



# PROYECTO SGE

## 2ª EVALUACIÓN

---

CFGS Desarrollo de Aplicaciones  
Multiplataforma  
Informática y Comunicaciones

---

**APLICACIÓN CRUD CON FASTAPI E  
INTEGRACIÓN EN CLIENTE MÓVIL**

*Curso: 2024/25*

**Nombre y Apellidos: Álvaro Cilleruelo Sinovas**  
**Email: alvaro.cilsin@educa.jcly.es**

# Índice

Índice.....	2
1 - Introducción.....	4
2 - Estado del arte .....	5
2.1 – Definición de arquitectura de microservicios .....	5
2.2 – Definición de API .....	5
2.3 – Estructura de una API.....	5
2.3.1 – Protocolo utilizado.....	5
2.3.2 – Métodos HTTP.....	5
2.3.3 – URL en una API .....	5
2.3.4 – Encabezados y cuerpo .....	6
2.3.5 – Códigos de estado .....	6
2.4 – API en Python.....	6
2.4.1 – Flask .....	6
2.4.2 – FastAPI.....	7
2.4.3 – Elección en el proyecto .....	7
3 - Descripción general del proyecto .....	9
3.1 – Objetivos.....	9
3.2 – Entorno de trabajo .....	9
4 - Documentación técnica.....	11
4.1 – Análisis del sistema.....	11
4.1.1 - API <i>Backend</i> .....	11
4.1.2 – Aplicación móvil .....	11
4.2 - Diseño de la base de datos .....	11
4.3 – Implementación .....	12
4.3.1 – API <i>Backend</i> .....	12
4.3.2 – Aplicación móvil .....	16
4.4 – Pruebas .....	18
4.4.1 - Prueba para la API .....	19

4.4.2 - Prueba de Integración entre la Aplicación Móvil y la API .....	20
4.5 – Despliegue de la aplicación .....	20
5 – Manuales .....	21
5.1 – Manual del usuario .....	21
5.1.1 – API <i>backend</i> .....	21
5.1.2 – Aplicación móvil .....	22
5.2 – Manual de instalación .....	23
5.2.1 – Despliegue del <i>backend</i> (API) .....	23
5.2.2 – Despliegue de la aplicación móvil .....	24
5.2.2 – Despliegue en producción .....	24
6 – Conclusiones y posibles ampliaciones .....	25
6.1 – Conclusiones .....	25
6.2 – Posibles ampliaciones .....	26
7 – Bibliografía .....	26

## 1 - Introducción

El presente proyecto consiste en el desarrollo e integración de una API RESTful utilizando FastAPI, la cual servirá de *backend* para una aplicación móvil de juego titulada "Mansión Embrujada o Juego Tablero" la cual, hasta el momento, no tenía persistencia de datos. La aplicación, desarrollada en Kotlin, permite al usuario adentrarse en una mansión llena de desafíos y acertijos que deberá resolver para avanzar por diferentes habitaciones. En este entorno de juego, se gestiona de forma dinámica tanto la navegación por las salas como la resolución de enigmas, mientras se registran diversas estadísticas de juego, como el número de acertijos resueltos, los intentos fallidos, las salas visitadas y el tiempo transcurrido.

La API se encarga de ofrecer servicios esenciales para la aplicación, tales como la autenticación de usuarios mediante JWT, la gestión del registro y consulta de estadísticas de juego, y el suministro de acertijos mediante operaciones CRUD sobre una base de datos. Se ha implementado un mecanismo de precarga para dotar a la base de datos de un conjunto inicial de acertijos, permitiendo así que la experiencia de juego sea variada y escalable. Además, la integración entre la API y la aplicación móvil se realiza a través de peticiones HTTP, garantizando una comunicación fluida y segura entre ambos componentes.

Este proyecto no solo demuestra el uso de tecnologías modernas en el desarrollo de aplicaciones (como FastAPI, SQLAlchemy y JWT en el lado del servidor, y Kotlin junto con Retrofit para el consumo de la API en el cliente), sino que también destaca buenas prácticas en el diseño de software, la gestión de errores y la implementación de pruebas unitarias. En resumen, la solución propuesta permite a los usuarios disfrutar de una experiencia de juego interactiva y dinámica, respaldada por una infraestructura robusta y escalable que integra de manera eficiente el *backend* y el *frontend* de la aplicación.

Este proyecto es de código abierto y se encuentra disponible en los siguientes repositorios de GitHub:

- API backend: [RiddleFastAPI](#)
- Aplicación móvil: [JuegoTablero](#)

## 2 - Estado del arte

### 2.1 – Definición de arquitectura de microservicios

La arquitectura de microservicios es un estilo de diseño de software que consiste en descomponer una aplicación compleja en un conjunto de servicios pequeños, independientes y desplegables de manera autónoma. Cada uno de estos servicios está diseñado para ejecutar una función específica y se comunica con los demás a través de interfaces bien definidas, usualmente mediante protocolos de red ligeros como HTTP/REST o mensajería basada en eventos.

### 2.2 – Definición de API

Una API (por sus siglas en inglés, *Aplicación Programming Interface*) es un conjunto de definiciones, protocolos y herramientas que permiten que diferentes aplicaciones o servicios se comuniquen entre sí. Funciona como una interfaz de programación que establece un contrato entre el proveedor y el consumidor de información, especificando qué funcionalidades están disponibles y cómo deben ser invocadas.

### 2.3 – Estructura de una API

#### 2.3.1 – Protocolo utilizado

La mayoría de las API modernas se comunican mediante el protocolo HTTP, la cual permite la comunicación básica, o HTTPS, que es su versión segura gracias a la aplicación de cifrado.

#### 2.3.2 – Métodos HTTP

Los métodos HTTP definen las operaciones que se pueden realizar sobre los recursos de la API.

- GET: Se utiliza para obtener o leer recursos sin modificar el estado del servidor.
- POST: Se usa para crear nuevos recursos.
- PUT: Permite actualizar o reemplazar un recurso existente de forma completa.
- PATCH: Se utiliza para actualizar parcialmente un recurso.
- DELETE: Elimina un recurso.
- Otros métodos: Métodos como HEAD y OPTIONS se utilizan para obtener metainformación sobre los recursos o para la negociación de contenido.

#### 2.3.3 – URL en una API

La URL (*Uniform Resource Locator*) que se usa en las peticiones a la API suele estar compuesta por varios elementos:

- Esquema: Indica el protocolo, por ejemplo, http:// o https://.
- Host: Es el dominio o dirección IP del servidor que aloja la API, por ejemplo, api.ejemplo.com.
- Puerto: Especifica el puerto del servidor. Por defecto, el puerto 80 se usa para HTTP y el 443 para HTTPS, aunque puede variar según la configuración.

- Ruta (*Path*): Define la ubicación del recurso en el servidor. Ejemplo: /usuarios, /productos/123.
- Parámetros de ruta: Son segmentos dinámicos que forman parte de la ruta. Ejemplo: /usuarios/{id} donde {id} es un valor variable.
- Parámetros de consulta (*Query Parameters*): Se agregan al final de la URL después del signo de interrogación (?) para filtrar o modificar la solicitud. Ejemplo: /productos?categoria=electronica&orden=asc.
- Fragmento: Es una parte opcional que sigue a un signo de numeral (#), aunque en el caso de las API RESTful rara vez se utiliza.

### 2.3.4 – Encabezados y cuerpo

En una comunicación HTTP, encontramos dos elementos. Los encabezados (*headers*), los cuales permiten enviar información adicional o metadatos, y el cuerpo (*body*), que contiene los datos requeridos para métodos como POST, PUT o PATCH. Algunos ejemplos comunes de *headers* son *Content-Type* que especifica el formato del cuerpo (ej. *application/json*) o *Authorization* que se usa para enviar tokens o credenciales (ej. *Authorization: Bearer <token>*).

### 2.3.5 – Códigos de estado

Cada respuesta de la API incluye un código de estado que indica el resultado de la solicitud los cuales se clasifican según su primer dígito (1 al 5) en informativo, éxito, redirección, error de cliente o error del servidor. Los más comunes son:

- *200 OK*: La solicitud se procesó correctamente.
- *201 Created*: Un nuevo recurso fue creado.
- *400 Bad Request*: La solicitud contiene errores de sintaxis o parámetros incorrectos.
- *401 Unauthorized*: Se requiere autenticación o las credenciales son incorrectas.
- *403 Forbidden*: El acceso está prohibido a pesar de estar autenticado.
- *404 Not Found*: El recurso solicitado no existe.
- *500 Internal Server Error*: Error en el servidor.

## 2.4 – API en Python

Tanto FastAPI como Flask son frameworks populares en Python para crear APIs, pero presentan diferencias significativas que influyen en la elección según el tipo de proyecto.

### 2.4.1 – Flask

- Madurez y comunidad: Flask es uno de los *frameworks* más veteranos y ampliamente adoptados en el ecosistema Python. Cuenta con una gran comunidad y una abundante cantidad de extensiones y recursos disponibles.

- Enfoque minimalista: Flask se centra en ser un *microframework*, ofreciendo una base muy ligera sobre la que se puede construir la aplicación. Esto permite gran flexibilidad, ya que el desarrollador decide qué componentes o extensiones integrar para cumplir con sus necesidades.
- Sin soporte nativo para asincronía: Aunque es posible trabajar con asincronía en Flask mediante extensiones o adaptaciones, no cuenta de forma nativa con soporte para programación asíncrona. Esto puede ser una limitación en aplicaciones que requieren alto rendimiento o que manejen múltiples solicitudes concurrentes de forma eficiente.
- Documentación manual: La generación de documentación automática (por ejemplo, *Swagger*) no está integrada de forma nativa y, en muchos casos, requiere el uso de extensiones adicionales o configuraciones manuales.

#### 2.4.2 – FastAPI

- Alto Rendimiento y Soporte Asíncrono: FastAPI está diseñado desde cero para aprovechar las capacidades asíncronas de Python (usando *async/await*). Esto permite que la API maneje un alto volumen de solicitudes concurrentes de manera eficiente.
- Documentación Automática: Uno de los grandes atractivos de FastAPI es que genera automáticamente la documentación interactiva (Swagger UI y Redoc) basándose en los modelos Pydantic y en las definiciones de rutas. Esto facilita la integración y el *testing*, tanto para desarrolladores como para consumidores de la API.
- Validación y tipado automáticos: Gracias a Pydantic, FastAPI ofrece validación de datos y tipado automático de entrada y salida, lo que reduce la posibilidad de errores y mejora la claridad del contrato de la API. Esto es especialmente útil en proyectos que requieren precisión en los datos y escalabilidad en el desarrollo.
- Facilidad para crear microservicios: Su diseño modular y su rendimiento lo hacen ideal para arquitecturas basadas en microservicios, donde cada componente puede desarrollarse y desplegarse de forma independiente.

#### 2.4.3 – Elección en el proyecto

Para el proyecto, se ha optado por **FastAPI** debido a varias razones clave:

- Rendimiento y concurrencia: La aplicación de juego puede requerir un manejo eficiente de múltiples solicitudes (por ejemplo, autenticación, consulta de acertijos, registro de estadísticas) en un entorno de alta concurrencia. El soporte nativo para programación asíncrona de FastAPI es una ventaja importante.
- Documentación automática: La generación automática de documentación (Swagger UI y Redoc) facilita la integración con la aplicación móvil desarrollada en Kotlin, permitiendo que los desarrolladores puedan probar y entender rápidamente los endpoints de la API.
- Validación y tipado: La validación de datos y el tipado automático que ofrece FastAPI, a través de Pydantic, aseguran que la API maneje entradas y salidas de forma consistente y robusta. Esto reduce errores y mejora la mantenibilidad del código.

- **Desarrollo ágil y escalabilidad:** La estructura modular y la facilidad para integrar diferentes componentes hacen que FastAPI sea adecuado para proyectos que pueden crecer o evolucionar con el tiempo, permitiendo añadir nuevas funcionalidades sin alterar significativamente la base del sistema.



## 3 - Descripción general del proyecto

### 3.1 – Objetivos

El objetivo principal del proyecto es desarrollar e integrar una API robusta y escalable que cumpla los siguientes propósitos:

- Autenticación y Seguridad: Implementar un sistema de registro e inicio de sesión de usuarios utilizando JWT, asegurando el acceso a rutas protegidas y permitiendo la personalización de la experiencia de juego.
- Gestión de acertijos: Proveer un conjunto de acertijos (*riddles*) mediante operaciones CRUD, de forma que la aplicación móvil pueda consultar y presentar desafíos dinámicos a los usuarios. Esto incluye la precarga de acertijos por defecto en la base de datos, ampliando la jugabilidad y permitiendo la incorporación de nuevas categorías en el futuro.
- Registro y consulta de estadísticas: Permitir el registro de las estadísticas de cada partida (acertijos resueltos, intentos fallidos, salas visitadas y tiempo empleado) y la consulta de un historial de partidas, lo que posibilita la comparación y el seguimiento del rendimiento del usuario a lo largo del tiempo.
- Integración y Comunicación: Establecer una comunicación fluida y segura entre la API y la aplicación móvil a través de peticiones HTTP/HTTPS, utilizando estándares modernos que facilitan el desarrollo, la documentación (Swagger, Redoc) y el testing de la API.

### 3.2 – Entorno de trabajo

- Lenguaje de programación: Python se emplea para el desarrollo de la API por su facilidad de uso y gran ecosistema de librerías. Kotlin se emplea para el desarrollo de la aplicación móvil por su curva de aprendizaje y su natividad en Android.
- Framework web: FastAPI ha sido la elección para el desarrollo de la API, gracias a su alto rendimiento, soporte nativo para programación asíncrona, integración con Pydantic para la validación de datos y la generación automática de documentación interactiva (Swagger UI y Redoc).
- ORM: SQLAlchemy se utiliza para la gestión de la base de datos, facilitando la interacción con los modelos y la realización de operaciones CRUD.
- Autenticación: Se ha implementado un sistema de autenticación basado en JWT para garantizar el acceso seguro a los recursos protegidos de la API.
- Base de Datos: Se utiliza un gestor de base de datos para almacenar la información de usuarios, estadísticas de juego y acertijos. En este caso se emplea SQLite, el cual es una solución óptima para la etapa de desarrollo por su ligereza y alto rendimiento. No obstante, se recomienda cambiarla en el paso a producción para mayor escalabilidad y seguridad.

- IDE y Editor de Código: Se ha utilizado PyCharm para el desarrollo de la API, aprovechando su integración con el lenguaje y las herramientas que ofrece para facilitar la programación. Para su integración en la aplicación móvil se ha empleado Android Studio por, al igual que en el caso anterior, su integración con Kotlin.
- Control de versiones: Git se ha utilizado para gestionar el código fuente, facilitando la colaboración, el seguimiento de cambios y la integración continua. GitHub se ha empleado como repositorio en la nube para hacer el proyecto accesible desde distintos entornos de trabajo.
- Testing: Se han desarrollado pruebas unitarias utilizando *pytest*, lo que garantiza la calidad y fiabilidad del código, permitiendo detectar y corregir errores de forma temprana.
- Otras Herramientas y Dependencias: Uvicorn, el cual es un servidor ASGI (*Asynchronous Server Gateway Interface*) para ejecutar la aplicación FastAPI y CORS Middleware, para gestionar el acceso a la API desde diferentes dominios, lo cual es fundamental en entornos de desarrollo y producción.

## 4 - Documentación técnica

### 4.1 – Análisis del sistema

El sistema se ha diseñado para ofrecer una experiencia de juego interactiva en la que el usuario, a través de una aplicación móvil desarrollada en Kotlin, explora una mansión llena de habitaciones y acertijos. El sistema se divide en dos grandes bloques:

#### 4.1.1 - API *Backend*

La API, desarrollada con FastAPI en Python, actúa como backend del sistema. Entre sus principales funcionalidades se encuentran la autenticación, implementando un sistema de registro e inicio de sesión basado en JWT para garantizar la seguridad en el acceso a los recursos protegidos; la gestión de acertijos, exponiendo endpoints CRUD para la creación, consulta, actualización y eliminación de estos y contando con un mecanismo de precarga de acertijos por defecto para dotar de variedad a la experiencia de juego; y la gestión de estadísticas, permitiendo el registro y consulta de las estadísticas de juego, facilitando así el seguimiento del rendimiento del usuario y la comparación entre partidas. En la API, se utilizan excepciones (por ejemplo, `HTTPException`) para capturar errores y devolver códigos de estado HTTP adecuados (como 400, 401 o 404). Se han desarrollado pruebas automatizadas para la API usando `pytest` y el `TestClient` de FastAPI, garantizando que cada *endpoint* funcione según lo esperado.

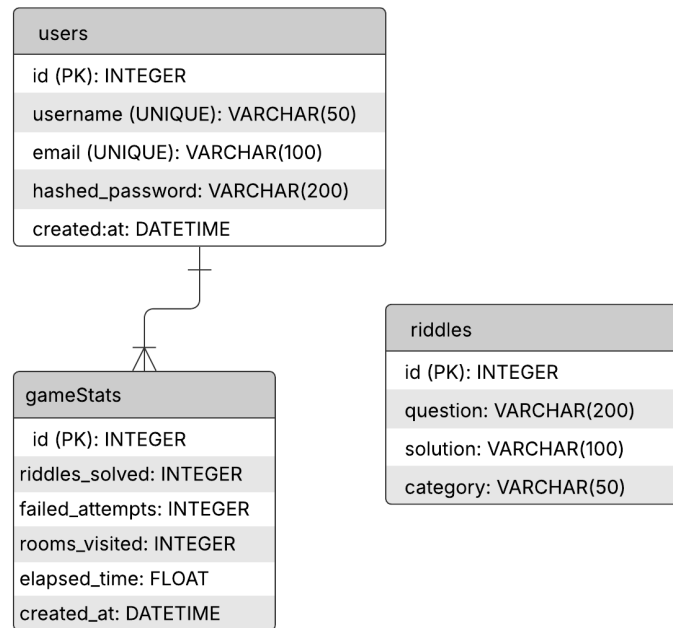
#### 4.1.2 – Aplicación móvil

La aplicación permite al usuario desplazarse por la mansión, resolver acertijos en cada sala y registrar estadísticas de juego (acertijos resueltos, intentos fallidos, salas visitadas y tiempo empleado). La lógica de la aplicación se encarga de actualizar la interfaz de usuario, validar respuestas y gestionar la navegación entre habitaciones. Además, la aplicación consume servicios proporcionados por la API para obtener acertijos dinámicos y registrar las estadísticas de cada partida. En la aplicación móvil se implementan notificaciones visuales (por ejemplo, Toasts) para informar al usuario de respuestas incorrectas o problemas en la conexión con la API. Se han implementado pruebas de integración en el lado de la aplicación móvil (utilizando `MockWebServer`) para simular la comunicación con la API.

### 4.2 - Diseño de la base de datos

La base de datos del sistema ha sido diseñada para almacenar la información de los usuarios, las estadísticas de juego y los acertijos. La estructura se ha modelado utilizando SQLAlchemy con SQLite y se ha definido en un esquema relacional con las siguientes tablas:

- **Users:** contiene la información de registro de cada usuario.
- **GameStats:** registra las estadísticas de cada partida.
- **Riddles:** almacena los acertijos con un campo de categoría para personalizar la jugabilidad.



Este diseño relacional permite gestionar de forma eficiente la información de cada componente del sistema y establecer relaciones claras entre usuarios y sus estadísticas.

## 4.3 – Implementación

La implementación del proyecto se ha realizado siguiendo un enfoque modular y escalable, tanto en el backend (API) como en la aplicación móvil.

### 4.3.1 – API Backend

#### 4.3.1.1 - Modelos (*models*)

Utilizando SQLAlchemy, se definen los modelos de datos (**User**, **GameStats**, **Riddles**) que reflejan la estructura de la base de datos.

```

from sqlalchemy import Column, Integer, ForeignKey, DateTime, Float
from sqlalchemy.orm import relationship
from datetime import datetime, timezone

from app.data.database import Base

class GameStat(Base):
    __tablename__ = "game_stats"
    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer, ForeignKey("users.id"), nullable=False)
    riddles_solved = Column(Integer, default=0)
    failed_attempts = Column(Integer, default=0)
    rooms_visited = Column(Integer, default=0)
    elapsed_time = Column(Float, default=0.0) # Tiempo en segundos
    created_at = Column(DateTime, default=datetime.now(timezone.utc))

    user = relationship(argument="User", back_populates="game_stats")
  
```

```

from sqlalchemy import Column, Integer, String, DateTime
from sqlalchemy.orm import relationship
from datetime import datetime, timezone
from app.data.database import Base

class User(Base): 12 usages  👤 cille
    __tablename__ = "users"

    id: int = Column(Integer, primary_key=True, index=True)
    username: str = Column(String(50), unique=True, index=True, nullable=False)
    email: str = Column(String(100), unique=True, index=True, nullable=False)
    hashed_password: str = Column(String(200), nullable=False)
    created_at: datetime = Column(DateTime, default=datetime.now(timezone.utc))

    game_stats = relationship( argument: "GameStat", back_populates="user")

```

#### 4.3.1.2 - Esquemas (schemas)

Con Pydantic se crean modelos de validación y serialización que definen el contrato de datos para cada endpoint.

```

from datetime import datetime
from pydantic import BaseModel, ConfigDict

class GameStatCreate(BaseModel): 3 usages  👤 cille
    riddles_solved: int
    failed_attempts: int
    rooms_visited: int
    elapsed_time: float

class GameStatRead(BaseModel): 5 usages  👤 cille *
    model_config = ConfigDict(from_attributes=True)
    id: int
    user_id: int
    riddles_solved: int
    failed_attempts: int
    rooms_visited: int
    elapsed_time: float
    created_at: datetime

```

#### 4.3.1.3 - Autenticación

Se implementa un sistema basado en JWT para la autenticación, con funciones para la verificación de contraseñas y la generación de tokens.

```

SECRET_KEY = "mi_clave_super_secreta"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/api/login")

def verify_password(plain_password: str, hashed_password: str) -> bool: 1 usage 1 cille
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password: str) -> str: 1 usage 1 cille
    return pwd_context.hash(password)

def create_access_token(data: dict, expires_delta: timedelta = None): 1 usage 1 cille *
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.now(timezone.utc) + expires_delta
    else:
        expire = datetime.now(timezone.utc) + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

def get_user_by_username(db: Session, username: str): 2 usages 1 cille
    return db.query(models.User).filter(models.User.username == username).first()

def authenticate_user(db: Session, username: str, password: str): 1 usage 1 cille
    user = get_user_by_username(db, username)
    if not user:
        return None
    if not verify_password(password, user.hashed_password):
        return None
    return user

def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(database.get_db)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Credenciales inválidas",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    user = get_user_by_username(db, username)
    if user is None:
        raise credentials_exception
    return user

```

#### 4.3.1.4 - Endpoints (routes)

Se disponen de *routers* para cada funcionalidad (usuarios, estadísticas, acertijos) que exponen operaciones CRUD y de autenticación.

```

<< /api/riddles
router = APIRouter(
    prefix="/api/riddles",
    tags=["Riddles"]
)

<< /api/riddles/
@router.post(path: "/", response_model=schemas.RiddleResponse)  # cille
def create_riddle(riddle: schemas.RiddleCreate, db: Session = Depends(database.get_db)):
    new_riddle = models.Riddle(
        question=riddle.question,
        solution=riddle.solution,
        category=riddle.category
    )
    db.add(new_riddle)
    db.commit()
    db.refresh(new_riddle)
    return new_riddle

<< /api/riddles/
@router.get(path: "/", response_model=List[schemas.RiddleResponse])  # cille
def read_riddles(
    category: Optional[str] = Query(default=None, description="Filtrar acertijos por categoría"),
    db: Session = Depends(database.get_db)
):
    query = db.query(models.Riddle)
    if category:
        query = query.filter(models.Riddle.category == category)
    riddles = query.all()
    return riddles

<< /api/riddles/{riddle_id}
@router.get(path:("/{riddle_id}", response_model=schemas.RiddleResponse)  # cille
def read_riddle(riddle_id: int, db: Session = Depends(database.get_db)):
    riddle = db.query(models.Riddle).filter(models.Riddle.id == riddle_id).first()
    if not riddle:
        raise HTTPException(status_code=404, detail="Acertijo no encontrado")
    return riddle

```

```

<< /api/riddles/{riddle_id}
@router.put(path:("/{riddle_id}", response_model=schemas.RiddleResponse)  # cille
def update_riddle(riddle_id: int, updated_riddle: schemas.RiddleCreate, db: Session = Depends(database.get_db)):
    riddle = db.query(models.Riddle).filter(models.Riddle.id == riddle_id).first()
    if not riddle:
        raise HTTPException(status_code=404, detail="Acertijo no encontrado")
    riddle.question = updated_riddle.question
    riddle.solution = updated_riddle.solution
    riddle.category = updated_riddle.category
    db.commit()
    db.refresh(riddle)
    return riddle

<< /api/riddles/{riddle_id}
@router.delete(path:("/{riddle_id}", status_code=204)  # cille
def delete_riddle(riddle_id: int, db: Session = Depends(database.get_db)):
    riddle = db.query(models.Riddle).filter(models.Riddle.id == riddle_id).first()
    if not riddle:
        raise HTTPException(status_code=404, detail="Acertijo no encontrado")
    db.delete(riddle)
    db.commit()
    return

```

#### 4.3.1.5 - Precarga de Datos y *lifespan*

Se ha configurado un manejador de *lifespan* para precargar la base de datos con acertijos por defecto al iniciar la API.

```
@asynccontextmanager 1 usage  ▲ cille *
async def lifespan(application: FastAPI):
    db = Session(bind=engine)
    try:
        default_riddles(db)
        yield
    finally:
        db.close()

# /
app = FastAPI(
    title="API de Mansión Embrujada",
    description="Backend para la aplicación de juego en Kotlin.",
    version="1.0.0",
    lifespan=lifespan
)

def default_riddles(db: Session): 2 usages  ▲ cille *
    if db.query(Riddle).first() is not None:
        return

    init_riddles = [
        {"question": "¿Qué es algo que no se puede ver, pero siempre está contigo?", "solution": "Sombra",
         "category": "General"},
        {"question": "Cuanto más grande, menos se ve. ¿Qué es?", "solution": "Oscuridad", "category": "General"},
        {"question": "Tiene dientes pero no muerde. ¿Qué es?", "solution": "Peine", "category": "General"},
        {"question": "Vuelo sin alas, lloro sin ojos. ¿Qué soy?", "solution": "Nube", "category": "General"},
        {"question": "Si me nombras, desapareceré. ¿Qué soy?", "solution": "Silencio", "category": "General"},
    ]

    for item in init_riddles:
        riddle = Riddle(
            question=item["question"],
            solution=item["solution"],
            category=item["category"]
        )
        db.add(riddle)
    db.commit()
```

### 4.3.2 – Aplicación móvil

#### 4.3.2.1 - Desarrollo en Kotlin

Contamos con la lógica de navegación entre habitaciones, validación de respuestas y muestra de estadísticas de la aplicación original. Se implementa una pantalla básica de inicio de sesión.



```

class LoginActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.login)

        val usernameEditText = findViewById<EditText>(R.id.usernameEditText)
        val passwordEditText = findViewById<EditText>(R.id.passwordEditText)
        val loginButton = findViewById<Button>(R.id.loginButton)

        loginButton.setOnClickListener {
            val username = usernameEditText.text.toString()
            val password = passwordEditText.text.toString()

            if (username.isNotBlank() && password.isNotBlank()) {
                login(username, password)
            } else {
                Toast.makeText(context, "Complete los campos", Toast.LENGTH_SHORT).show()
            }
        }
    }

    private fun login(username: String, password: String) {
        lifecycleScope.launch {
            try {
                val apiService = RetrofitClient.instance
                val response = apiService.login(LoginRequest(username, password))
                if (response.isSuccessful) {
                    response.body()?.let { loginResponse ->
                        val sharedPref = getSharedPreferences("APP_PREFS", MODE_PRIVATE)
                        with(sharedPref.edit()) {
                            putString("JWT_TOKEN", loginResponse.access_token)
                            apply()
                        }
                        startActivity(Intent(context, MainActivity::class.java))
                        finish()
                    } ?: run {
                        Toast.makeText(context, "Respuesta nula", Toast.LENGTH_SHORT).show()
                    }
                } else {
                    Toast.makeText(context, "Usuario o contraseña incorrectos", Toast.LENGTH_SHORT).show()
                }
            } catch (e: Exception) {
                Log.e("LoginActivity", "Error: ${e.message}")
                Toast.makeText(context, "Error: ${e.message}", Toast.LENGTH_SHORT).show()
            }
        }
    }
}

```

#### 4.3.2.2 - Integración con la API:

La aplicación consume la API a través de **Retrofit** para obtener acertijos y registrar estadísticas. Se utiliza JWT para la autenticación, unificando los usuarios de la API con los de la aplicación móvil.

```

private fun uploadGameStats(
    riddlesSolved: Int,
    failedAttempts: Int,
    roomsVisited: Int,
    elapsedTimeSec: Int,
    onComplete: () -> Unit
) {
    val sharedPref = getSharedPreferences(name: "APP_PREFS", MODE_PRIVATE)
    val token = sharedPref.getString(key: "JWT_TOKEN", defValue: null) ?: run {
        onComplete()
        return
    }
    lifecycleScope.launch {
        try {
            val apiService = RetrofitClient.instance
            val request = GameStatsCreate(
                riddles_solved = riddlesSolved,
                failed_attempts = failedAttempts,
                rooms_visited = roomsVisited,
                elapsed_time = elapsedTimeSec.toFloat()
            )
            val response = apiService.postGameStats(authHeader: "Bearer $token", request)
            if (response.isSuccessful) {
                Log.d(tag: "VictoryActivity", msg: "Game stats uploaded successfully")
            } else {
                Log.e(tag: "VictoryActivity", msg: "Error uploading game stats: ${response.code()}")
            }
        } catch (e: Exception) {
            Log.e(tag: "VictoryActivity", msg: "Exception uploading game stats: ${e.message}")
        } finally {
            onComplete()
        }
    }
}

private fun fetchAllGameStats(callback: (List<GameStatsResponse>) -> Unit) {
    val sharedPref = getSharedPreferences(name: "APP_PREFS", MODE_PRIVATE)
    val token = sharedPref.getString(key: "JWT_TOKEN", defValue: null) ?: run {
        callback(emptyList())
        return
    }
    lifecycleScope.launch {
        try {
            val apiService = RetrofitClient.instance
            val response = apiService.getGameStats(authHeader: "Bearer $token")
            if (response.isSuccessful) {
                val statsList = response.body() ?: emptyList()
                callback(statsList)
            } else {
                Log.e(tag: "VictoryActivity", msg: "Error fetching game stats: ${response.code()}")
                callback(emptyList())
            }
        } catch (e: Exception) {
            Log.e(tag: "VictoryActivity", msg: "Exception fetching game stats: ${e.message}")
            callback(emptyList())
        }
    }
}

```

## 4.4 – Pruebas

Se han realizado dos pruebas automatizadas que se documentan a continuación:

#### 4.4.1 - Prueba para la API

Utilizando pytest y el TestClient de FastAPI, se han desarrollado *test* unitarios para los *endpoints* de registro e inicio de sesión,

Se simula el registro de un usuario y se verifica que, al intentar duplicarlo, se produce un error 400. Asimismo, se valida que el endpoint de *login* devuelva un token JWT.

```
from fastapi.testclient import TestClient

from app.main import app
from app.data.database import get_db, engine, Base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_TEST_DATABASE_URL = "sqlite:///memory:"
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base.metadata.create_all(bind=engine)

# Sobrescribe la dependencia de la base de datos para las pruebas.
def override_get_db(): 1 usage new *
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()

app.dependency_overrides[get_db] = override_get_db

client = TestClient(app)

def test_signup_and_login(): new *
    import uuid
    # Genera un usuario y email únicos para el test
    unique_username = "testuser_" + str(uuid.uuid4())
    unique_email = f"test_{uuid.uuid4()}@example.com"

    # Registro exitoso
    response = client.post(url="/api/signup", json={
        "username": unique_username,
        "email": unique_email,
        "password": "secret123"
    })
    assert response.status_code == 200, response.text
    data = response.json()
    assert data["username"] == unique_username

    # Intentar registrar el mismo usuario nuevamente y esperar error
    response_dup = client.post(url="/api/signup", json={
        "username": unique_username,
        "email": unique_email,
        "password": "secret123"
    })
    assert response_dup.status_code == 400, response_dup.text

    # Login exitoso
    response_login = client.post(url="/api/login", json={
        "username": unique_username,
        "password": "secret123"
    })
    assert response_login.status_code == 200, response_login.text
    login_data = response_login.json()
    assert "access_token" in login_data
```

#### 4.4.2 - Prueba de Integración entre la Aplicación Móvil y la API

En el lado de Android se ha utilizado MockWebServer (de OkHttp) para simular respuestas de la API. Con Retrofit, se realizan llamadas al *endpoint* de *login* y se verifica que la aplicación procesa correctamente la respuesta (por ejemplo, extrayendo y utilizando el *token* JWT).

Estas pruebas aseguran que la integración entre el *backend* y la app móvil funcione de forma coordinada y sin errores.

```
package dam.pmdm.juegotablero

import dam.pmdm.juegotablero.models.login.LoginRequest
import dam.pmdm.juegotablero.models.login.LoginResponse
import kotlinx.coroutines.runBlocking
import okhttp3.mockwebserver.MockResponse
import okhttp3.mockwebserver.MockWebServer
import org.junit.After
import org.junit.Before
import org.junit.Test
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import dam.pmdm.juegotablero.network.ApiService
import junit.framework.TestCase.assertEquals

class ApiIntegrationTest {

    private lateinit var mockWebServer: MockWebServer
    private lateinit var apiService: ApiService

    @Before
    fun setup() {
        mockWebServer = MockWebServer()
        mockWebServer.start()

        apiService = Retrofit.Builder()
            .baseUrl(mockWebServer.url { path: "/" }) // Usa la URL del servidor simulado
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ApiService::class.java)
    }

    @After
    fun teardown() {
        mockWebServer.shutdown()
    }

    @Test
    fun testLoginIntegration() = runBlocking {
        // Define una respuesta simulada para el login
        val fakeResponse = """
            {"access_token": "fake-token-123", "token_type": "bearer"}
        """.trimIndent()
        mockWebServer.enqueue(MockResponse().setBody(fakeResponse).setResponseCode(200))

        // Realiza la llamada al endpoint login
        val response = apiService.login(LoginRequest( username: "testuser", password: "secret123"))
        assertEquals( expected: 200, response.code())
        val loginResponse: LoginResponse? = response.body()
        assertEquals( expected: "fake-token-123", loginResponse?.access_token)
    }
}
```

#### 4.5 – Despliegue de la aplicación

El despliegue de la aplicación se ha realizado en un entorno **local** para la fase de desarrollo y pruebas. La API en localhost y la aplicación en un emulador móvil de Android Studio. El siguiente paso, el cual no se ha llevado a cabo, sería utilizar Docker para “contenerizar” la API y facilitar su ejecución en distintos entornos. En una fase posterior de producción, se evaluaría el despliegue en un entorno **remoto** (por ejemplo, en un proveedor de servicios en la nube como AWS), lo que permitiría un acceso global y escalable.

## 5 – Manuales

### 5.1 – Manual del usuario

#### 5.1.1 – API *backend*

Para usar los repositorios de la API, es necesario conocer las distintas rutas. No todas se usan en la aplicación móvil. Para ello, la aplicación está documentada de forma interactiva tanto con Swagger UI ([http:// servidor:puerto/docs](http://servidor:puerto/docs)) como con Redoc (<http:// servidor:puerto/redoc>).

- Swagger UI:


#### API de Mansión Embrujada <sup>1.0.0</sup> <sup>GA3.3</sup>

/openapi.json






Backend para la aplicación de juego en Kotlin.

Authorize 

##### Users

POST	/api/signup	Create User	▼
POST	/api/login	Login	▼
GET	/api/users/me	Read Users Me	▼ 

##### GameStats

GET	/api/game_stats/	Read Game Stats	▼ 
POST	/api/game_stats/	Create Game Stat	▼ 
GET	/api/game_stats/{stat_id}	Read Game Stat	▼ 
PUT	/api/game_stats/{stat_id}	Update Game Stat	▼ 
DELETE	/api/game_stats/{stat_id}	Delete Game Stat	▼ 

##### Riddles

POST	/api/riddles/	Create Riddle	▼
GET	/api/riddles/	Read Riddles	▼
GET	/api/riddles/{riddle_id}	Read Riddle	▼
PUT	/api/riddles/{riddle_id}	Update Riddle	▼
DELETE	/api/riddles/{riddle_id}	Delete Riddle	▼

- Redoc:

Search...

- Users > API de Mansión Embrujada (1.0.0)
- GameStats > Download OpenAPI specification [Download](#)
- Riddles > Backend para la aplicación de juego en Kotlin

Users

Create User

REQUEST BODY SCHEMA: application/json

username	string (required)	<= 50 (characters)
email	string (required)	
password	string (required)	<= 10 (characters)

Responses

- 200 Successful Response
- 422 Validation Error

API Explorer

Request samples

Failed

Content type: application/json

```
{
  "username": "testing",
  "email": "testing",
  "password": "testing"
}
```

Response samples

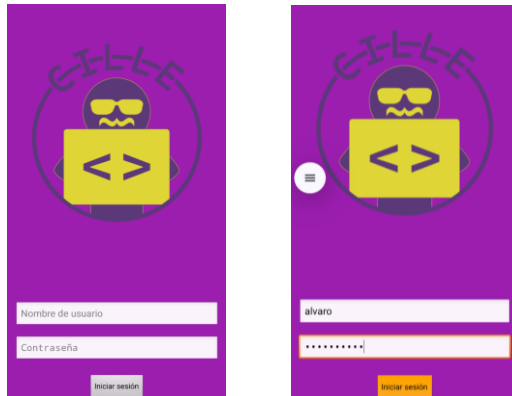
200 OK

Content type: application/json

```
{
  "id": 1,
  "username": "testing",
  "email": "testing",
  "password": "testing",
  "created_at": "2020-08-26T14:01:22Z"
}
```

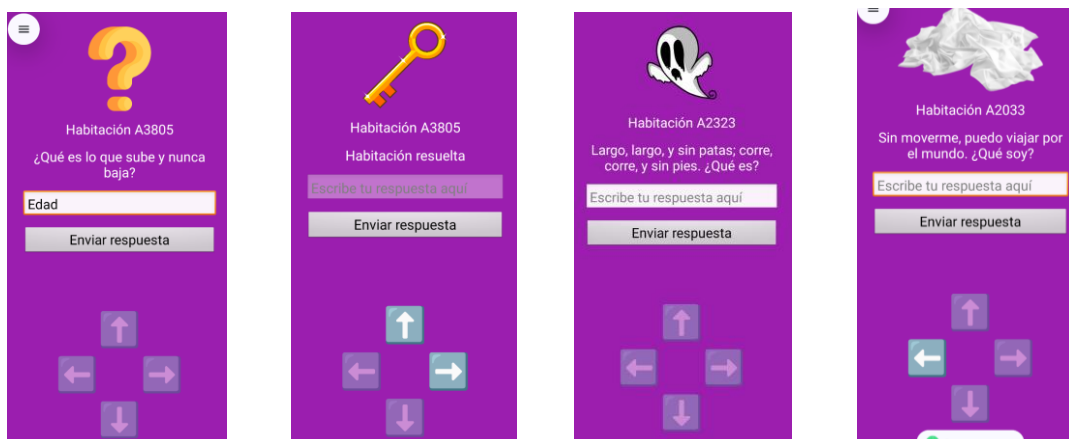
### 5.1.2 – Aplicación móvil

- Pantalla de login:



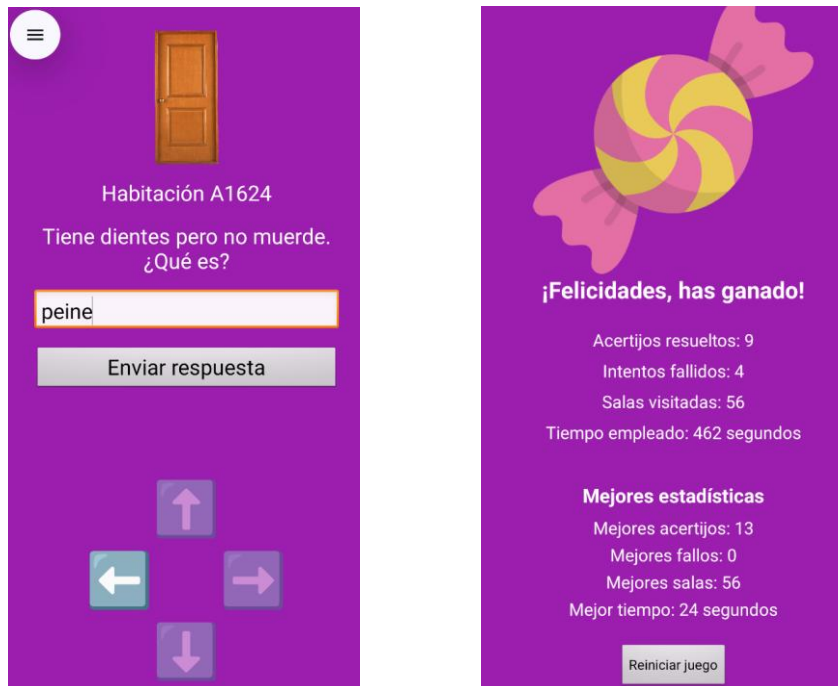
El usuario debe introducir sus credenciales (nombre de usuario y contraseña) para acceder al juego. En caso de no tener una cuenta, se proporciona la opción de registrarse, lo cual se realiza directamente a través de la API ya que no está, por el momento, programado en la aplicación móvil. Al pulsar el botón "Iniciar sesión", la aplicación se conecta con el *backend* y, si las credenciales son correctas, se genera un token de autenticación JWT que permite acceder a las funcionalidades protegidas de la API e inicia una navegación a la pantalla principal.

- Pantalla principal (juego):



La pantalla principal permite al usuario explorar la mansión. Se visualiza el nombre y descripción de la sala actual, junto con el acertijo que debe resolver. En la parte inferior se encuentran los botones de movimiento (norte, sur, este y oeste) que permiten desplazarse entre las habitaciones. Al introducir la respuesta en el campo de texto y pulsar el botón "Enviar respuesta", se mostrará si la respuesta es correcta. Permite repetir intentos y, en el momento que la respuesta es correcta, se liberan las habitaciones adyacentes a esta. Esto puede cambiar si aparece un fantasma, ya que hacen falta dos acertijos para derrotarlo. El objetivo es encontrar la puerta y adivinar su acertijo para así acceder a la pantalla de victoria.

- Pantalla de victoria:



Cuando el usuario resuelve correctamente la habitación final, se muestra la pantalla de victoria. En ella se visualizan las estadísticas de la partida (número de acertijos resueltos, intentos fallidos, salas visitadas y tiempo empleado) y se comparan de forma individual con el mejor desempeño registrado en la API. Desde esta pantalla el usuario también puede reiniciar el juego pulsando el botón "Reiniciar juego".

## 5.2 – Manual de instalación

A continuación, se presentan los pasos para un despliegue en un entorno local y se indican consideraciones para un futuro despliegue en servidor remoto.

### 5.2.1 – Despliegue del *backend* (API)

#### 5.2.1.1 - Requisitos previos

Instalar Python y contar con *pip* para gestionar las dependencias.

#### 5.2.1.2 - Configuración del entorno de desarrollo

- Clonar el repositorio del proyecto: `git clone` <https://github.com/alvarocille/RiddleFastAPI.git>

- Crear y activar un entorno virtual:

```
python -m venv venv
# En Windows:
venv\Scripts\activate
# En macOS/Linux:
source venv/bin/activate
```

- Instalar las dependencias del proyecto: `pip install -r requirements.txt`

### 5.2.1.3 – Ejecución de la API en un servidor local

En mi caso, utilicé tanto la herramienta integrada de PyCharm pero también hice algunas pruebas empleando Uvicorn ( `uvicorn app.main:app --reload` ).

### 5.2.3.4 – Acceso a la API

La API estará disponible en <http://127.0.0.1:8000> aunque hay que tener en cuenta que en emuladores la dirección a referenciar es <http://10.0.2.2:8000>. Ya se podría probar a través del navegador o un cliente API como es el caso de Postman. En <http://127.0.0.1:8000/docs> y <http://127.0.0.1:8000/redoc> se encontrará la documentación de la API con Swagger UI y Redoc, respectivamente.

## 5.2.2 – Despliegue de la aplicación móvil

### 5.2.2.1 - Requisitos previos

Tener instalado Android Studio y configurar un emulador o un dispositivo físico de desarrollo para las pruebas. Hay que tener en cuenta que en función de una u otra hay que modificar la URL de la API en el código accediendo al archivo `app.dam.pmdm.juegotablero.network.RetrofitClient`:

```
object RetrofitClient {  
    private const val BASE_URL = "http://10.0.2.2:8000/api/" // EMULADOR  
    // private const val BASE_URL = "http://127.0.0.1:8000/api/" // DISPOSITIVO FÍSICO
```

### 5.2.2.2 – Configuración y ejecución

- Clonar el repositorio del proyecto: *git clone* <https://github.com/alvarocille/JuegoTablero.git>
- Abrir el proyecto en Android Studio y sincronizar el proyecto para que se descarguen todas las dependencias definidas en el archivo “`build.gradle.kts`”.
- Ejecutar la aplicación desde Android Studio en el emulador o dispositivo.

## 5.2.2 – Despliegue en producción

- Servidor Remoto: Para un despliegue en producción, se recomienda utilizar un servidor remoto o servicios cloud (por ejemplo, AWS, Google Cloud, DigitalOcean) que permitan escalabilidad y alta disponibilidad.
- HTTPS: Es fundamental configurar HTTPS para asegurar la comunicación entre la aplicación móvil y la API.
- Configuración de Dominios y DNS: Se deberá apuntar un dominio personalizado a la dirección IP del servidor.
- Orquestación de Contenedores: En entornos más complejos, se puede utilizar Docker Compose o Kubernetes para gestionar múltiples contenedores (por ejemplo, API, base de datos, servicios de caché).



## 6 – Conclusiones y posibles ampliaciones

### 6.1 – Conclusiones

El desarrollo del proyecto ha permitido la integración exitosa de una API RESTful con una aplicación móvil, logrando un sistema distribuido que gestiona la autenticación de usuarios, el suministro dinámico de acertijos y el registro de estadísticas de juego. Entre los **aspectos más destacados** y las conclusiones se pueden mencionar:

- Integración exitosa: La aplicación móvil, desarrollada en Kotlin, se conecta de forma fluida con el backend desarrollado con FastAPI. El uso de tecnologías modernas como JWT para la autenticación y Retrofit para la comunicación HTTP ha facilitado la integración y garantizado la seguridad y escalabilidad del sistema.
- Aprendizaje y consolidación de conceptos: El proyecto ha permitido profundizar en el desarrollo de APIs en Python, el uso de frameworks modernos (FastAPI), la gestión de bases de datos con SQLAlchemy y la implementación de pruebas automatizadas. Además, se han afianzado conocimientos sobre la arquitectura de microservicios y la integración entre sistemas heterogéneos.
- Gestión de errores y validación de datos: Se implementó un manejo adecuado de errores, utilizando códigos de estado HTTP y excepciones para notificar condiciones inesperadas. La validación de datos a través de Pydantic ha contribuido a la robustez y coherencia de la API.
- Desarrollo modular y escalable: La separación del código en módulos (modelos, esquemas, rutas y lógica de negocio) ha facilitado el mantenimiento del sistema y la incorporación de futuras mejoras. Asimismo, la precarga de acertijos y el uso de endpoints para estadísticas han permitido que la aplicación se comporte de forma dinámica y adaptable a distintos escenarios.

No obstante, el proyecto también presentó **dificultades**, entre las que se destacan:

- Integración de tecnologías asíncronas: La incorporación de programación asíncrona en FastAPI requirió un ajuste en la forma de gestionar la base de datos y en el manejo de contextos (por ejemplo, en el uso de “lifespan” en lugar de “on\_event”), lo que implicó un aprendizaje adicional para adaptar el código a las nuevas prácticas recomendadas.
- Configuración y pruebas en entornos aislados: La creación de un entorno de pruebas para la API (utilizando pytest y una base de datos en memoria) y la integración con la aplicación móvil (usando MockWebServer) implicó desafíos en la configuración y el aislamiento del entorno, pero finalmente se logró establecer un sistema de tests automatizados que garantizan la calidad del código.
- Manejo de datos en la aplicación móvil: Inicialmente, la aplicación móvil generaba la instancia del juego antes de obtener los acertijos desde la API, lo que ocasionaba inconsistencias. La solución fue retrasar la inicialización de la lógica del juego hasta que se completara la consulta, lo que requirió una reestructuración parcial del código.

El grado de satisfacción en el trabajo realizado es alto, ya que se alcanzaron los objetivos planteados inicialmente y se logró integrar de forma efectiva el backend y el frontend del sistema, consolidando aprendizajes importantes en el ámbito del desarrollo de aplicaciones distribuidas y de microservicios.

## 6.2 – Posibles ampliaciones

- Implementación de **Caché** y **Rate Limiting**: Integrar una solución de caché para almacenar temporalmente respuestas de consultas frecuentes (como la lista de acertijos). Esto reduciría la carga sobre la base de datos y mejoraría los tiempos de respuesta. Asimismo, se puede incorporar un sistema de *rate limiting* para proteger la API contra ataques de denegación de servicio (DDoS).
- Campo **rol** en la tabla Users para evitar problemas de seguridad que suponen el modelo actual. En la versión inicial, alguien que se registre en la aplicación tiene acceso directo a todos los métodos de la API.
- Pantalla de **registro** en la aplicación móvil ya que el método actual es accediendo directamente a la ruta “/signup” de la API.

## 7 – Bibliografía

- Documentos de clase y explicaciones de la profesora.
- FastAPI: Tiangolo, S. FastAPI Documentation. <https://fastapi.tiangolo.com>
- SQLAlchemy: SQLAlchemy Project. SQLAlchemy Documentation. <https://docs.sqlalchemy.org>
- Pydantic: Pydantic Documentation. <https://pydantic-docs.helpmanual.io>
- JWT: JWT.io – JSON Web Tokens. <https://jwt.io>
- Pytest: Pytest Documentation. <https://docs.pytest.org/en/stable/>
- OkHttp y MockWebServer: Square, Inc. Documentation. <https://square.github.io/okhttp/>