description="The Legend of Zelda: Tears of the Kingdom is the",
                details="...",
                releases="Nintendo Switch: 2023-5-12"
            ), Game(
                title="Tunic",
                description="Tunic is an action adventure about a tiny fox in",
                details="...",
                releases="..."
            ), Game(
                title="Sekiro",
                description="Enter a dark and brutal new gameplay experience",
                details="...",
                releases="..."
            )
        ]

        with open("app/images/zelda_totk.png", "rb") as file:
            predefined_games[0].image = file.read()

        with open("app/images/tunic.png", "rb") as file:
            predefined_games[1].image = file.read()

        with open("app/images/sekiro.png", "rb") as file:
            predefined_games[2].image = file.read()

        for game in predefined_games:
            if not db.query(Game).filter(Game.title == game.title).first():
                db.add(game)

        user = User(username="test", password="test")
        if not db.query(User).filter(User.username == user.username).first():
            db.add(user)

        db.commit()
    finally:
        db.close()
```

Para representar los datos que se reciben y envían desde la API se usan los **Schemas**, como modelo con el que se interactúa tiene uno o más schemas. Para crear los schemas se usa la librería *pydantic*.

Los usuarios tienen dos schemas, ambos comparten el nombre y la contraseña, pero **UserResponse** también añade el id del usuario ya que es un schema que se usa en pruebas del token y de esta forma podemos verificar que el id devuelto es el mismo que el del usuario en la base de datos. **UserCreate** se usa cuando necesitamos registrar un nuevo usuario.

```
class UserCreate(BaseModel):
    username: str
    password: str

class UserResponse(BaseModel):
    idUser: int
    username: str
    password: str
```

PROYECTO FINAL 2ª EVALUACIÓN - APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

Los juegos tienen un solo schema, en este se incluye toda la información de la base de datos. Para poder enviar la imagen a través de la API es necesario enviarla como una cadena de texto en formato base64, para hacer esto hay que crear una configuración en el schema que codifica la imagen y devuelve una instancia del schema a partir de los valores del model.

La librería usada para codificar la imagen es *base64*, esta es interna de python por lo que no hace falta instalarla.

```
class GameResponse(BaseModel): 4 usages 1 Hugo
    idGame: int
    title: str
    description: str
    image: str
    details: str
    releases: str

    model_config = ConfigDict(from_attributes=True)

    # Convierte la imagen binaria a texto plano
    @classmethod 1 usage 1 Hugo
    def from_game_model(cls, game):
        image_base64 = base64.b64encode(game.image).decode("utf-8")
        return cls.model_validate({
            "idGame": game.idGame,
            "title": game.title,
            "description": game.description,
            "image": image_base64,
            "details": game.details,
            "releases": game.releases,
        })
```


PROYECTO FINAL 2ª EVALUACIÓN - APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

Para UserGame he hecho tres schemas, en estos he tenido que incluir algunos campos **Optional**, esto permite que esos campos puedan valores recibir nulos.

UserGameInsert es el usado cuando se quiere crear un UserGame nuevo, el id del usuario no es necesario ya que se obtiene al comprobar el token

UserGameUpdate se usa cuando queremos actualizar un UserGame, sus campos no incluyen los id de juego y usuario ya que no queremos que esos campos cambien y el resto son Optional, de esta forma solo se envía el campo a actualizar.

UserGameResponse genera el UserGame que se va a devolver cuando se hace un get, en este schema también hay que crear una configuración, en este caso convierte la fecha guardada en la base de datos a un int.

```
class UserGameInsert(BaseModel): 2 usages 1 Hugo
    idGame: int
    score: Optional[int]
    timePlayed: int
    gameState: str
    lastChange: datetime

class UserGameResponse(BaseModel): 4 usages 1 Hugo
    idUser: int
    idGame: int
    score: Optional[int]
    timePlayed: int
    gameState: str
    lastChange: int

    model_config = ConfigDict(from_attributes=True)

    @classmethod 1 usage 1 Hugo
    def convert_timestamp(cls, user_game):
        return cls.model_validate({
            "idUser": user_game.idUser,
            "idGame": user_game.idGame,
            "score": user_game.score,
            "timePlayed": user_game.timePlayed,
            "gameState": user_game.gameState,
            "lastChange": int(user_game.lastChange.timestamp() * 1000)
        })

class UserGameUpdate(BaseModel): 2 usages 1 Hugo
    score: Optional[int] = None
    timePlayed: Optional[int] = None
    gameState: Optional[str] = None
    lastChange: Optional[datetime] = None
```

Para poder hacer peticiones a la API hay que configurar las rutas a las que enviar las solicitudes, estas se definen en los **routers**.

El primer router es el de usuarios, un router tiene esta estructura, un **prefix**, que es el comienzo que va a tener el endpoint, y unos **tags**, que sirven para identificar el router.

```
router = APIRouter(
    prefix="/user",
    tags=["Users"]
)
```

Después del router hay que definir las distintas funciones que va a realizar la API, esta primera es la encargada de hacer que un usuario pueda iniciar sesión, las funciones del router siempre tienen el decorador `@router.<método>`, el método le dice a la API que tipo de petición espera, dentro de este decorador se escribe como un string la parte final del endpoint y adicionalmente podemos agregar *responses*, estas sirven para generar la documentación automática de Swagger, es un diccionario en el que se pasa como llave el código de respuesta y como valor otro diccionario con una descripción

PROYECTO FINAL 2ª EVALUACIÓN - APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

de ese código y opcionalmente también se puede añadir un campo *content* al diccionario para especificar que tipo de body devuelve.

La función tiene como parámetros **form_data** con el nombre y contraseña que haya introducido el usuario y una **db** que llama al método *get_db* que definimos en el archivo *database.py* para obtener la base de datos.

En el cuerpo de la función busca el usuario en la base de datos el nombre, si no lo encuentra devuelve el error *404* y si la contraseña es incorrecta el *401*, si las credenciales son correctas se genera un token y se devuelve.

```
@router.post(path: "/login", responses={ 1 Hugo
    404: {...},
    401: {...}
})
def get_user(form_data: OAuth2PasswordRequestForm = Depends(), db:Session=Depends(get_db)):
    user = db.query(User).filter(User.username == form_data.username).first()
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    if form_data.password != user.password:
        raise HTTPException(status_code=401, detail="Wrong credentials")

    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(data={"sub": user.username}, expires_delta=access_token_expires)
    return {
        "idUser": user.idUser,
        "token": f"Bearer {access_token}",
        "token_type": "bearer"
    }
```

La función para crear un usuario es similar al método anterior, los cambios destacables son que la función recibe como parámetro uno de los schemas que definimos anteriormente, si el usuario existe devuelve el código 400, sino devuelve el id que le ha asignado al insertarlo en la base de datos y un token para que pueda iniciar sesión.

```
@router.post(path: "/insert", responses={ 1 Hugo
    400: {...}
})
def insert_user(user: UserCreate, db:Session=Depends(get_db)):
    existing_user = db.query(User).filter(User.username == user.username).first()
    if existing_user:
        raise HTTPException(status_code=400, detail="The user already exists")

    new_user = User(
        username = user.username,
        password = user.password
    )
    db.add(new_user)
    db.commit()
    db.refresh(new_user)

    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(data={"sub": user.username}, expires_delta=access_token_expires)

    return {"idUser": new_user.idUser, "token": f"Bearer {access_token}", "token_type": "bearer"}
```


PROYECTO FINAL 2ª EVALUACIÓN - APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

El router para los juegos tiene la misma estructura que el de usuario, solo se cambia el prefijo y el tags.

El método get de este router es más sencillo, en el decorador se añade el parámetro **response_model** para indicar que se va a devolver, se hace la consulta a la base de datos ordenándolos por título y antes de devolver los juegos se llama al método del schema para convertir la imagen a base64.

```
router = APIRouter(
    prefix="/game",
    tags=["Games"]
)

@router.get(path="/", response_model=List[GameResponse]) # Hugo
def get_games(db:Session=Depends(get_db)) -> List[GameResponse]:
    games = db.query(Game).order_by(asc(Game.title)).all()
    return [GameResponse.from_game_model(game) for game in games]
```

El router para UserGames es el siguiente, solo cambia el prefix y las tags.

```
router = APIRouter(
    prefix="/user_game",
    tags=["UserGames"]
)
```

El router UserGame tiene cuatro funciones, la primera es para insertar UserGames nuevos, se obtiene como parametro el UserGame como el schema para insertar, la base de datos y **current_user** se obtiene de la función **get_current_user** que se verá en después, es el método encargado de la autenticación.

En el cuerpo se crea un modelo de UserGame a partir de los parámetros de la función y se añade a la base de datos.

```
@router.post(path="/", responses={...})
def insert_user_game(user_game: UserGameInsert, db: Session = Depends(get_db), current_user: User = Depends(get_current_user)):
    new_user_game = UserGame(idUser=current_user.idUser, **user_game.model_dump(exclude_unset=True))
    db.add(new_user_game)
    db.commit()
    db.refresh(new_user_game)
```

La siguiente función es para obtener los UserGames de la base de datos, se hace una consulta a partir del id del usuario que llama a la función y se devuelve una lista de los juegos con el timestamp convertido en un entero a través de la configuración creada en el schema.

PROYECTO FINAL 2ª EVALUACIÓN - APLICACIÓN CRUD CON FASTAPI Y CLIENTE MÓVIL

```
@router.get(path="/", response_model=List[UserGameResponse], responses={...})
def get_user_games(current_user: User = Depends(get_current_user), db: Session = Depends(get_db)) -> List[UserGameResponse]:
    user_games = db.query(UserGame).filter(UserGame.idUser == current_user.idUser).all()
    return [UserGameResponse.convert_timestamp(user_game) for user_game in user_games]
```

Esta función es la encargada de actualizar un UserGame, para ello el endpoint acaba con el nombre de un parámetro escrito entre llaves, esto es para poder pasar el id del juego desde el endpoint.

Dentro del cuerpo de la función se busca el UserGame que coincida con los ids, si existe llama al método **update()** para actualizarlo, a este método se le pasa un diccionario a partir del schema, para hacer esto se usa **model_dump()** con el parametro **exclude_unset=True** para no pasar los parametro nulos.

```
@router.patch(path="/{idGame}", responses={...})
def update_user_game(id_game: int, updates: UserGameUpdate, db: Session = Depends(get_db), current_user: User = Depends(get_current_user)):
    user_game = db.query(UserGame).filter(*criterion: UserGame.idUser == current_user.idUser, UserGame.idGame == id_game)
    if not user_game.first():
        raise HTTPException(status_code=404, detail="UserGame not found")
    user_game.update(updates.model_dump(exclude_unset=True))
    db.commit()
```

El último método es el de borrado, usa el mismo endpoint que en el anterior, dentro del cuerpo se busca el UserGame seleccionado y si no existe devuelve el código 404, en caso de existir lo borra de la base de datos.

```
@router.delete(path="/{idGame}", responses={...})
def delete_user_game(id_game: int, db: Session = Depends(get_db), current_user: User = Depends(get_current_user)):
    user_game = db.query(UserGame).filter(*criterion: UserGame.idUser == current_user.idUser, UserGame.idGame == id_game).first()
    if not user_game:
        raise HTTPException(status_code=404, detail="UserGame not found")
    db.delete(user_game)
    db.commit()
```

El siguiente archivo es **jwt_util.py**, aquí es donde se define todo lo necesario para que la API pueda tener autenticación, para hacerlo se usan las librerías *python-jose[cryptography]* y *python-multipart*,

Al principio del código se definen las constantes que vamos a usar, la **SECRET_KEY** deberá ser privada y esta es solo de ejemplo, para mantenerla oculta se puede guardar en un **.env** que no esté guardado en git y generar una clave nueva con el comando *openssl rand -hex 32*, **ALGORITHM** define que algoritmo usaremos para crear el token y la última variable el tiempo que durará antes de que deje de funcionar.

```
# Constantes
SECRET_KEY = "clave_secreta"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="auth/login")
```

La primera función del fichero permite crear tokens a partir de un diccionario con la información que queremos almacenar dentro y el tiempo en el que caducará, estas variables se unen y se guardan dentro del token cifrándolo con la **SECRET_KEY** y el **ALGORITHM** y lo devuelve.

```
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    to_encode = data.copy()
    expire = datetime.now(timezone.utc) + (expires_delta or timedelta(minutes=15))
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

La siguiente función verifica que un token que recibe sea valido, para ello intenta decodificarlo y comprueba que el nombre de usuario que contiene exista en la base de datos y lo obtiene para devolverle, si falla en algún punto del proceso da por hecho que el token no es válido.

```
async def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(get_db)):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise HTTPException(status_code=401, detail="Token inválido")
        user = db.query(User).filter(User.username == username).first()
        if not user:
            raise HTTPException(status_code=401, detail="Token inválido")

        return user
    except JWTError:
        raise HTTPException(status_code=401, detail="Token inválido o expirado")
```

El último archivo es **test_main.py**, es el encargado de realizar la pruebas vistas en [Pruebas unitarias](#).

4.4 Pruebas

Las pruebas realizadas están en el apartado [Pruebas unitarias](#).

4.5 Despliegue de la aplicación

La aplicación y API están pensadas para funcionar en local, pero sería sencillo hacerlas funcionar en remoto, para ello habría que modificar la ruta de los servicios de la aplicación y hacer que se pueda acceder al ordenador donde está ejecutándose la API abriendo el puerto 8000 del router y redirigiendo todas las peticiones a dicho puerto a nuestro pc.

5. Manuales

5.1 Manual de usuario

- Inicio de sesión

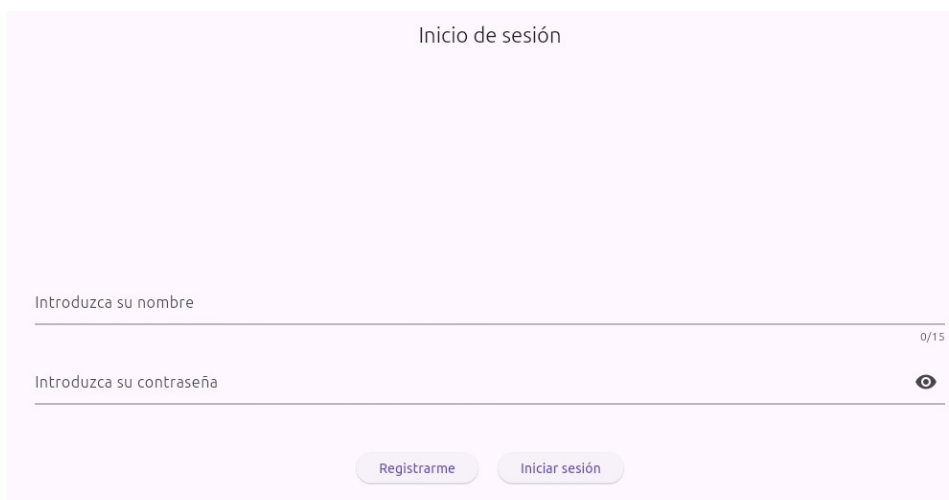
Lo que veremos al abrir la aplicación por primera vez será el inicio de sesión, la siguiente captura es la pantalla, en ella hay dos campos para introducir el nombre y la contraseña, además de un botón para registrarnos y otro para iniciar sesión.

Si nuestra primera vez usando esta aplicación deberemos introducir un nombre que no esté registrado y una contraseña con una longitud mínima de 8 caracteres.

En caso de que ya tengamos un usuario podremos iniciar sesión usando las mismas credenciales con las que nos registramos inicialmente.

Si queremos comprobar si hemos introducido correctamente la contraseña podemos pulsar el icono con el ojo para mostrar y para ocultarla volvemos a pulsarlo.

Una vez inicias sesión o te registras la aplicación guarda el usuario y la próxima vez que la abras no tendrás que iniciar sesión a no ser que la cierres manualmente.

La imagen muestra una interfaz de usuario para el inicio de sesión. El título "Inicio de sesión" está centrado en la parte superior. Hay dos campos de entrada: "Introduzca su nombre" y "Introduzca su contraseña". El campo de nombre tiene un contador "0/15" a la derecha. El campo de contraseña tiene un icono de ojo a la derecha para alternar la visibilidad. En la parte inferior, hay dos botones: "Registrarme" y "Iniciar sesión".

Inicio de sesión

Introduzca su nombre 0/15

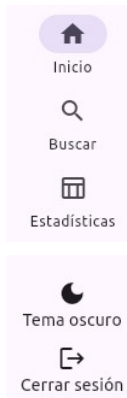
Introduzca su contraseña

Registrarme Iniciar sesión

- Menús

La aplicación tiene 2 menús distintos dependiendo de si se está visualizando en una pantalla estrecha o no.

• **Menú amplio:**

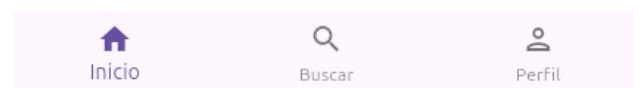


Este es el menú amplio, consta de 5 botones:

- **Inicio:** Cambia la pantalla visible a la de inicio
- **Buscar:** Cambia la pantalla visible a la de búsqueda
- **Estadísticas:** Cambia la pantalla de inicio a la de estadísticas
- **Tema oscuro/claro:** Alterna el tema entre oscuro y claro
- **Cerrar sesión:** Cierra la sesión y cambia la pantalla a la de inicio de sesión

• **Menú compacto:**

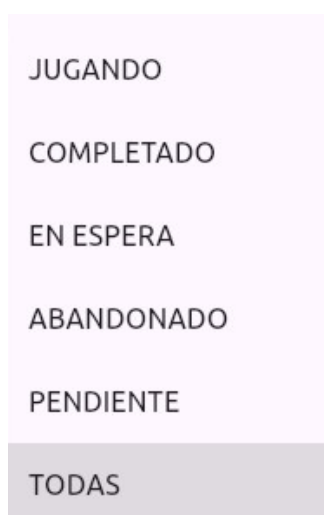
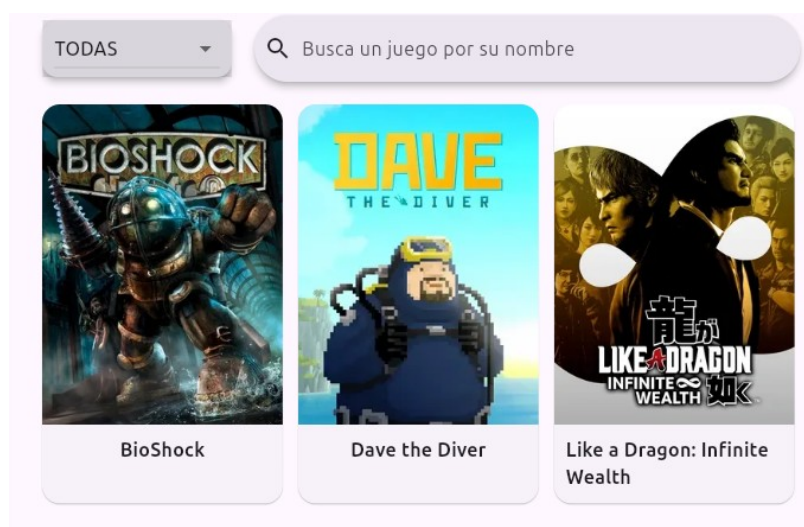
Este es el menú compacto, consta de 3 botones:



- **Inicio:** Cambia la pantalla visible a la de inicio
- **Buscar:** Cambia la pantalla visible a la de búsqueda
- **Estadísticas:** Cambia la pantalla de inicio a la de perfil, es similar a la de estadísticas pero incluye un apartado de opciones para cambiar el tema y cerrar la sesión

- Pantalla de inicio

En la pantalla de inicio el usuario puede ver los juegos que ha añadido a su colección, además puede filtrarlos por estados con el selector de arriba a la izquierda y por nombre en el buscador de la derecha.



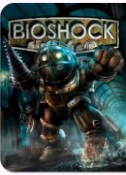
- Pantalla de búsqueda

En esta pantalla el usuario puede buscar todos los juegos de la base de datos, al igual que en la pantalla anterior también puede filtrarlos por nombre usando el buscador.

Los juegos que no están incluidos en la colección tiene un botón “+”, pulsándolo el juego se agregará a la colección del usuario.


Dependiendo del tamaño disponible en pantalla la información y cantidad de juegos puede varia para ajustarse mejor a varios tamaño

Busca un juego por su nombre




BioShock

BioShock is a horror-themed first-person shooter set in a steampunk underwater dystopia. The player is urged to turn everything into a weapon: biologically modifying their




Dave the Diver

Marine adventure set in the mysterious Blue Hole. Explore the sea with Dave by day, and run a sushi restaurant at night. Uncover the secrets of the Blue Hole, and unwrap



Disco Elysium

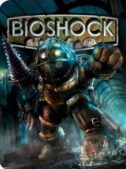
A CRPG in which, waking up in a hotel room a total amnesiac with highly opinionated voices in his head, a middle-aged detective on a m...



Helldivers 2


The Galaxy's Last Line of Offence.

Busca un juego por su nombre




BioShock

BioShock is a horror-themed first-person shooter set in a steampunk underwater dystopia. The player is urged to turn everything into a weapon: biologically modifying their own body with Plasmids, hacking devices and systems, upgrading their




Dave the Diver

Marine adventure set in the mysterious Blue Hole. Explore the sea with Dave by day, and run a sushi restaurant at night. Uncover the secrets of the Blue Hole, and unwrap this deep sea mystery involving three friends, each with distinct personalities. New



Disco Elysium


A CRPG in which, waking up in a hotel room a total amnesiac with highly opinionated voices in his head, a middle-aged detective on a murder case inadvertently ends up playing a part in the political dispute between a labour union and a larger internationa



Helldivers 2


The Galaxy's Last Line of Offence. Enlist in the Helldivers and join the fight for freedom across a hostile galaxy in a fast, frantic, and ferocious third-person shooter.

Busca un juego por su nombre




BioShock

Des: 2K Boston, 2K Australia
Ed: 2K Games



Dave the Diver

Des: MINTROCKET
Ed: MINTROCKET




Disco Elysium

Des: ZA/UM
Ed: ZA/UM

- Pantalla de estadísticas o perfil


En esta pantalla podremos ver los últimos juegos modificados, las estadísticas del usuario y en caso del perfil las opciones

Últimas actualizaciones:




Like a Dragon: Infinite Wealth

Two larger-than-life heroes, Ichiban Kasuga and Kazuma Kiryu are brought together by the hand of fate, or perhaps something more sinister... Live it up in



Dave the Diver

Marine adventure set in the mysterious Blue Hole. Explore the sea with Dave by day, and run a sushi restaurant at night. Uncover the secrets of the Blue Hole, and



BioShock

Estadísticas:

Jugando1

Completado0

En espera0

Abandonado0

Pendiente2

Total3

Tiempo total0


Nota promedio5.00

Opciones:

Cerrar sesión


Tema oscuro

Últimas actualizaciones:



Like a Dragon: Infinite Wealth

Des: Ryu Ga Gotoku Studios
Ed: Sega



Dave the Diver

Des: MINTROCKET
Ed: MINTROCKET

Estadísticas:

Jugando1

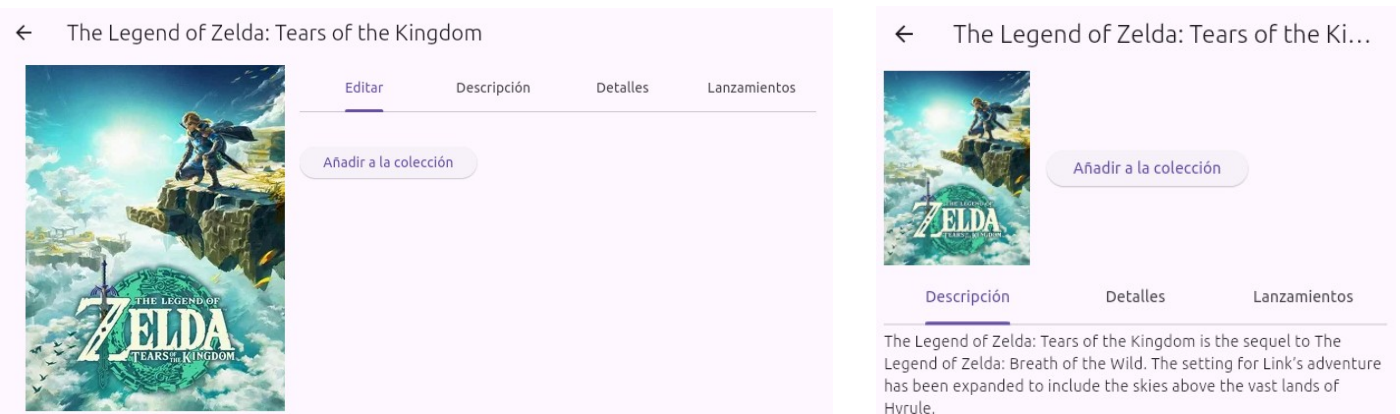
Completado0

- Pantalla de detalles

A esta pantalla se puede acceder clicando sobre un juego en las anteriores pantallas.

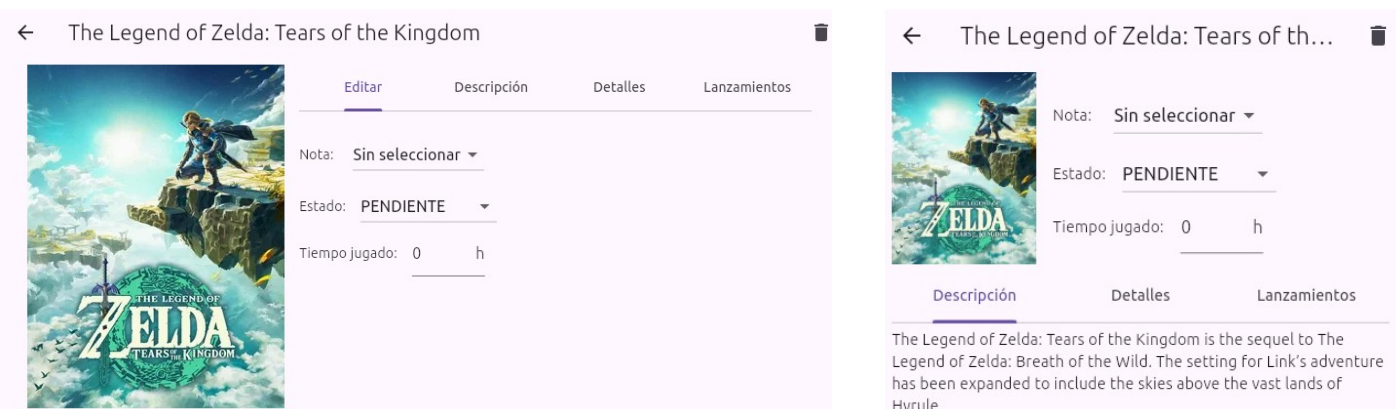
Aquí podremos ver la información del juego que hemos abierto, está organizada por pestañas, descripción, detalles y lanzamientos por las que podremos navegar. También podremos añadir, editar o eliminar un juego de nuestra colección

Esta pantalla tiene una versión para pantalla compacta y otra para pantalla anchas, lo que veremos al abrir un juego que no pertenece a nuestra colección sera el botón “Añadir a la colección”, que nos añadirá el juego con valores por defecto.



Una vez añadido, el botón cambiará por un formulario en el que podremos introducir una nota, cambiar el estado del juego y el tiempo que lo hemos jugado, al cambiar alguno de estos valores se guardará automáticamente en la base de datos.

Además aparecerá el botón para eliminar el juego arriba a la derecha, si lo pulsamos nos pedirá confirmar nuestra acción y si lo hacemos borrará el juego de la colección y volverá a aparecer el botón.



5.2 Manual de instalación

Para poder utilizar la aplicación en local hay que seguir los siguientes pasos:

1º Instalar lenguajes necesarios:

Ya que ni la aplicación ni la API están compiladas lo primero que necesitamos son los lenguajes sobre los que se ejecutan para poder lanzarlos y realizar los cambios que veamos oportunos.

Para descargar Flutter y dart seguiremos los pasos de su página web

docs.flutter.dev/get-started/install, en ella tendremos que seleccionar en que dispositivo estamos trabajando, elegir el tipo de apps que vamos a desarrollar y seguir las instrucciones que nos digan, dependiendo del sistema operativo pueden variar pero en mi caso usando linux y haciendo app de escritorio solo tengo que copiar dos comandos en la terminal.

Para instalar python podemos ir a python.org/downloads/ y bajarnos el instalador de la versión que necesitemos, ejecutarlo y usar la configuración por defecto asegurándonos de que la opción para añadir python al PATH este activa.

2º Instalar aplicaciones para lanzar los programas:

Para poder editar y ejecutar más fácilmente la app y la API recomiendo instalar los IDEs IntelliJ IDEA y PyCharm, alternativamente también se puede usar Visual Studio Code con plugins para python y flutter.

Para configurar IntelliJ IDEA descargar de jetbrains.com/es-es/idea/download la versión Community, cuando se haya instalado ejecutarlo y en la pestaña plugins buscar *Flutter* e instalarlo, esto también instalará el de dart.

Para instalar PyCharm descargar de jetbrains.com/es-es/pycharm/download la versión Community.

También necesitamos docker y para facilitar el trabajo con este instalaremos docker desktop desde docker.com/products/docker-desktop/, bajaremos hasta el apartado *Download* y elegiremos la versión que queremos. Al ejecutar el instalador se configurará automáticamente.

3º Descargar el código:

Podremos descargar el código de la API desde el repositorio

github.com/HugoReyHol/gestion_juegos_api y la aplicación desde

github.com/HugoReyHol/gestion_juegos, al descargarnos la app tenemos que asegurarnos que estamos usando la rama **api-version**, sino no se conectará con la API.

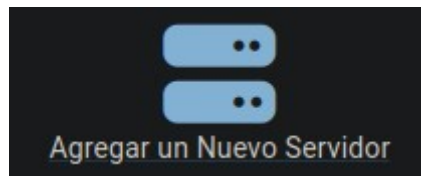
4º Crear la base de datos PostgreSQL:

Dentro del proyecto de la API hay un archivo `docker-compose.yaml` con el que podremos crear el contenedor para la base de datos, pero antes asegurándonos de cambiar las contraseñas, usuarios y emails si lo vemos necesario, en caso de hacer los cambios actualizar también la `SQLALCHEMY_DATABASE_URL` del archivo `database.py`.

Una vez localizado el archivo podemos moverlo donde queramos crear el contenedor, abrir la terminal y usar el comando `docker compose up -d`, esto creará la base de datos y su gestor web.

Lo último que tenemos que hacer para tener la base de datos es crearla dentro del contenedor, para ello encendemos los contenedores que acabamos de crear, abrimos un navegador y buscamos `localhost:80` para abrir el gestor. Para iniciar sesión tendremos que escribir el email y contraseña del apartado `pgadmin` del archivo `.yaml`, si queremos usarlo en español seleccionaremos `spanish`.

Para conectarnos a la base de datos tendremos que clicar sobre *Agregar un Nuevo Servidor*.



Al hacerlo nos abrirá un formulario, en el escribiremos el nombre que queramos darle y cambiaremos a la pestaña conexión, en esta introduciremos el nombre del contenedor del `.yaml`, en nuestro caso `db`, además del nombre y contraseña, si no la hemos cambiado son `dbuser` y `dbpass`.

Cuando hayamos realizado la conexión le daremos clic derecho iremos a *Crear* y clicaremos *Base de datos...*, esto nos pedirá rellenar un campo llamado Base de Datos, escribiremos `gestion_juegos` y lo guardaremos.

Con estos pasos la base de datos se estará creada y funcional.

5º Lanzar la API:

Para lanzar la API primero necesitaremos instalar las librerías que requiere el proyecto, abriremos el proyecto `gestion_juegos_api` con PyCharm, abajo a la izquierda clicaremos el icono *terminal* y ejecutaremos el comando `pip install -r requirements.txt`, cuando acabe de ejecutarse se habrá instalado todo lo que necesitamos para lanzar la API, para asegurarnos ejecutaremos el archivo llamado `test_main.py` dándole clic derecho y seleccionando la opción con el play verde, si no da ningún error quiere decir que hemos realizado correctamente la instalación y podremos ejecutar `main.py` para lanzar la API.

6º Abrir la aplicación:

Abriremos el proyecto `gestion_juegos` con IntelliJ IDEA, arriba desplegaremos el selector y seleccionaremos la opción (desktop), después ejecutaremos el archivo `main.dart`.

Después de seguir estos pasos tendremos nuestra aplicación funcional.

6. Conclusiones y posibles ampliaciones

Hacer la API con FastAPI ha sido bastante sencillo, donde he tenido más problemas ha sido creando las configuraciones en los schemas que las requerían ya que la aplicación que tenía no podía procesar correctamente los datos.

Hacer las llamadas y consumir los datos desde dart también ha resultado ser fácil y usar una API ha hecho que el código que tenía escrito para acceder a la base de datos ahora sea más sencillo.

Como he tenido que hacer la API en muy poco tiempo no he tenido la oportunidad de investigar tan a fondo como me hubiese gustado los distintos apartados o de refactorizar el código.

Algunas ampliaciones que realizaría a la API serían:

- Tener en cuenta la caducidad del token para cerrar la sesión del usuario cuando lleve mucho tiempo sin utilizar la aplicación.
- Usar la librería `python-dotenv` para almacenar la clave secreta del token y no seguirla con git.
- Crear un método para insertar juegos a la base de datos sin usar el archivo `db_initialize.py`.

7. Bibliografía

Información de las API: material dado en clase

Uso de JWT: fastapi.tiangolo.com/tutorial/security/oauth2-jwt/