UNIÓN EUROPEA
Fondo Social Europeo
El FSE invierte en tu futuro

PROYECTO DE DESARROLLO DE APLICACIONES MULTIPLATAFORMA

GESTOR DE RESERVAS DE CAMPOS Y PISTAS EN UN POLIDEPORTIVO

CICLO FORMATIVO DE GRADO SUPERIOR

DESARROLLO DE APLICACIONES MULTIPLATAFORMA (IFCS02)

CURSO 2024-2025

Autor/a/es:
Daniel Piekarski Kowalski

Tutor/a:
Javier Palacios González

DEPARTAMENTO DE INFORMÁTICA Y COMUNICACIONES

IES LUIS VIVES

Resumen

Este proyecto consiste en el desarrollo de un sistema de reservas para pistas deportivas en un polideportivo. Está compuesto por una API REST desarrollada en Python, una aplicación web implementada con Blazor Server y una aplicación móvil para administradores desarrollada en Android.

El sistema utiliza una base de datos PostgreSQL para almacenar la información de usuarios, pistas y reservas. La autenticación se gestiona de forma segura mediante tokens JWT. La aplicación web permite a los usuarios consultar la disponibilidad de las pistas y realizar reservas, mientras que la aplicación móvil está orientada a la gestión administrativa del polideportivo.

Tanto la API como la base de datos y la aplicación web están desplegadas en la nube a través de la plataforma Render, lo que permite el acceso desde cualquier lugar y facilita su mantenimiento y escalabilidad.

Abstract

This project focuses on the development of a reservation system for sports courts in a sports center. It is composed of a REST API developed in Python, a web application implemented with Blazor Server, and a mobile application for administrators developed in Android.

The system uses a PostgreSQL database to store information about users, courts, and reservations. Authentication is securely managed using JWT tokens. The web application allows users to check court availability and make reservations, while the mobile application is intended for the administrative management of the sports center.

The API, database, and web application are all deployed in the cloud using the Render platform, which enables access from anywhere and facilitates maintenance and scalability.

Índice

1. INTRODUCCIÓN

El proyecto consiste en el desarrollo de una aplicación web para un polideportivo que permite a los usuarios realizar reservas de campos y pistas disponibles, integrando mecanismos de autenticación y autorización. Además, se ha creado una aplicación móvil destinada a los administradores, desde la cual pueden gestionar las instalaciones de forma eficiente.

Este proyecto se ha desarrollado como parte de la formación del Ciclo Formativo de Grado Superior en Desarrollo de Aplicaciones Multiplataforma (CFGS), en la que se han aplicado los conocimientos adquiridos en los siguientes módulos:

- Desarrollo de interfaces, ya que se ha utilizado el lenguaje de programación C#.
- Aplicaciones móviles, para hacer la aplicación móvil de administradores.
- Programación de servicios y procesos, aplicando técnicas como el hash de contraseñas.
- Sistemas de gestión empresarial, ya que la API se ha creado con Python.

1.1. OBJETIVO

El objetivo del proyecto es aprender a crear aplicaciones web utilizando un framework nuevo para mí, que permita trabajar de manera sencilla con el lenguaje de programación C#, un lenguaje que ya conozco.
A su vez, quiero mejorar mi nivel en el lenguaje Kotlin para el desarrollo de aplicaciones móviles.
También aprenderé a implementar mecanismos de autenticación y autorización de usuarios mediante tokens JWT.
Además, se reforzarán otros conocimientos como la creación de APIs, programar en C#, diseñar páginas web HTML y CSS.

1.2. ALCANCE

El alcance de este proyecto incluye el desarrollo de dos aplicaciones principales:

· **Aplicación administrativa:** Los administradores podrán agregar, modificar y eliminar pistas, así como eliminar los horarios de cada campo o pista deportiva.

· **Aplicación web para usuarios:**

- Autenticación y autorización.
- Visualización de disponibilidad de campos y pistas.
- Realización y cancelación de reservas.
- Consultar reservas activas y ver tu historial de reservas.
- Agregar una foto de perfil.

· **Tecnologías empleadas:**

- Bases de datos: PostgreSQL.
- Aplicación administrativa: Desarrollada en Android Studio utilizando Kotlin.
- Aplicación web: Implementada con Blazor Server.
- API: Desarrollada con FastAPI y Python.

1.3. JUSTIFICACIÓN

Actualmente, la reserva de pistas deportivas, como las de pádel, suelen realizarse mediante llamadas, lo que además implica que solo se puede llamar en horarios en los que haya alguien disponible para atender el teléfono y adaptarse en el momento a los horarios disponibles, sin posibilidad de consultar con otros participantes antes de confirmar. Este método puede generar inconvenientes, retrasos y falta de flexibilidad para los usuarios.
La aplicación web permitirá a los usuarios realizar reservas de forma rápida y sencilla, mientras que los administradores tendrán una aplicación móvil para gestionar las instalaciones de manera sencilla y eficiente.

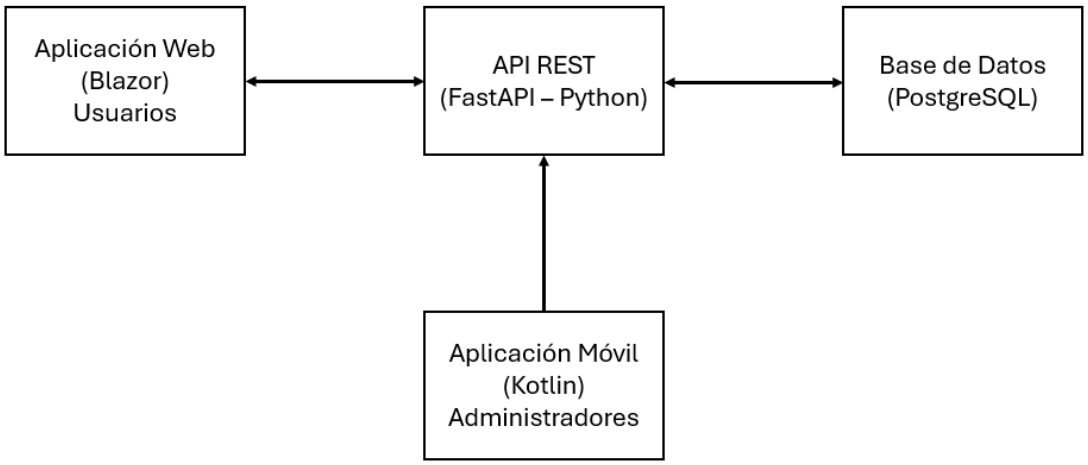
2. IMPLEMENTACIÓN

2.1. ANÁLISIS DE LA APLICACIÓN

La aplicación desarrollada tiene como objetivo gestionar la reserva de pistas deportivas (pádel, tenis y fútbol) en un polideportivo. Está compuesta por tres partes principales: una API REST desarrollada en Python con FastAPI, una aplicación web para usuarios finales creada con Blazor Server y una aplicación móvil desarrollada con Android Studio para que los administradores puedan gestionar pistas y horarios.

El sistema permite a los usuarios registrarse, iniciar sesión, consultar la disponibilidad de pistas, reservar tramos horarios, consultar sus reservas y ver su historial. Por su parte, los administradores pueden gestionar las pistas (crear, editar o eliminar), así como controlar la disponibilidad horaria de cada pista.

Para implementar esta aplicación se necesita, una API backend desarrollada con FastAPI (Python), una base de datos PostgreSQL para almacenar la información, un frontend web en Blazor Server, una aplicación móvil en Android Studio (Kotlin) para administrar las instalaciones, un entorno de despliegue en la nube (Render.com) y mecanismos de autenticación basados en JWT. Además, es necesario contar con conocimientos en desarrollo web, programación móvil y gestión de bases de datos.



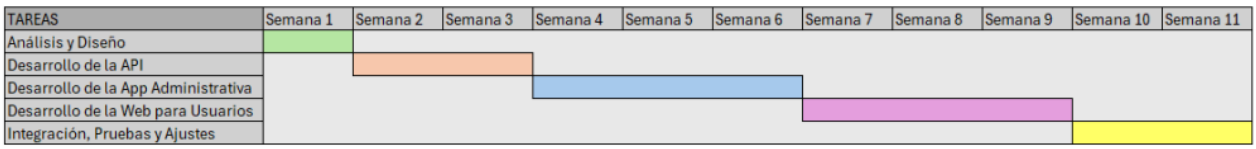
2.1.1.1. Requisitos funcionales de la aplicación

· Autenticar y autorizar usuarios para que puedan hacer reservas y que estas se guarden para cada usuario.
· Visualizar la disponibilidad horaria de cada pista y permitir hacer reservas si están disponibles.
· Consultar tus reservas activas y el historial de reservas del usuario.
· Mostrar información del usuario registrado.
· Aplicación móvil para que alguien pueda gestionar el polideportivo, en este caso solo los usuarios autorizados con el rol "admin".

2.1.2. Análisis y selección de las tecnologías

· **API REST (Backend)**:
Para desarrollar la API utilicé Python junto con el framework FastAPI, ya que permite crear un servicio robusto y escalable. Además, tenía experiencia previa creando APIs en este lenguaje.
· **Despliegue y base de datos**:
Para el despliegue de la API, la base de datos PostgreSQL y la aplicación web, elegí la plataforma Render.com. Esta plataforma facilita el despliegue en la nube mediante Dockerfile, ofrece escalabilidad automática y permite el acceso desde cualquier dispositivo con conexión a Internet. Además, facilita el mantenimiento y la actualización continua del sistema.
· **Aplicación web**:
Para la aplicación web utilicé Blazor, ya que es un framework de .NET, que en lugar de utilizar JavaScript, usa C#, lo que me resulta más cómodo ya que es un lenguaje que he utilizado durante este curso.
· **Aplicación móvil**:
La aplicación móvil la creo en Android Studio con Kotlin, ya que es una buena opción por su sencillez para crear interfaces (layouts) y programar su funcionalidad.
Además, es un lenguaje que he aprendido este curso y quería mejorar mi nivel.

2.1.3. Planificación de la realización del proyecto



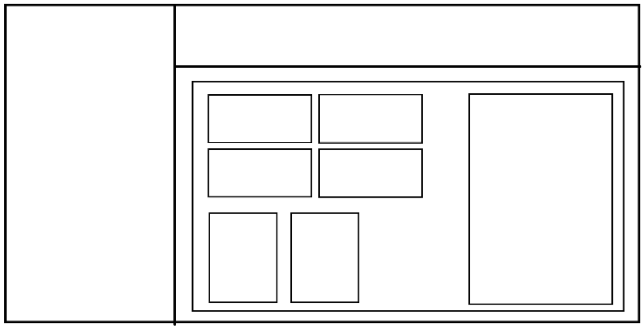
2.1.4. Otros

- Control de versiones:
Se ha utilizado Git como sistema de control de versiones y GitHub como repositorio remoto. Esto ha permitido llevar un seguimiento del desarrollo, organizar el trabajo por etapas y mantener una copia de seguridad del código.

2.2. DISEÑO

- **Prototipado:**

Primer diseño a la hora de reservar pistas, con la imagen satelital del polideportivo.



- **Diagramas E/R o modelo de datos**

A continuación, se presenta el diagrama entidad-relación (E/R) que representa la estructura lógica de la base de datos utilizada en la aplicación. Este diagrama refleja las entidades principales del sistema, como los usuarios, pistas y reservas, así como las relaciones existentes entre ellas.

Cada usuario puede realizar varias reservas, por lo que existe una relación 1:N entre "Usuario" y "Reserva".
Cada pista puede estar reservada muchas veces, por lo que también hay una relación 1:N entre "Pista" y "Reserva".
La entidad "Reserva" actúa como tabla intermedia que conecta usuarios y pistas con atributos como la fecha y hora de la reserva.

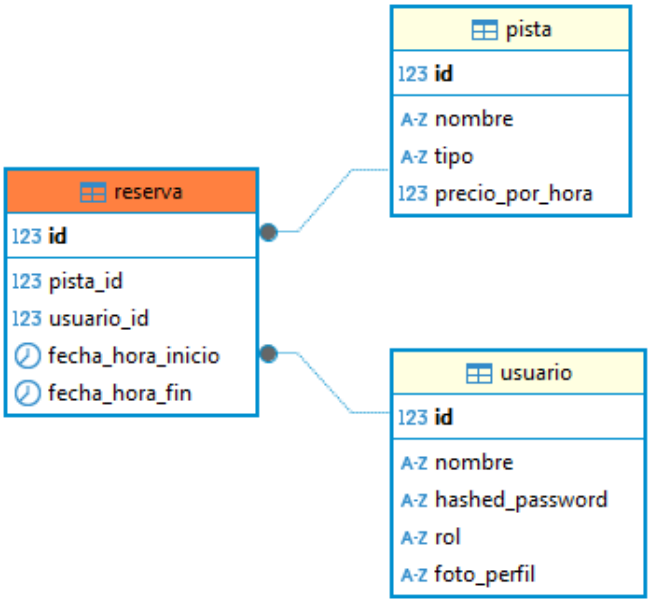


Diagrama entidad-relación generado con DBeaver.

- **Diagramas de clase genérico (Atributos y métodos principales)**

A continuación, se presentan dos diagramas de clases que reflejan los atributos más relevantes y los métodos principales de las clases utilizadas tanto en la API como en la

aplicación desarrollada con Blazor. Estos diagramas permiten comprender la estructura general del sistema y la interacción entre sus componentes.

- Diagrama 1 – Clases de la API

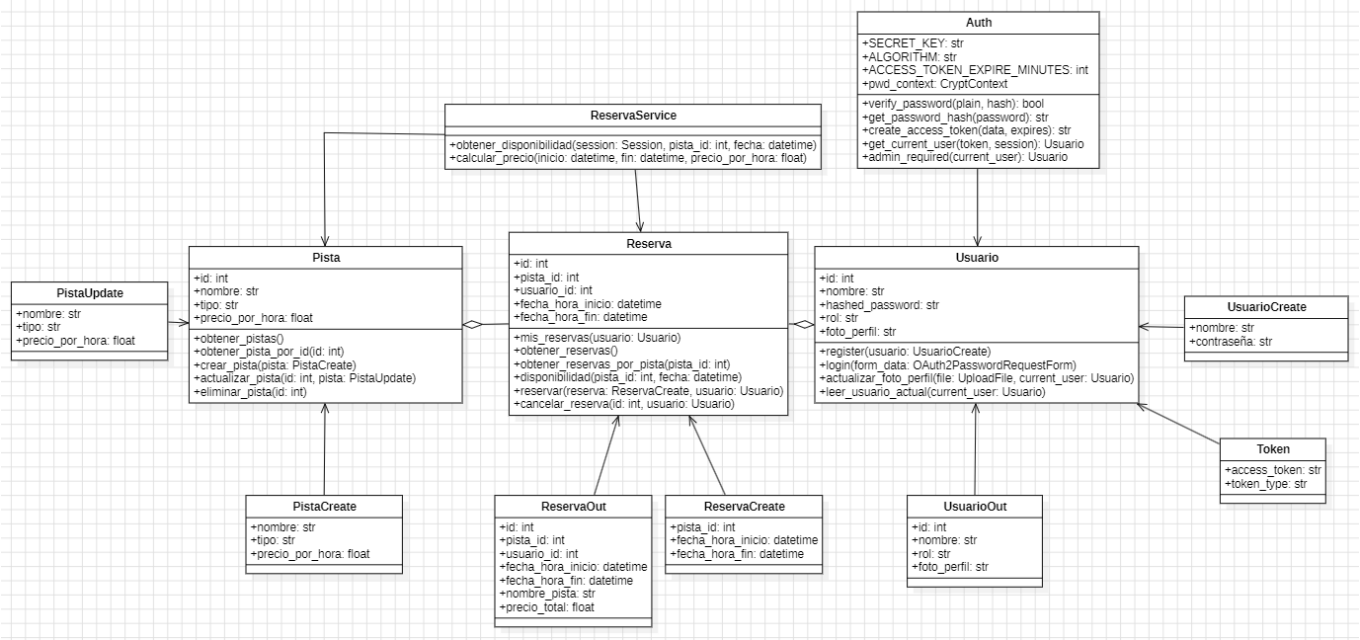


Diagrama de clases genérico hecho en StarUML.

- Diagrama 2 – Clases de la aplicación Blazor

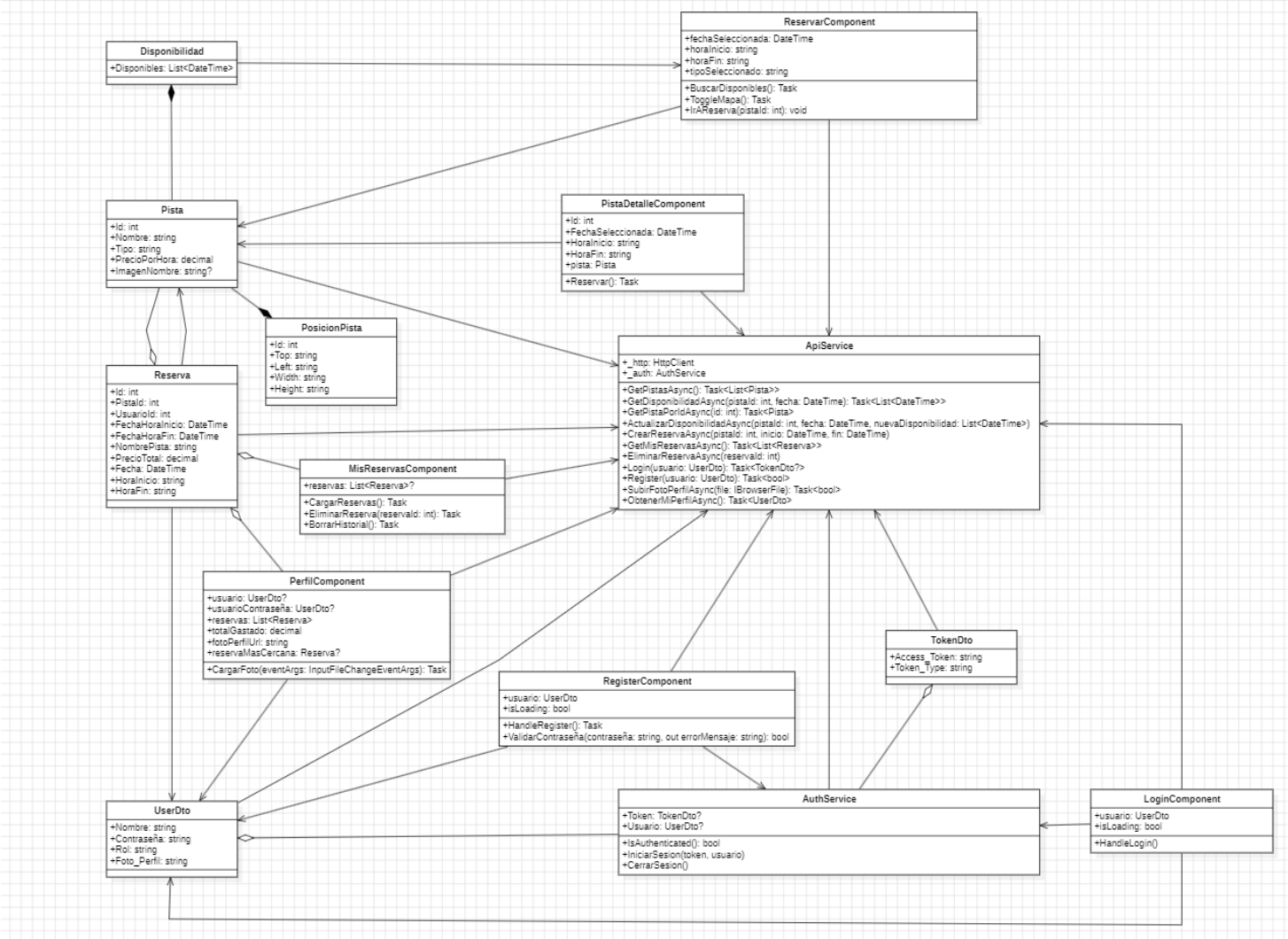


Diagrama de clases genérico hecho en StarUML.

- **Casos de uso (El desarrollo de cada caso de uso en anexos)**

Los casos de uso definen las funcionalidades principales del sistema desde la perspectiva de los usuarios, describiendo cómo interactúan con la aplicación para cumplir sus objetivos.

A continuación, se listan los casos de uso identificados en el proyecto:
- **Registrar usuario:** Permite crear una nueva cuenta para acceder al sistema.
- **Iniciar sesión:** Permite al usuario autenticarse para acceder a funciones privadas.
- **Hacer una reserva:** Permite al usuario seleccionar pista, fecha y hora para reservar una pista.
- **Consultar mis reservas:** Permite al usuario ver sus reservas realizadas y activas.
- **Cancelar reserva:** Permite al usuario cancelar una reserva antes de que comience.

Para los administradores con rol “admin”, se identifican los siguientes casos de uso adicionales:
- **Añadir pista:** Permite crear nuevas pistas en el sistema desde la aplicación móvil.
- **Editar pista:** Permite modificar los datos de una pista existente.

- Eliminar pista: Permite borrar una pista del sistema, siempre que no tenga reservas activas.
- Gestionar horarios: Permite configurar los horarios disponibles para cada pista.

El desarrollo detallado de cada caso de uso se encuentra en los anexos.

- **Base de datos. Estructura de tablas a utilizar, soporte lógico y físico.**

- Estructura de tablas

La base de datos utilizada para esta aplicación está modelada en PostgreSQL y gestionada mediante SQLModel. La base de datos almacena la información de los usuarios, pistas y reservas. A continuación, se describen las principales tablas:

· Usuario

Contiene los datos de los usuarios registrados en la aplicación.
- Campos principales: id, nombre, hashed_password, rol, foto_perfil.
- Clave primaria: id.
- Relación: 1:N con Reserva.

· Pista

Representa las pistas o campos deportivos disponibles para reservar.
- Campos principales: id, nombre, tipo, precio_hora.
- Clave primaria: id.
- Relación: 1:N con Reserva.

· Reserva

Almacena las reservas realizadas por los usuarios.
- Campos principales: id, pista_id, usuario_id, fecha_hora_inicio, fecha_hora_fin.
- Claves foráneas:
    - pista_id referencia a Pista
    - usuario_id referencia a Usuario.
- Actúa como tabla intermedia entre Usuario y Pista.

- Relaciones lógicas:

· Un usuario puede tener múltiples reservas, pero cada reserva pertenece a un único usuario.
· Una pista puede ser reservada muchas veces y cada reserva se asocia a una sola pista.
· El sistema valida para evitar solapamientos de reservas, asegurando que no se puedan reservar pistas en horarios ya ocupados.

- Soporte físico:

La base de datos se implementa físicamente utilizando PostgreSQL. La conexión con la base de datos se establece mediante SQLAlchemy, utilizando una URL de conexión definida en las variables de entorno (fichero .env). SQLModel crea y sincroniza el esquema con la línea de código:

SQLModel.metadata.create_all(engine)

Este proceso se ejecuta al inicio para asegurar la creación de las tablas.
- Motor de base de datos: PostgreSQL
- ORM utilizado: SQLModel
- Conexión: Definida mediante la variable de entorno DATABASE_URL en el archivo .env.

2.3. IMPLEMENTACIÓN

· API REST (Backend):

· Modelos y Schemas:

Las estructuras de datos utilizadas en la aplicación, incluyendo los modelos y schemas para usuarios, pistas y reservas, se presentan en el Anexo X en formato de imágenes.

· Seguridad y Autenticación con JWT:

- verify_password() y **get_password_hash():**
Estas funciones utilizan el algoritmo bcrypt para manejar de forma segura las contraseñas. **verify_password()** compara una contraseña en texto plano con su versión hasheada. **get_password_hash()** genera un hash seguro a partir de una contraseña. Este mecanismo garantiza que las contraseñas nunca se almacenen ni transmitan en texto claro.

```python
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)
```

- create_access_token(): Se encarga de crear un token JWT al iniciar sesión. El token incluye información del usuario y una fecha de expiración. Se firma con una clave secreta y un algoritmo especificado, lo que permite validar su autenticidad en futuras peticiones.
- get_current_user(): Valida el token incluido en las solicitudes autenticadas. Si el token es válido, se decodifica para extraer el nombre de usuario y se consulta la base de datos para recuperar los datos del usuario. Si el token es inválido o ha expirado, lanza una excepción HTTP 401 que bloquea el acceso.

```python
def create_access_token(data: dict, expires_delta: timedelta = None):
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

def get_current_user(token: str = Depends(oauth2_scheme), session: Session = Depends(get_session)) -> Usuario:
    credenciales_invalidas = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="No se pudieron validar las credenciales.",
        headers={"WWW-Authenticate": "Bearer"},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")
        if username is None:
            raise credenciales_invalidas
    except JWTError:
        raise credenciales_invalidas
    user = session.exec(select(Usuario).where(Usuario.nombre == username)).first()
    if user is None:
        raise credenciales_invalidas
    return user
```

- admin_required(): Función que restringe el acceso a rutas solo para usuarios con rol de administrador. Utiliza **get_current_user()** para obtener el usuario autenticado y verifica si su rol es “admin”. Si no, lanza una excepción HTTP 403.

```python
def admin_required(current_user: Usuario = Depends(get_current_user)):
    if current_user.rol != "admin":
        raise HTTPException(status_code=403, detail="Se requieren privilegios de administrador.")
    return current_user
```

· Endpoints de Autenticación y Gestión de Usuario:

- Registro de usuario (POST **/auth/register**):

Permite crear una nueva cuenta de usuario.

- Recibe un objeto con datos de usuario (UsuarioCreate), verifica si el nombre ya está registrado, y en caso contrario, guarda el usuario con la contraseña hasheada (utilizando bcrypt).
- Por defecto, asigna el rol "usuario".
- Retorna los datos del usuario creado (UsuarioOut), sin incluir la contraseña.

```python
@router.post("/register", response_model=UsuarioOut)
def register(usuario: UsuarioCreate, session: Session = Depends(get_session)):
    usuario_exist = session.exec(select(Usuario).where(Usuario.nombre == usuario.nombre)).first()
    if usuario_exist:
        raise HTTPException(status_code=400, detail="Nombre ya registrado")
    hashed_password = auth.get_password_hash(usuario.contraseña)
    new_usuario = Usuario(nombre=usuario.nombre, hashed_password=hashed_password, rol="usuario")
    session.add(new_usuario)
    session.commit()
    session.refresh(new_usuario)
    return new_usuario
```

- Login de usuario **(**POST **/auth/login**):

Autentica a un usuario y genera un token JWT para el acceso a rutas protegidas.

- Recibe credenciales a través de OAuth2PasswordRequestForm.
- Valida el usuario y contraseña con las funciones de autenticación.
- En caso de éxito, crea y retorna un token JWT que incluye el nombre de usuario y el rol.
- El token se usa para futuras solicitudes autenticadas.

```python
@router.post("/login", response_model=Token)
def login(form_data: OAuth2PasswordRequestForm = Depends(), session: Session = Depends(get_session)):
    usuario = session.exec(select(Usuario).where(Usuario.nombre == form_data.username)).first()
    if not usuario or not auth.verify_password(form_data.password, usuario.hashed_password):
        raise HTTPException(status_code=400, detail="Credenciales inválidas")
    access_token = auth.create_access_token(data={"sub": usuario.nombre, "rol": usuario.rol})
    return {"access_token": access_token, "token_type": "bearer"}
```

- Actualizar foto de perfil (PUT **/auth/usuario/foto**):

Permite al usuario autenticado subir o actualizar su foto de perfil.

- El archivo debe ser tipo JPEG o PNG; se rechazan otros formatos.
- La foto se guarda en una carpeta específica (uploads/fotos_perfil), renombrada con un identificador único para evitar conflictos.
- La ruta de la foto se almacena en el campo foto_perfil del usuario en la base de datos.
- Retorna un mensaje de éxito junto con la URL relativa de la foto.

```python
@router.put("/usuario/foto")
async def actualizar_foto_perfil(
    file: UploadFile = File(...),
    current_user: Usuario = Depends(auth.get_current_user),
    session: Session = Depends(get_session),
):
    if file.content_type not in ["image/jpeg", "image/png"]:
        raise HTTPException(status_code=400, detail="Formato no soportado. Usa JPEG o PNG")

    extension = os.path.splitext(file.filename)[1]
    nombre_archivo = f"user_{current_user.id}_{int(datetime.utcnow().timestamp())}{extension}"
    ruta_guardar = os.path.join(UPLOAD_DIR, nombre_archivo)

    with open(ruta_guardar, "wb") as buffer:
        shutil.copyfileobj(file.file, buffer)

    current_user.foto_perfil = f"/uploads/fotos_perfil/{nombre_archivo}"
    session.add(current_user)
    session.commit()
    session.refresh(current_user)

    return {"mensaje": "Foto actualizada correctamente", "foto_perfil": current_user.foto_perfil}
```

- Obtener datos del usuario actual (GET **/auth/usuario/me**):
Devuelve los datos del usuario que está autenticado en la sesión.
    - Utiliza el token JWT para obtener el usuario actual.
    - Retorna la información del usuario sin la contraseña.

```python
@router.get("/usuario/me", response_model=UsuarioOut)
def leer_usuario_actual(current_user: Usuario = Depends(auth.get_current_user)):
    return current_user
```

· Endpoints de Gestión de Pistas:

- Obtener todas las pistas (GET **/api/pistas/**):
Devuelve la lista completa de pistas registradas en la base de datos.
    - No requiere autenticación.
    - Retorna un arreglo con todos los objetos Pista.

```python
@router.get("/")
def obtener_pistas(session: Session = Depends(get_session)):
    return session.exec(select(Pista)).all()
```

- Obtener pista por ID (GET **/api/pistas/{id}**):
Busca y devuelve una pista específica según su identificador único.
    - Si la pista no existe, retorna un error HTTP 404 con mensaje "Pista no encontrada".
    - No requiere autenticación.

```python
@router.get("/{id}")
def obtener_pista_por_id(id: int, session: Session = Depends(get_session)):
    pista = session.get(Pista, id)
    if not pista:
        raise HTTPException(status_code=404, detail="Pista no encontrada")
    return pista
```

- Crear nueva pista (POST **/api/pistas/**):
Permite a un administrador registrar una nueva pista.
    - Requiere que el usuario tenga rol de administrador (verificado con el método **admin_required()**).
    - Recibe datos de la pista a crear (PistaCreate) y los almacena en la base de datos.
    - Retorna el objeto de la pista creada con su información completa.

```python
@router.post("/", dependencies=[Depends(admin_required)])
def crear_pista(pista: PistaCreate, session: Session = Depends(get_session)):
    nueva_pista = Pista(**pista.dict())
    session.add(nueva_pista)
    session.commit()
    session.refresh(nueva_pista)
    return nueva_pista
```

- Eliminar pista (DELETE **/api/pistas/{id}**):

Permite a un administrador borrar una pista existente.

- Requiere autenticación y rol de administrador.
- Si la pista no existe, retorna error 404.
- Retorna un mensaje de confirmación tras la eliminación.

```python
@router.delete("/{id}", dependencies=[Depends(admin_required)])
def eliminar_pista(id: int, session: Session = Depends(get_session)):
    pista = session.get(Pista, id)
    if not pista:
        raise HTTPException(status_code=404, detail="Pista no encontrada")

    session.delete(pista)
    session.commit()
    return {"mensaje": "Pista eliminada correctamente"}
```

- Actualizar pista (PUT **/api/pistas/{id}**):

Permite modificar los datos de una pista existente.

- Solo accesible para administradores.
- Recibe un objeto con los campos a actualizar (PistaUpdate).
- Actualiza únicamente los campos enviados (no envía valores por defecto para campos no modificados).
- Si la pista no existe, retorna error 404.
- Retorna la pista actualizada con la información modificada.

```python
@router.put("/{id}", dependencies=[Depends(admin_required)])
def actualizar_pista(id: int, pista_update: PistaUpdate, session: Session = Depends(get_session)):
    pista = session.get(Pista, id)
    if not pista:
        raise HTTPException(status_code=404, detail="Pista no encontrada")

    pista_data = pista_update.dict(exclude_unset=True)
    for key, value in pista_data.items():
        setattr(pista, key, value)

    session.add(pista)
    session.commit()
    session.refresh(pista)
    return pista
```

· Endpoints de Gestión de Reservas:

- Obtener las reservas del usuario actual (GET **/api/reservas/mis-reservas**):

Devuelve todas las reservas realizadas por el usuario autenticado.

- Cada reserva incluye también el nombre de la pista y el precio total calculado dinámicamente.
- El cálculo se hace con la función `calcular_precio(fecha_inicio, fecha_fin, precio_por_hora)`.
- Solo accesible para usuarios autenticados.

```python
@router.get("/mis-reservas", response_model=list[ReservaOut])
def mis_reservas(
    usuario: Usuario = Depends(get_current_user),
    session: Session = Depends(get_session)
):
    reservas = session.exec(
        select(Reserva).where(Reserva.usuario_id == usuario.id)
    ).all()

    reservas_out = []
    for reserva in reservas:
        pista = session.exec(select(Pista).where(Pista.id == reserva.pista_id)).first()

        precio_total = calcular_precio(reserva.fecha_hora_inicio, reserva.fecha_hora_fin, pista.precio_por_hora)

        reservas_out.append({
            "id": reserva.id,
            "pista_id": reserva.pista_id,
            "usuario_id": reserva.usuario_id,
            "fecha_hora_inicio": reserva.fecha_hora_inicio,
            "fecha_hora_fin": reserva.fecha_hora_fin,
            "nombre_pista": pista.nombre,
            "precio_total": precio_total
        })

    return reservas_out
```

- Obtener todas las reservas (GET **/api/reservas/**):
Devuelve una lista con todas las reservas almacenadas en la base de datos.
    - Útil para administración o vistas internas.
    - No requiere autenticación específica (puede ajustarse según necesidades).

```python
@router.get("/")
def obtener_reservas(session: Session = Depends(get_session)):
    return session.exec(select(Reserva)).all()
```

- Obtener reservas por pista (GET **/api/reservas/{pista_id}**):
Devuelve todas las reservas asociadas a una pista concreta según su ID.
    - Si no existen reservas para esa pista, retorna error 404.
    - Puede utilizarse para mostrar disponibilidad por pista.

```python
@router.get("/{pista_id}")
def obtener_reservas_por_pista(pista_id: int, session: Session = Depends(get_session)):
    reservas = session.exec(select(Reserva).where(Reserva.pista_id == pista_id)).all()
    if not reservas:
        raise HTTPException(status_code=404, detail="No hay reservas para esta pista.")
    return reservas
```

- Comprobar disponibilidad (GET **/api/reservas/{pista_id}/disponibilidad?fecha=...**):
Permite verificar qué horas están disponibles para una pista en una fecha concreta.
    - Usa la función **obtener_disponibilidad** del servicio de reservas.
    - Devuelve un listado con las horas aún disponibles ese día.

```python
@router.get("/{pista_id}/disponibilidad")
def disponibilidad(pista_id: int, fecha: datetime, session: Session = Depends(get_session)):
    return {"disponibles": obtener_disponibilidad(session, pista_id, fecha)}
```

- Crear una nueva reserva (POST **/api/reservas/**):
Permite a un usuario autenticado reservar una pista en un horario concreto.
    - Verifica que no haya solapamiento de horarios con reservas existentes.
    - Si hay una superposición, retorna error 400 ("La pista ya está reservada en ese horario").
    - Guarda la nueva reserva con el ID del usuario autenticado como propietario.

```python
@router.post("/")
def reservar(
    reserva: ReservaCreate,
    session: Session = Depends(get_session),
    usuario: Usuario = Depends(get_current_user)
):
    existe_reserva = session.exec(
        select(Reserva).where(
            Reserva.pista_id == reserva.pista_id,
            Reserva.fecha_hora_inicio < reserva.fecha_hora_fin,
            Reserva.fecha_hora_fin > reserva.fecha_hora_inicio
        )
    ).first()

    if existe_reserva:
        raise HTTPException(status_code=400, detail="La pista ya está reservada en ese horario")

    nueva_reserva = Reserva(
        pista_id=reserva.pista_id,
        fecha_hora_inicio=reserva.fecha_hora_inicio,
        fecha_hora_fin=reserva.fecha_hora_fin,
        usuario_id=usuario.id
    )

    session.add(nueva_reserva)
    session.commit()
    session.refresh(nueva_reserva)
    return nueva_reserva
```

- Cancelar una reserva (DELETE **/api/reservas/{id}**):
Elimina una reserva concreta si pertenece al usuario autenticado.
- Si la reserva no existe, retorna error 404.
- Si el usuario intenta cancelar una reserva que no es suya, retorna error 403.
- Retorna un mensaje de confirmación: {"detail": "Reserva cancelada"}.

```python
@router.delete("/{id}")
def cancelar_reserva(
    id: int,
    session: Session = Depends(get_session),
    usuario: Usuario = Depends(get_current_user)
):
    reserva = session.get(Reserva, id)

    if not reserva:
        raise HTTPException(status_code=404, detail="Reserva no encontrada")

    if reserva.usuario_id != usuario.id:
        raise HTTPException(status_code=403, detail="No autorizado para eliminar esta reserva")

    session.delete(reserva)
    session.commit()
    return {"detail": "Reserva cancelada"}
```

· Funciones auxiliares de reservas:

- **obtener_disponibilidad(session, pista_id, fecha)**
Calcula los horarios disponibles para una pista en una fecha concreta.
    - Omite las horas ya ocupadas por otras reservas ese día.
    - Retorna una lista con las horas aún libres entre las 08:00 y las 22:00.

- **calcular_precio(inicio, fin, precio_por_hora)**
Devuelve el coste total de la reserva.
    - Multiplica las horas reservadas por el precio por hora de la pista.
    - El resultado se redondea a 2 decimales.

```python
def obtener_disponibilidad(session: Session, pista_id: int, fecha: datetime):
    reservas = session.exec(select(Reserva).where(Reserva.pista_id == pista_id)).all()
    horarios_disponibles = []
    hora_apertura, hora_cierre = 8, 22

    for hora in range(hora_apertura, hora_cierre):
        fecha_hora = fecha.replace(hour=hora, minute=0, second=0, microsecond=0)
        if not any(r.fecha_hora_inicio < fecha_hora + timedelta(hours=1) and r.fecha_hora_fin > fecha_hora for r in reservas):
            horarios_disponibles.append(fecha_hora)

    return horarios_disponibles

def calcular_precio(inicio: datetime, fin: datetime, precio_por_hora: float) -> float:
    duracion_horas = (fin - inicio).total_seconds() / 3600
    return round(precio_por_hora * duracion_horas, 2)
```

· Aplicación Web (Frontend):

· Gestión de servicios de API (ApiService):

- Proporciona los métodos para consumir los distintos endpoints de la API REST del polideportivo.
- Maneja la comunicación HTTP con el backend, incluyendo solicitudes GET, POST, PUT y DELETE.
- Integra la autorización mediante token JWT para las operaciones que requieren autenticación.

- Obtención de pistas (**GetPistasAsync()**):
Recupera la lista completa de pistas disponibles desde la API REST.
    - Consume el endpoint ***GET /api/pistas/***.
    - Retorna una lista con los objetos pista.

```
public async Task<List<Pista>> GetPistasAsync()
{
    return await _http.GetFromJsonAsync<List<Pista>>("https://api-polideportivo.onrender.com/api/pistas/");
}
```

- Consulta de pista por ID (**GetPistaPorIdAsync()**):
Obtiene los detalles de una pista específica según su identificador.
    - Consume el endpoint GET /api/pistas/{id}.
    - Retorna la pista o null si no existe.

```
public async Task<Pista?> GetPistaPorIdAsync(int id)
{
    var url = $"https://api-polideportivo.onrender.com/api/pistas/{id}";
    return await _http.GetFromJsonAsync<Pista>(url);
}
```

- Consulta de disponibilidad (**GetDisponibilidadAsync()**):
Solicita los horarios libres de una pista para una fecha concreta.
    - Consume el endpoint, ***GET /api/reservas/{pista_id}/disponibilidad?fecha=....***
    - Devuelve una lista de horas disponibles para reservar.

```
public async Task<List<DateTime>> GetDisponibilidadAsync(int pistaId, DateTime fecha)
{
    var url = $"https://api-polideportivo.onrender.com/api/reservas/{pistaId}/disponibilidad?fecha={fecha:yyyy-MM-dd}";
    var response = await _http.GetFromJsonAsync<Disponibilidad>(url);
    return response.Disponibles;
}
```

- Creación de reserva (**CrearReservaAsync()**):
Permite a un usuario autenticado reservar una pista en un horario determinado.
    - Envía datos a ***POST /api/reservas/*** con la pista y horarios.

- Verifica el token JWT para autorización.
- Gestiona errores si el horario está ocupado.

```
public async Task CrearReservaAsync(int pistaId, DateTime inicio, DateTime fin)
{
    if (string.IsNullOrEmpty(_auth.Token?.Access_Token))
        throw new InvalidOperationException("No autenticado.");

    var reserva = new
    {
        pista_id = pistaId,
        fecha_hora_inicio = inicio.ToString("o"),
        fecha_hora_fin = fin.ToString("o")
    };

    using var request = new HttpRequestMessage(HttpMethod.Post, "https://api-polideportivo.onrender.com/api/reservas/")
    {
        Content = JsonContent.Create(reserva)
    };
    request.Headers.Authorization =
        new AuthenticationHeaderValue("Bearer", _auth.Token.Access_Token);

    var response = await _http.SendAsync(request);
    response.EnsureSuccessStatusCode();
}
```

- Obtención de reservas propias (**GetMisReservasAsync()**):
Recupera las reservas realizadas por el usuario autenticado.
- Consume ***GET /api/reservas/mis-reservas***.
- Requiere token JWT válido.
- Retorna una lista con las reservas del usuario.

```
public async Task<List<Reserva>> GetMisReservasAsync()
{
    var request = new HttpRequestMessage(HttpMethod.Get, "https://api-polideportivo.onrender.com/api/reservas/mis-reservas");
    if (!string.IsNullOrEmpty(_auth.Token?.Access_Token))
    {
        request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", _auth.Token.Access_Token);
    }
    else
    {
        throw new InvalidOperationException("No hay token de autenticación disponible.");
    }

    var response = await _http.SendAsync(request);
    if (!response.IsSuccessStatusCode)
        return new List<Reserva>();

    var content = await response.Content.ReadAsStringAsync();
    return JsonSerializer.Deserialize<List<Reserva>>(content, new JsonSerializerOptions { PropertyNameCaseInsensitive = true }) ?? new List<Reserva>();
}
```

- Eliminación de reserva (**EliminarReservaAsync()**):
Permite al usuario autenticado cancelar una reserva propia.
- Envía una petición ***DELETE /api/reservas/{id}***.
- Valida que la reserva pertenezca al usuario.
- Retorna confirmación o error.

```
public async Task EliminarReservaAsync(int reservaId)
{
    if (string.IsNullOrEmpty(_auth.Token?.Access_Token))
        throw new InvalidOperationException("No autenticado.");

    using var request = new HttpRequestMessage(HttpMethod.Delete, $"https://api-polideportivo.onrender.com/api/reservas/{reservaId}");
    request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", _auth.Token.Access_Token);

    var response = await _http.SendAsync(request);
    response.EnsureSuccessStatusCode();
}
```

- Autenticación (**Login()**):
Envía las credenciales del usuario para obtener un token JWT.
    - Consume ***POST /auth/login*** con nombre de usuario y contraseña.
    - Almacena el token y datos del usuario en AuthService.

```
public async Task<TokenDto?> Login(UserDto usuario)
{
    var dict = new Dictionary<string, string>
{
    { "username", usuario.Nombre },
    { "password", usuario.Contraseña }
};

    var content = new FormUrlEncodedContent(dict);
    var response = await _http.PostAsync("https://api-polideportivo.onrender.com/auth/login", content);

    if (response.IsSuccessStatusCode)
    {
        var token = await response.Content.ReadFromJsonAsync<TokenDto>();
        if (token != null)
        {
            _auth.IniciarSesion(token, usuario);
        }
        return token;
    }

    return null;
}
```

- Registro de usuario (**Register()**):
Permite crear una nueva cuenta de usuario.
    - Envía los datos a ***POST /auth/register***.
    - Retorna éxito o fallo de la operación.

```
public async Task<bool> Register(UserDto usuario)
{
    var response = await _http.PostAsJsonAsync("https://api-polideportivo.onrender.com/auth/register", usuario);
    return response.IsSuccessStatusCode;
}
```

- Subida de foto de perfil (**SubirFotoPerfilAsync()**):
Permite al usuario autenticado subir o actualizar su foto de perfil.
    - Envía un archivo tipo JPEG o PNG a ***PUT /auth/usuario/foto***.
    - Usa multipart/form-data con token JWT para autorización.

```
public async Task<bool> SubirFotoPerfilAsync(IBrowserFile file)
{
    if (string.IsNullOrEmpty(_auth.Token?.Access_Token))
        throw new InvalidOperationException("No autenticado.");

    var content = new MultipartFormDataContent();
    var stream = file.OpenReadStream(maxAllowedSize: 5 * 1024 * 1024);
    var fileContent = new StreamContent(stream);
    fileContent.Headers.ContentType = new MediaTypeHeaderValue(file.ContentType);

    content.Add(fileContent, "file", file.Name);

    var request = new HttpRequestMessage(HttpMethod.Put, "https://api-polideportivo.onrender.com/auth/usuario/foto");
    request.Content = content;
    request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", _auth.Token.Access_Token);

    var response = await _http.SendAsync(request);
    return response.IsSuccessStatusCode;
}
```

- Obtención del perfil de usuario (**ObtenerMiPerfilAsync()**):
Solicita la información del usuario autenticado.
    - Consume ***GET /auth/usuario/me***.
    - Retorna datos sin contraseña.

```
public async Task<UserDto> ObtenerMiPerfilAsync()
{
    var request = new HttpRequestMessage(HttpMethod.Get, "https://api-polideportivo.onrender.com/auth/usuario/me");
    request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", _auth.Token.Access_Token);

    var response = await _http.SendAsync(request);
    response.EnsureSuccessStatusCode();

    var json = await response.Content.ReadAsStringAsync();
    return JsonSerializer.Deserialize<UserDto>(json, new JsonSerializerOptions { PropertyNameCaseInsensitive = true })!;
}
```

· Gestión de sesión (AuthService):

- Administra el estado de autenticación del usuario en el frontend.
- Guarda el token JWT y los datos del usuario autenticado.
- Proporciona una propiedad IsAuthenticated para saber si hay un usuario autenticado (token no es null).

- Iniciar sesión (**IniciarSesion()**):
    - Recibe un token y un objeto usuario.
    - Guarda estos datos en la sesión activa.

- Cerrar sesión (**CerrarSesion()**):
    - Limpia el token y los datos del usuario, terminando la sesión.

```
public class AuthService
{
    12 referencias
    public TokenDto? Token { get; private set; }
    3 referencias
    public UserDto? Usuario { get; private set; }

    2 referencias
    public bool IsAuthenticated => Token != null;

    1 referencia
    public void IniciarSesion(TokenDto token, UserDto usuario)
    {
        Token = token;
        Usuario = usuario;
    }

    1 referencia
    public void CerrarSesion()
    {
        Token = null;
        Usuario = null;
    }
}
```

· Página Reservar (BuscarDisponibles()):

- Inicialización y limpieza:
    - Reinicia la lista pistasFiltradas y el mensaje de error mensajeError.
    - Limpia el diccionario disponibilidadMapa que almacena la disponibilidad de cada pista.

- Validación de horario:
    - Convierte las cadenas horaInicio y horaFin a horas enteras (inicioHora, finHora).
    - Comprueba que la hora final sea mayor que la hora inicial, mostrando un error y deteniendo la búsqueda si no es así.

- Verificación de datos previos:
    - Si la lista de pistas (pistas) es nula, termina la ejecución.

- Preparación para la búsqueda:
    - Activa el indicador cargando para mostrar que se está procesando la búsqueda.
    - Actualiza el estado de la interfaz con StateHasChanged().
    - Guarda las horas y fecha confirmadas para usarlas luego en la UI.

- Búsqueda y filtrado de pistas:
    - Inicializa una nueva lista para pistasFiltradas.
    - Recorre cada pista en la lista completa:
        -Si se ha seleccionado un tipo de pista, ignora las pistas que no coincidan.
        -Consulta la disponibilidad de la pista para la fecha seleccionada mediante **ApiService.GetDisponibilidadAsync().**
        - Genera la lista de horas necesarias entre el inicio y fin seleccionados.
        - Verifica que todas las horas necesarias estén disponibles en la respuesta.
        - Guarda el estado de disponibilidad en el diccionario disponibilidadMapa.
        - Si la pista está disponible para todo el rango, la añade a la lista filtrada.

- Finalización:
    - Desactiva el indicador cargando al terminar la búsqueda.

```
private async Task BuscarDisponibles()
{
    pistasFiltradas = null;
    mensajeError = null;
    disponibilidadMapa.Clear();

    var inicioHora = TimeSpan.Parse(horaInicio).Hours;
    var finHora = TimeSpan.Parse(horaFin).Hours;

    if (finHora <= inicioHora)
    {
        mensajeError = "La hora final debe ser mayor que la hora inicial.";
        return;
    }

    if (pistas is null) return;

    cargando = true;
    StateHasChanged();

    horaInicioConfirmada = horaInicio;
    horaFinConfirmada = horaFin;
    fechaConfirmada = fechaSeleccionada;

    pistasFiltradas = new();

    foreach (var pista in pistas)
    {
        if (!string.IsNullOrEmpty(tipoSeleccionado) && pista.Tipo != tipoSeleccionado)
            continue;

        var disponibilidad = await ApiService.GetDisponibilidadAsync(pista.Id, fechaConfirmada);

        var horasNecesarias = Enumerable.Range(inicioHora, finHora - inicioHora)
                                        .Select(h => fechaConfirmada.Date.AddHours(h))
                                        .ToList();

        bool disponible = horasNecesarias.All(h => disponibilidad.Contains(h));

        disponibilidadMapa[pista.Id] = disponible;

        if (disponible)
        {
            pistasFiltradas.Add(pista);
        }
    }

    cargando = false;
}
```

· Aplicación Móvil para Administradores:

· Comunicación con la API REST (RetrofitClient):

- Gestiona el consumo de los endpoints del backend del polideportivo mediante Retrofit.
- Configura la URL base de la API y convierte las respuestas JSON usando Gson.
- Integra el token JWT en las cabeceras para todas las operaciones protegidas.
- Implementa múltiples interfaces para dividir la lógica por dominio: autenticación, gestión de pistas y reservas.

```kotlin
object RetrofitClient {
    private const val BASE_URL = "https://api-polideportivo.onrender.com/"

    private val retrofit: Retrofit by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    val instance: PistaApi by lazy {
        retrofit.create(PistaApi::class.java)
    }

    val authService: AuthService by lazy {
        retrofit.create(AuthService::class.java)
    }

    val reservaApi: ReservaApi by lazy {
        retrofit.create(ReservaApi::class.java)
    }

}
```

· Autenticación (AuthService):

- Envía las credenciales del administrador (usuario y contraseña) al backend.
- Consume el endpoint ***POST /auth/login*** con datos tipo x-www-form-urlencoded.
- Retorna un objeto TokenResponse que incluye el token JWT.
- Este token se almacena localmente y se utiliza en las cabeceras de las siguientes llamadas API.
- Si el rol del usuario no es "admin", se restringe el acceso desde la aplicación.

```kotlin
interface AuthService {
    @FormUrlEncoded
    @POST("auth/login")
    fun login(
        @Field("username") username: String,
        @Field("password") password: String
    ): Call<TokenResponse>
}
```

· Gestión de pistas (PistaApi):

- Obtención de pistas (**obtenerPistas()**):
    - Recupera la lista completa de pistas registradas en el sistema.
    - Consume ***GET /api/pistas/*** con token JWT en la cabecera.
    - Retorna una lista de objetos Pista.

```kotlin
@GET("/api/pistas/")
fun obtenerPistas(
    @Header("Authorization") token: String
): Call<List<Pista>>
```

- Creación de pista (**crearPista()**):
    - Permite registrar una nueva pista desde la aplicación móvil.
    - Consume ***POST /api/pistas/*** enviando el objeto Pista en el cuerpo.
    - Requiere token válido de administrador.
    - Retorna la pista recién creada con todos sus campos.

```
@POST("/api/pistas/")
fun crearPista(
    @Header("Authorization") token: String,
    @Body pista: Pista
): Call<Pista>
```

- Actualización de pista (**actualizarPista()**):
    - Modifica los campos deseados de una pista existente.
    - Consume ***PUT /api/pistas/{id}*** con el objeto PistaUpdate.
    - No se sobrescriben campos no enviados.
    -Requiere autenticación como administrador.

```
@PUT("/api/pistas/{id}")
fun actualizarPista(
    @Header("Authorization") token: String,
    @Path("id") pistaId: Int,
    @Body pistaUpdate: PistaUpdate
): Call<Pista>
```

- Eliminación de pista (**eliminarPista()**):
    - Permite eliminar una pista por su ID.
    - Consume ***DELETE /api/pistas/{id}*** con token en la cabecera.
    - Retorna éxito (204) o error 404 si la pista no existe.

```
@DELETE("/api/pistas/{id}")
fun eliminarPista(
    @Header("Authorization") token: String,
    @Path("id") pistaId: Int
): Call<Void>
```

· Gestión de reservas (ReservaApi):

- Consulta de disponibilidad (**obtenerDisponibilidad()**):
    - Permite consultar las horas libres de una pista en una fecha concreta.
    - Consume ***GET /api/reservas/{pista_id}/disponibilidad?fecha=YYYY-MM-DD***.
    - Retorna un objeto Disponibilidad con la lista de horas libres.
    - Utilizado en interfaces de revisión y programación de reservas.

```
@GET("/api/reservas/{pista_id}/disponibilidad")
fun obtenerDisponibilidad(
    @Header("Authorization") token: String,
    @Path("pista_id") pistaId: Int,
    @Query("fecha") fecha: String
): Call<Disponibilidad>
```

- Creación de reserva (**reservar()**):
    - Permite crear una nueva reserva desde la aplicación.
    - Consume ***POST /api/reservas/*** enviando un objeto Reserva.
    - Requiere token de administrador.
    - Valida conflictos de horario antes de procesar la reserva.
    - Retorna la reserva confirmada o error si hay solapamiento.

```
@POST("/api/reservas/")
fun reservar(
    @Header("Authorization") token: String,
    @Body reservaCreate: Reserva
): Call<Reserva>
```

2.4. IMPLANTACIÓN

La implantación del sistema se compone de tres elementos principales: la API REST desarrollada en Python, la aplicación web desarrollada con Blazor Server y la aplicación móvil para administradores desarrollada en Android Studio con Kotlin:

- En la nube:

Tanto la API como la base de datos (PostgreSQL) y la aplicación web están desplegadas en la plataforma Render, que permite el despliegue de servicios web mediante archivos Dockerfile.

· API REST:
Accesible desde: https://api-polideportivo.onrender.com/docs
Render proporciona la gestión de variables de entorno y credenciales necesarias para el funcionamiento del backend.
· Base de datos PostgreSQL:
Se accede a través de herramientas como DBeaver, utilizando las credenciales proporcionadas por Render.
· Aplicación web (Blazor Server):
Disponible en: https://tfg-polideportivo.onrender.com

- En entorno local:

· API REST (FastAPI + SQLite):
Utilizando los comandos Docker que hay en el repositorio:
- La API se despliega mediante un contenedor Docker en el puerto 8000.
- Utiliza una base de datos SQLite, que genera un archivo database.db con los datos de la API.

· Aplicación web (Blazor Server):
- Se requiere Visual Studio 2022 con el SDK .NET 8 y al ejecutar la aplicación será accesible desde http://localhost:5087

· Aplicación móvil (Android):
- Se puede instalar directamente en un dispositivo Android conectado mediante ADB al ejecutar el proyecto desde Android Studio.

- Alternativamente, se puede generar un archivo .apk, el cual puede transferirse e instalarse en cualquier dispositivo Android que permita la instalación desde fuentes desconocidas.

2.5. DOCUMENTACIÓN

Se ha generado la siguiente documentación como parte del desarrollo del proyecto, incluida en los Anexos al final del documento:

- Casos de uso detallados para las principales funcionalidades de la aplicación, tanto en la parte de usuario como en la parte de administración desde la aplicación móvil.
- Descripción de los actores, flujos principales, alternativos, reglas de negocio, excepciones y condiciones asociadas a cada funcionalidad.
- Esta documentación sirve como referencia funcional y técnica, útil para la comprensión, mantenimiento y futuras ampliaciones del sistema.

Para más detalle, consultar los siguientes anexos:

Anexo 1: Registrar usuario
Anexo 2: Iniciar sesión
Anexo 3: Hacer una reserva
Anexo 4: Consultar mis reservas
Anexo 5: Cancelar reserva
Anexo 6: Añadir pista
Anexo 7: Editar pista
Anexo 8: Eliminar pista
Anexo 9: Gestionar horarios

3. CONCLUSIONES

Una vez finalizado el proyecto, puedo afirmar que ha supuesto un gran impulso en mi formación, tanto a nivel técnico como personal. A lo largo del desarrollo, consolidé conocimientos adquiridos durante el ciclo y aprendí y apliqué nuevas tecnologías que no había tratado en profundidad en clase.

Desde el punto de vista técnico, aprendí a crear aplicaciones web con Blazor, lo que me permitió mejorar considerablemente mi nivel en C#. También profundicé y entendí mejor el desarrollo de APIs con FastAPI en Python, mejorando así mis habilidades en desarrollo backend. Además, reforcé mi nivel en Kotlin, especialmente para acceder a una API. Por último, comprendí y apliqué los sistemas de autenticación usando JWT para crear usuarios con contraseñas almacenadas mediante hash.

Más allá de los aspectos técnicos, mejoré mi capacidad de organizar el tiempo, lo que me permitió realizar todo lo necesario para este proyecto y aprender cosas nuevas que podré emplear en un futuro.

3.1. RESULTADOS Y DISCUSIÓN

La temporalización real del proyecto se mantuvo bastante cercana a la planificada, aunque dediqué más tiempo al aprendizaje de nuevas tecnologías como Blazor y FastAPI.

Las principales dificultades surgieron en la gestión de la autenticación mediante JWT, la sincronización de reservas para evitar duplicados, el diseño de los componentes de la aplicación y la implementación del sistema para que las imágenes subidas se guardasen correctamente para cada usuario. Estos desafíos me permitieron mejorar mis habilidades técnicas y afrontar problemas reales de integración.

Finalmente, el proyecto cumplió con los objetivos funcionales previstos, y la integración entre la API, la aplicación web y la aplicación móvil resultó estable.

3.2. TRABAJO FUTURO

El proyecto ha alcanzado los objetivos principales planteados, ofreciendo una solución funcional para la reserva de pistas deportivas con gestión de usuarios, autenticación y control de disponibilidad.
Como posibles ampliaciones futuras, se podrían implementar notificaciones en tiempo real para avisar a los usuarios sobre cambios en sus reservas o en la disponibilidad, integrar métodos de pago para facilitar la gestión económica, mejorar la apariencia de la aplicación web para su uso en dispositivos móviles y añadir algunos detalles funcionales.

4. BIBLIOGRAFÍA

- *Guardrex. (n.d.). Tutoriales de ASP.NET Core Blazor. Microsoft Learn.* https://learn.microsoft.com/es-es/aspnet/core/blazor/tutorials/?view=aspnetcore-9.0
- *Mardem. (n.d.). Best books for beginners to learn Blazor? : r/Blazor.* https://www.reddit.com/r/Blazor/comments/10s100m/best_books_for_beginners_to_learn_blazor/
- *Curso de Introducción a #FastAPI 2024 - #Backend con #Python. (n.d.). YouTube.* https://www.youtube.com/playlist?list=PLHftsZss8mw7pSRpCyd-TM4Mu43XdyB3R
- *Andrés Guzmán Dev. (2024, April 24). Autenticación JWT en API REST: JSON Web Token [Video]. YouTube.* https://www.youtube.com/watch?v=i1IwZ2X82nc
- *Magaña, L. M. L. (2020, January 17). Qué es Json Web Token y cómo funciona. OpenWebinars.net.* https://openwebinars.net/blog/que-es-json-web-token-y-como-funciona
- *Init. Introducción a FastAPI Desde Cero Para Creación de API REST Con Python. YouTube. mayo 2023.* https://www.youtube.com/watch?v=12NIB_RjxMo
- *Joss Programming. (2023, October 2). Cómo Consumir APIs en Android con Retrofit y Kotlin: Guía Práctica [Video]. YouTube.* https://www.youtube.com/watch?v=5fyrlA-4msA
- *Cómo consumir una API desde una aplicación Android - Tutoriales. (2023, April 25). CodersLink.* https://coderslink.com/talento/blog/como-consumir-una-api-desde-una-aplicacion-android/

~ *Cómo obtener datos de Internet | Android Developers. (n.d.). Android Developers. https://developer.android.com/codelabs/basic-android-kotlin-compose-getting-data-internet?hl=es-419#0*
~ *render2web. Blazor desde Cero: Layouts, Páginas, Code Behind y Razor Explicado. YouTube. febrero 2025. https://www.youtube.com/watch?v=8ZXewRCUtho*
~ *Eric Roby. FastAPI JWT Tutorial | How to add User Authentication. YouTube. julio 2023. https://www.youtube.com/watch?v=0A_GCXBCNUQ*
~ *rithmic. (2021, October 24). FastAPI User Authentication with JWT [Video]. YouTube. https://www.youtube.com/watch?v=lKk_hudmAfE*

ANEXOS

I. ANEXO 1: Caso de uso – Registrar usuario

1. Nombre
Registrar usuario

2. Actor principal
Usuario nuevo

3. Objetivo
Permitir crear una nueva cuenta de usuario para acceder al sistema.

4. Descripción
El usuario introduce los datos requeridos (nombre y contraseña) y el sistema registra la cuenta.

5. Precondiciones
- El nombre de usuario no debe estar registrado previamente.
- Contraseña mínimo con 8 caracteres, una mayúscula y un número.

6. Postcondiciones
- Se crea la cuenta y el usuario puede iniciar sesión.

7. Flujo principal
1. El usuario accede al formulario de registro.
2. Introduce sus nombre y contraseña.
3. La aplicación envía los datos a la API.
4. La API valida y crea la cuenta.
5. La aplicación le deja acceder.

8. Flujos alternativos
- Nombre ya registrado: se muestra error.
- Contraseña inválida: se solicita corrección.

9. Reglas de negocio
La contraseña debe cumplir criterios mínimos de seguridad.
El nombre debe ser válido y único.

10. Excepciones
- Error de conexión: se informa al usuario.
- Error en servidor: se recomienda intentar más tarde.

II. ANEXO 2: Caso de uso – Iniciar sesión

1. Nombre
Iniciar sesión

2. Actor principal
Usuario

3. Objetivo
Permitir al usuario autenticarse en el sistema para acceder a funcionalidades privadas.

4. Descripción
El usuario introduce su nombre de usuario y contraseña. El sistema valida las credenciales y, si son correctas, le otorga acceso generando un token JWT.

5. Precondiciones
- El usuario debe tener una cuenta registrada.

6. Postcondiciones
- El usuario obtiene un token de autenticación válido para usar la aplicación.

7. Flujo principal
1. El usuario accede a la página de login.
2. El usuario introduce sus credenciales (usuario y contraseña).
3. La aplicación envía las credenciales a la API.
4. La API valida las credenciales.
5. La API responde con un token JWT si las credenciales son correctas.
6. La aplicación guarda el token y concede acceso a funcionalidades restringidas.

8. Flujos alternativos
- Credenciales incorrectas: se muestra mensaje de error.
- Usuario no registrado: se ofrece opción de registrarse.

9. Reglas de negocio
- Solo usuarios registrados pueden iniciar sesión.
- El token tiene un tiempo limitado de validez.

10. Excepciones
- Error de conexión: se informa al usuario y sugiere reintentar.
- Token expirado: el usuario debe iniciar sesión nuevamente.

III. ANEXO 3: Caso de uso – Hacer una reserva

1. Nombre
Hacer una reserva

2. Actor principal
Usuario autenticado

3. Objetivo
Permitir al usuario reservar una pista disponible en una fecha y horario específicos a través de la aplicación web.

4. Descripción
El usuario inicia sesión, selecciona una pista, consulta las horas disponibles para una fecha determinada, elige la hora o tramo horario y confirma la reserva. La aplicación registra la reserva y actualiza la disponibilidad.

5. Precondiciones
- El usuario debe estar autenticado con sesión activa y token válido.
- La pista debe existir en el sistema y tener horarios configurados.

6. Postcondiciones
- Se crea una reserva asociada al usuario y a la pista seleccionada.
- La disponibilidad de la pista para ese horario queda bloqueada para futuras reservas.

7. Flujo principal
1. El usuario navega a la sección de reservas y selecciona una pista.
2. La aplicación muestra un calendario o selector de fechas.
3. El usuario elige una fecha.
4. La aplicación consulta a la API las horas disponibles para esa pista y fecha.
5. El usuario selecciona la hora o rango horario deseado.
6. El usuario confirma la reserva.
7. La aplicación envía los datos a la API para registrar la reserva.
8. La API valida disponibilidad y guarda la reserva.
9. La aplicación muestra confirmación al usuario.

8. Flujos alternativos
- Horario no disponible: Si la hora ya no está libre, el sistema muestra un mensaje de error y solicita elegir otro horario.
- Usuario no autenticado: Si el usuario no ha iniciado sesión, se redirige a la página de login.
- Error de conexión o servidor: El sistema informa y ofrece intentar nuevamente.

9. Reglas de negocio
- Solo usuarios autenticados pueden reservar.
- La reserva debe ser dentro del horario permitido (8:00 a 22:00).
- No se permiten solapamientos de reservas para la misma pista.
- Duración mínima y máxima de reserva según configuración (ejemplo mínimo 1 hora).

10. Excepciones
- Token expirado o inválido: se redirige a login.
- Solicitud con datos inválidos: se informa al usuario y solicita corrección.
- Error interno del servidor: se informa al usuario y recomienda intentar luego.

IV. ANEXO 4: Caso de uso – Consultar mis reservas

1. Nombre
Consultar mis reservas

2. Actor principal
Usuario autenticado

3. Objetivo
Permitir al usuario ver todas sus reservas activas y pasadas.

4. Descripción
El usuario accede a su perfil y consulta la lista de reservas asociadas a su cuenta.

5. Precondiciones
- Usuario autenticado.
- Haber hecho alguna reserva.

6. Postcondiciones
- Se muestra al usuario la información actualizada de sus reservas.

7. Flujo principal
1. El usuario accede a la sección “Mis reservas”.
2. La aplicación solicita la lista de reservas a la API.
3. La API devuelve las reservas asociadas al usuario.
4. La aplicación muestra las reservas.

8. Flujos alternativos
-No hay reservas: se muestra mensaje indicándolo.

9. Reglas de negocio
- Solo se muestran las reservas del usuario autenticado.

10. Excepciones
- Error en la comunicación: mensaje de error y opción de reintentar.

V. ANEXO 5: Caso de uso – Cancelar reserva

1. Nombre
Cancelar reserva

2. Actor principal
Usuario autenticado

3. Objetivo
Permitir al usuario cancelar una reserva antes de su inicio.

4. Descripción
El usuario selecciona una reserva activa y solicita su cancelación, el sistema verifica que se puede cancelar y la elimina o marca como cancelada.

5. Precondiciones
- El usuario debe estar autenticado.

6. Postcondiciones
- La reserva queda cancelada y la disponibilidad de la pista se actualiza.

7. Flujo principal
1. El usuario accede a “Mis reservas”.
2. Selecciona la reserva que desea cancelar.
3. Solicita la cancelación.
4. El sistema cancela la reserva y actualiza disponibilidad.
5. Se notifica al usuario que la reserva ha sido cancelada.

8. Flujos alternativos
- Reserva ya cancelada: mensaje informativo.

9. Reglas de negocio
- Usuarios solo pueden cancelar sus propias reservas.

10. Excepciones
- Error de comunicación o servidor: se informa al usuario.

VI. ANEXO 6: Caso de uso – Añadir pista

1. Nombre
Añadir pista

2. Actor principal
Administrador autenticado con rol 'admin' en la aplicación móvil.

3. Objetivo
Permitir al administrador crear nuevas pistas en el sistema desde la aplicación móvil.

4. Descripción
El administrador accede a la opción para añadir pista, introduce los datos requeridos y crea una nueva pista en el sistema.

5. Precondiciones
- El usuario debe estar autenticado con rol 'admin'.
- El nombre de la pista debe ser único.

6. Postcondiciones
- Se crea la nueva pista y queda disponible en el sistema.

7. Flujo principal
1. El administrador abre la aplicación móvil y accede a “Añadir pista”.
2. Introduce los datos de la pista.
3. Envía los datos al sistema.
4. La API valida y crea la pista.
5. La aplicación confirma la creación.

8. Flujos alternativos
- Nombre de pista duplicado o datos inválidos: se muestra error y se solicita corrección.

9. Reglas de negocio
- Solo usuarios con rol 'admin' pueden añadir pistas.
- El nombre de la pista debe ser único.

10. Excepciones
- Error de conexión o servidor: se informa al usuario y se recomienda reintentar.

VII. ANEXO 7: Caso de uso – Editar pista

1. Nombre
Editar pista

2. Actor principal
Administrador autenticado con rol 'admin' en la aplicación móvil.

3. Objetivo
Permitir al administrador modificar los datos de una pista existente en el sistema.

4. Descripción
El administrador selecciona una pista existente, edita sus datos y guarda los cambios a través de la aplicación móvil.

5. Precondiciones
- El usuario debe estar autenticado con rol 'admin'.
- La pista debe existir en el sistema.

6. Postcondiciones
- Se actualizan los datos de la pista seleccionada en el sistema.

7. Flujo principal
1. El administrador accede a la lista de pistas.
2. Selecciona la pista que desea editar.
3. Modifica los datos necesarios.
4. Envía los cambios al sistema.
5. La API valida y guarda los cambios.
6. La aplicación confirma la actualización.

8. Flujos alternativos
- Datos inválidos: se muestra error y se solicita corrección.

9. Reglas de negocio
- Solo usuarios con rol 'admin' pueden editar pistas.
- No se puede duplicar el nombre con otra pista ya existente.

10. Excepciones
- Error de conexión o servidor: se informa al usuario y se recomienda reintentar.

VIII. ANEXO 8: Caso de uso – Eliminar pista

1. Nombre
Eliminar pista

2. Actor principal
Administrador autenticado con rol 'admin' en la aplicación móvil.

3. Objetivo
Permitir al administrador borrar una pista del sistema.

4. Descripción
El administrador selecciona una pista y solicita su eliminación. Si no tiene reservas activas, el sistema la elimina.

5. Precondiciones
- El usuario debe estar autenticado con rol 'admin'.
- La pista debe existir en el sistema.

6. Postcondiciones
- La pista es eliminada del sistema.

7. Flujo principal

1. El administrador accede a la lista de pistas.
2. Selecciona la pista a eliminar.
3. Solicita su eliminación.
4. Si todo es válido, se elimina la pista.
5. La aplicación confirma la eliminación.

8. Flujos alternativos

- Datos inválidos: se muestra error y se solicita corrección.

9. Reglas de negocio

- Solo usuarios con rol 'admin' pueden eliminar pistas.

10. Excepciones

- Error de conexión o servidor: se informa al usuario y se recomienda reintentar.

IX. ANEXO 9: Caso de uso – Gestionar horarios

1. Nombre

Gestionar horarios

2. Actor principal

Administrador autenticado con rol 'admin' en la aplicación móvil.

3. Objetivo

Permitir al administrador configurar los horarios disponibles para una pista.

4. Descripción

El administrador selecciona una pista, accede a su configuración de horarios y gestiona los horarios disponibles.

5. Precondiciones

- El usuario debe estar autenticado con rol 'admin'.
- La pista debe existir en el sistema.

6. Postcondiciones

- Se actualizan los horarios disponibles para la pista en el sistema.

7. Flujo principal

1. El administrador accede a la lista de pistas.
2. Selecciona la pista deseada.
3. Elimina horarios disponibles.
4. Envía los cambios al sistema.
5. La API valida y guarda los horarios.
6. La aplicación confirma la actualización.

8. Flujos alternativos
- Horarios inválidos o en conflicto: se muestra error y se solicita corrección.

9. Reglas de negocio
- Solo usuarios con rol 'admin' pueden gestionar horarios.
- No se pueden crear horarios superpuestos.

10. Excepciones
- Error de conexión o servidor: se informa al usuario y se recomienda reintentar.

X. ANEXO 10: Modelos y Schemas de la Aplicación

· Modelos:

```python
class Usuario(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    nombre: str = Field(index=True, unique=True)
    hashed_password: str
    rol: str = Field(default="usuario")
    foto_perfil: Optional[str] = None
```

```python
class Pista(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    nombre: str
    tipo: str
    precio_por_hora: float
```

```python
class Reserva(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    pista_id: int = Field(foreign_key="pista.id")
    usuario_id: int = Field(foreign_key="usuario.id")
    fecha_hora_inicio: datetime
    fecha_hora_fin: datetime

    class Config:
        arbitrary_types_allowed = True
```

· Schemas:

```python
class UsuarioCreate(BaseModel):
    nombre: str
    contraseña: str

class UsuarioOut(BaseModel):
    id: int
    nombre: str
    rol: str
    foto_perfil: Optional[str] = None

    class Config:
        orm_mode = True

class Token(BaseModel):
    access_token: str
    token_type: str
```

```python
class ReservaCreate(BaseModel):
    pista_id: int
    fecha_hora_inicio: datetime
    fecha_hora_fin: datetime

class ReservaOut(BaseModel):
    id: int
    pista_id: int
    usuario_id: int
    fecha_hora_inicio: datetime
    fecha_hora_fin: datetime
    nombre_pista: str
    precio_total: float

    class Config:
        orm_mode = True
```

```python
class PistaCreate(BaseModel):
    nombre: str
    tipo: str
    precio_por_hora: float

class PistaUpdate(BaseModel):
    nombre: Optional[str] = None
    tipo: Optional[str] = None
    precio_por_hora: Optional[float] = None
```