SISTEMA DE RESERVAS ADAPTABLE A CADA TIPO DE NEGOCIO

Author: Noel Yazdani

INDICE

4. Arquitectura y lógica de negocio

- Descripción del sistema de reservas
- Roles del sistema (admin / usuario)
- Entidades principales
    - User
    - Resource
    - Reservation
    - ResourceCategory
    - CustomFields
- Esquemas Pydantic (entrada/salida)
- Relaciones entre entidades
- Reglas de negocio
- Creación del usuario MySQL y base de datos
- Configuración de la conexión a la base de datos
    - Conexión de FastAPI
    - Conexión de Alembic
- Preparación de Alembic para detectar modelos
    - Configuración de `env.py`
    - Creación del paquete `models`
- Creación del primer modelo (User)
- Creación del esquema Pydantic (UserResponse)
- Generación de la primera migración
- Nota sobre sincronización de migraciones

5. Endpoints de la API

- Autenticación (JWT)
    - Registro de usuarios
    - Login
    - Generación y validación de tokens
    - Dependencia `get_current_user`

    - Dependencia `get_current_admin`
- Gestión de usuarios
    - Crear usuario (admin)
    - Obtener usuario por ID
    - Listar usuarios
    - Actualizar usuario
    - Eliminar usuario
    - Validaciones específicas (email único, roles, etc.)
- Gestión de recursos
    - Crear recurso
    - Listar recursos
    - Obtener recurso por ID
    - Actualizar recurso
    - Activar/desactivar recurso
    - Eliminar recurso
    - Gestión de categorías (ResourceCategory)
        - Crear categoría
        - Listar categorías
    - Gestión de campos personalizados (CustomFields)
        - Añadir campo personalizado
        - Eliminar campo personalizado
    - Gestión de reservas
    - Crear reserva
    - Listar reservas
    - Obtener reserva por ID
    - Cancelar reserva
    - Validaciones de disponibilidad
    - Reglas de negocio aplicadas en endpoints
- Validaciones y restricciones
    - Comprobación de disponibilidad de recursos
    - Restricciones por rol
    - Validación de fechas

    - Validación de existencia de entidades
    - Validación de campos personalizados
- Manejo de errores
    - Excepciones personalizadas
    - Respuestas estándar de error
    - Errores de autenticación
    - Errores de autorización
    - Errores de validación

1. Introducción

1.1 Objetivo del proyecto

El objetivo de este proyecto es desarrollar una **API REST profesional**, basada en **FastAPI**, que permita gestionar un **sistema de reservas genérico y configurable**, adaptable a distintos tipos de negocios (salas, pistas deportivas, mesas, recursos compartidos, etc.).

La API proporcionará:

- Gestión de usuarios y roles (admin / usuario).
- Gestión de recursos configurables.
- Creación, modificación y cancelación de reservas.
- Validación de disponibilidad.
- Autenticación segura mediante JWT.
- Estructura modular, escalable y mantenible.

El proyecto está diseñado para servir como base sólida para aplicaciones reales, integraciones con frontends modernos y despliegues en entornos productivos.

1.2 Descripción funcional

El sistema permitirá que un administrador configure los recursos que su negocio ofrece (por ejemplo, salas, pistas, mesas o cualquier elemento reservable). Los usuarios podrán consultar disponibilidad, realizar reservas y gestionar sus propias solicitudes.

El diseño será **genérico**, lo que significa que:

- No está limitado a un tipo concreto de negocio.
- Los recursos pueden tener campos personalizados.
- La lógica de disponibilidad será flexible.
- La API podrá integrarse con cualquier frontend o aplicación móvil.

1.3 Alcance inicial (MVP)

El MVP incluirá:

Usuarios

- Registro y login.
- Roles: admin y usuario.
- Gestión básica del perfil.

Recursos

- Creación, edición y eliminación de recursos por parte del admin.
- Campos base: nombre, descripción, tipo.
- Configuración opcional mediante JSON.

Reservas

- Crear reserva.
- Cancelar reserva.
- Consultar reservas propias.
- Validación de disponibilidad básica.

Autenticación

- Login con JWT.
- Protección de rutas según rol.

Documentación automática

- Swagger UI en `/docs`.
- Redoc en `/redoc`.

1.4 Extensiones futuras

El proyecto está diseñado para crecer. Algunas extensiones posibles:

- Campos personalizados por recurso (formularios dinámicos).
- Calendario visual de disponibilidad.
- Reglas avanzadas de horarios.
- Notificaciones por email.
- Panel de administración completo.
- Integración con frontend (React, Vue, Svelte…).
- Sistema de pagos.
- Reservas recurrentes.
- Integración con Google Calendar.

1.5 Tecnologías principales utilizadas

- **Python 3.12** – Lenguaje base.
- **FastAPI** – Framework para APIs REST.
- **Uvicorn** – Servidor ASGI para desarrollo.
- **SQLAlchemy** – ORM para modelos y consultas.
- **Alembic** – Migraciones de base de datos.
- **Pydantic** – Validación de datos.
- **JWT (python-jose)** – Autenticación.
- **passlib[bcrypt]** – Hashing seguro de contraseñas.
- **SQLite / MySQL** – Base de datos (decidiremos según conveniencia).
- **Git** – Control de versiones.

2. Instalación de dependencias

Esta sección describe los requisitos necesarios para preparar el entorno de desarrollo del proyecto, así como la instalación de las librerías y herramientas que utilizará la API.

2.1 Requisitos previos en el sistema (WSL2)

El proyecto requiere que el sistema tenga instaladas las siguientes herramientas:

Python 3.12

Lenguaje base para ejecutar FastAPI, SQLAlchemy, Alembic y el resto de dependencias del proyecto.

pip

Gestor de paquetes de Python. Permite instalar librerías dentro del entorno virtual del proyecto.

python3-venv

Herramienta para crear entornos virtuales aislados. Cada proyecto mantiene sus propias dependencias sin afectar al sistema.

Git

Control de versiones para gestionar el repositorio del proyecto.

2.2 Creación del directorio del proyecto

El proyecto se ubicará dentro del directorio general de desarrollo: ~/proyectos

Dentro de él se creará una carpeta específica para este proyecto FastAPI.

2.3 Creación del entorno virtual

Para aislar las dependencias del proyecto, se creará un entorno virtual llamado `venv`. Este entorno contendrá todas las librerías necesarias para FastAPI, sin afectar al sistema global.

Para generar el entorno: python3 -m venv venv

Para levantarlo: source venv/bin/activate

2.4 Instalación de dependencias del proyecto

Dentro del entorno virtual se instalarán las siguientes librerías:

FastAPI

Framework principal para construir la API REST.

Uvicorn

Servidor ASGI para ejecutar la aplicación en desarrollo.

SQLAlchemy

ORM para definir modelos, relaciones y consultas a la base de datos.

Alembic

Sistema de migraciones para versionar cambios en la base de datos.

PyMySQL

Driver necesario para que SQLAlchemy pueda conectarse a MySQL. Sin este paquete, Alembic y SQLAlchemy no pueden establecer conexión con la base de datos.

python-jose

Generación y validación de tokens JWT para autenticación.

passlib[bcrypt]

Hashing seguro de contraseñas.

bcrypt 4.0.1

Versión estable y compatible con Passlib en Python 3.12.

passlib[bcrypt]

Reinstalación recomendada para asegurar que Passlib detecte el backend correcto.

```
pip install fastapi uvicorn sqlalchemy alembic python-jose passlib[bcrypt]
pymysql bcrypt==4.0.1
```

2.5 Generación del archivo requirements.txt

Una vez instaladas las dependencias, se generará un archivo `requirements.txt` que contendrá la lista exacta de librerías utilizadas por el proyecto. Este archivo permite reproducir el entorno en otros equipos o servidores.

```
pip freeze > requirements.txt
```

3. Configuración del entorno

Esta sección describe la estructura interna del proyecto, la configuración de la base de datos, la inicialización de SQLAlchemy y Alembic, y la preparación del servidor de desarrollo.

3.1 Estructura de carpetas del proyecto

FastAPI no crea una estructura de directorio de manera automática, tendremos que hacerlo manual.

La estructura inicial será:

```
sistema-reservas/
│
├── app/
│   ├── main.py        Punto de entrada de la aplicación FastAPI.
│   ├── api/           Contendrá los routers (endpoints).
│   ├── models/        Modelos SQLAlchemy
│   ├── schemas/       Modelos Pydantic para validar entrada/salida
│   ├── core/          Configuración general, constantes, utilidades.
│   ├── services/      Lógica de negocio
│   ├── database.py    Configuración de SQLAlchemy y conexión a la base de datos.
│   └── security.py    Hashing, JWT, autenticación.
│
├── alembic/           Carpeta generada por Alembic para migraciones.
├── alembic.ini        Archivo de configuración de Alembic.
├── venv/
├── requirements.txt
└── README.md
```

3.2 Configuración de la base de datos (MySQL)

Usaremos MySQL porque:

- soporta concurrencia
- es ideal para un sistema de reservas
- SQLAlchemy funciona muy bien con él

La base de datos se creará manualmente en MySQL antes de aplicar migraciones.

Cadena de conexión (DATABASE_URL)

SQLAlchemy usa una URL con este formato:

mysql+pymysql://usuario:password@localhost:3306/nombre_base_datos

Esta URL se usará en:

- `app/database.py`
- `alembic.ini`

3.3 Configuración de SQLAlchemy

Crearemos un archivo `app/database.py` que contendrá:

- motor de conexión (`create_engine`)
- sesión de base de datos (`SessionLocal`)
- base declarativa (`Base`)

3.4 Configuración de Alembic

inicializamos Alembic: alembic init alembic

Esto creará automáticamente: alembic/ alembic.ini

Alembic necesita:

- un archivo `alembic.ini`
- una carpeta `alembic/`
- una referencia a `Base` para detectar modelos

Alembic permitirá:

- crear migraciones
- aplicar cambios en la base de datos
- versionar el esquema

3.5 Variables de entorno

Para no exponer contraseñas, usaremos un archivo `.env` con:

```
DATABASE_URL=mysql+pymysql://reservas:clave_segura@localhost:3306/sistema_reservas
SECRET_KEY=clave_para_jwt
ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=30
```

FastAPI no tiene `.env` nativo como Symfony, pero usaremos `python-dotenv` o `pydantic-settings` más adelante.

3.6 Ejecución del servidor

Una vez configurado todo, y previamente levantado el entorno, el servidor se ejecutará con:

```
uvicorn app.main:app --reload
```

Esto levantará la API en: http://localhost:8000

Y la documentación automática en: http://localhost:8000/docs

4. Arquitectura y lógica de negocio

Este apartado describe la arquitectura funcional y técnica del sistema de reservas, las entidades que lo componen, las reglas de negocio que lo gobiernan y la infraestructura necesaria para que FastAPI, SQLAlchemy y Alembic trabajen de forma coherente. También documenta la estructura de la base de datos, la configuración del entorno y la preparación del sistema para migraciones automáticas.

4.1 Descripción general del sistema

El sistema implementa una API REST para gestionar recursos reservables (salas, equipos, vehículos, aulas, etc.) y las reservas asociadas a ellos. Incluye autenticación mediante JWT, autorización basada en roles y validaciones de disponibilidad.

Los usuarios pueden:

- consultar recursos
- realizar reservas
- cancelar sus propias reservas

Los administradores pueden:

- gestionar recursos
- gestionar categorías
- gestionar usuarios
- ver y cancelar cualquier reserva

4.2 Roles del sistema

Usuario (role = "user")

- Ver recursos activos
- Crear reservas
- Ver sus propias reservas
- Cancelar sus propias reservas
- Actualizar su email y contraseña

Administrador (role = "admin")

- Crear, actualizar y eliminar recursos
- Crear, actualizar y eliminar categorías
- Ver todas las reservas
- Cancelar cualquier reserva
- Listar, actualizar y eliminar usuarios
- Acceso completo al sistema

4.3 Entidades principales

User

Representa a un usuario autenticado del sistema. Atributos:

- id
- email
- hashed_password
- role

Resource

Recurso reservable. Atributos:

- id
- name
- description
- is_active
- category_id

Reservation

Reserva realizada por un usuario sobre un recurso. Atributos:

- id
- user_id
- resource_id
- start_time
- end_time
- status

ResourceCategory

Clasificación de recursos. Atributos:

- id
- name

CustomFields

Campos personalizados asociados a un recurso. Atributos:

- id
- resource_id
- key
- value

4.4 Esquemas Pydantic

Cada entidad tiene:

- **Schemas de entrada** (para crear/actualizar)
- **Schemas de salida** (para devolver datos al cliente)

Ejemplo:

```
class ResourceCreate(BaseModel):
    name: str
    description: str | None = None
```

Los schemas de salida usan:

```
class Config:
    from_attributes = True
```

para convertir automáticamente desde modelos SQLAlchemy.

4.5 Relaciones entre entidades

- **User 1—N Reservation**
- **Resource 1—N Reservation**
- **Resource N—1 ResourceCategory**
- **Resource 1—N CustomFields**

4.6 Reglas de negocio

Reservas

- `start_time < end_time`
- El recurso debe existir
- El recurso debe estar activo
- No puede haber solapamiento de reservas
- Un usuario solo puede cancelar sus propias reservas
- Un admin puede cancelar cualquier reserva

Usuarios

- El email debe ser único
- Un usuario solo puede modificar su propio perfil
- Un admin puede modificar cualquier usuario

Categorías

- No se permiten nombres duplicados
- Solo los administradores pueden crearlas, modificarlas o eliminarlas

4.7 Configuración de MySQL

Creación del usuario y base de datos:

Entrar como root en MySQL: mysql -u root -p

Crear el usuario: CREATE USER 'fastapi'@'localhost' IDENTIFIED BY 'Examen123';

Crear la base de datos: CREATE DATABASE sistema_reservas;

Da permisos al usuario: GRANT ALL PRIVILEGES ON sistema_reservas.* TO 'fastapi'@'localhost';

Refresca permisos: FLUSH PRIVILEGES;

4.8 Configuración de la conexión a la base de datos

FastAPI y Alembic **no comparten la misma configuración**, por lo que la cadena de conexión debe definirse en dos lugares distintos.

4.8.1 Conexión de FastAPI (`app/database.py`)

DATABASE_URL = "mysql+pymysql://fastapi:Examen123@localhost:3306/sistema_reservas"

Gestiona:

- engine
- sesiones
- dependencia `get_db`

Es la conexión **en tiempo de ejecución**.

4.8.2 Conexión de Alembic (`alembic.ini`)

Modificar:

sqlalchemy.url = driver://user:pass@localhost/dbname

Por:

sqlalchemy.url = mysql+pymysql://fastapi:Examen123@localhost:3306/sistema_reservas

Alembic **no usa** `database.py`, por lo que esta configuración es obligatoria.

Resumen de responsabilidades

Archivo	Lo usa	Función
`app/database.py`	FastAPI	Conexión real de la API
`alembic.ini`	Alembic	Migraciones y versionado

4.9 Preparar Alembic para detectar modelos

Alembic necesita acceder a los modelos SQLAlchemy para generar migraciones automáticas.

4.9.1 Modificar `alembic/env.py`

Sustituir:

```
target_metadata = None
```

Por:

```
from app.database import Base
from app import models  # Importa todos los modelos del paquete
target_metadata = Base.metadata
```

Esto permite que Alembic lea todos los modelos registrados en `Base`.

4.9.2 Crear el paquete `models`

Crear archivo:

```
touch app/models/__init__.py
```

Contenido:

```
from .user import User
```

Cada vez que se cree un nuevo modelo, se añadirá aquí.

Esto evita tener que modificar `env.py` en el futuro.

4.10 Crear el primer modelo: User

El resto de modelos(tablas) se creara de la misma manera-

Crear archivo: touch app/models/user.py

Asignaremos Rol : User por defecto.

Codigo:

```
from sqlalchemy import Column, Integer, String
from app.database import Base
# Modelo User: representa la tabla "users" en la base de datos
class User(Base):
    __tablename__ = "users"  # Nombre de la tabla
    id = Column(Integer, primary_key=True, index=True)
```

```
    # Campo "email": texto, único, obligatorio
    email = Column(String(255), unique=True, nullable=False)
    # Campo "hashed_password": contraseña encriptada
    hashed_password = Column(String(255), nullable=False)
    # Campo "role": admin o user
    role = Column(String(50), default="user")
```

4.11 Crear el esquema Pydantic

Los schemas son equivalentes a:

- DTOs/Formularios validados
- Request/response models

Crear archivo: touch app/schemas/user.py

Codigo:

```
from pydantic import BaseModel
# Esquema para mostrar usuarios (respuesta)
class UserResponse(BaseModel):
    id: int
    email: str
    role: str
    class Config:
        from_attributes = True  # Permite convertir desde SQLAlchemy
```

Este esquema define la estructura de respuesta para la API.

4.12 Crear la primera migración

Ahora que tenemos un modelo real, creamos la migración:

```
alembic revision --autogenerate -m "create users table"
```

Y aplicamos la migración:

```
alembic upgrade head
```

Esto creará la tabla `users` en MySQL.

Si se elimina una migración manualmente, es necesario limpiar la tabla `alembic_version` en MySQL: TRUNCATE alembic_version;

5. Endpoints de la API

5.1 Autenticación (JWT)

El sistema utiliza autenticación basada en **JSON Web Tokens (JWT)** para proteger los endpoints y gestionar permisos. Incluye:

- registro de usuarios
- inicio de sesión
- generación de tokens JWT
- verificación de tokens
- dependencias para obtener el usuario actual
- validación de rol administrador

Componentes principales:

- `app/core/security.py` → hashing y JWT
- `app/schemas/auth.py` → schemas de login y token
- `app/dependencies/auth.py` → dependencias de seguridad
- `app/routers/auth.py` → endpoints de autenticación

La autenticación se basa en:

- **hashing de contraseñas** con Passlib
- **tokens JWT** firmados con clave secreta
- **dependencias de seguridad** para obtener el usuario actual
- **roles** para restringir acceso

5.1.1 Archivo de seguridad: `app/core/security.py`

Este archivo centraliza toda la lógica de seguridad:

- hash de contraseñas
- verificación
- creación de tokens
- expiración
- firma JWT

5.1.2 Schemas de autenticación: `app/schemas/auth.py`

Estos schemas definen:

- los datos que recibe el login

- la respuesta con el token

5.1.3 Dependencias de autenticación: `app/dependencies/auth.py`

Aquí resolvemos:

- extraer token del header
- decodificarlo
- cargar usuario desde la base de datos
- validar rol admin

5.1.4 Router de autenticación: `app/routers/auth.py`

Endpoints:

- `/auth/register` → registrar usuario
- `/auth/login` → obtener token

5.1.5 Registrar router en `main.py`

```
# main.py
from fastapi import FastAPI
from app.routers import auth

app = FastAPI()
app.include_router(auth.router)
```

5.2 Gestión de usuarios

La API expone endpoints para consultar y administrar usuarios:

- `GET /users/me` → devuelve el usuario autenticado (requiere token válido).
- `GET /users/` → lista todos los usuarios (requiere rol admin).
- `GET /users/{id}` → devuelve un usuario por id (requiere rol admin).
- `DELETE /users/{id}` → elimina un usuario (requiere rol admin).

Estos endpoints usan:

- `get_current_user` para obtener el usuario autenticado.
- `get_current_admin` para restringir operaciones a administradores.
- `UserResponse` como esquema de salida.

5.2.1 Router de usuarios:

En `app/routers/users.py`

5.2.2 Registrar el router:

En `main.py`:

Añadimos users al import e incluimos su ruta

Asegúrate de tener `app/routers/__init__.py` con:

```
# app/routers/__init__.py

from . import auth
from . import users
```

5.3 Gestión de recursos

Este apartado describe la gestión de los recursos del sistema, incluyendo:

- **Resource** (entidad principal reservable)
- **ResourceCategory** (clasificación de recursos)
- **CustomFields** (atributos personalizados por recurso)

Los endpoints asociados permiten realizar operaciones CRUD y aplicar reglas de negocio relacionadas con disponibilidad, clasificación y extensibilidad de los recursos.

5.3.1 Gestión de recursos (Resource)

Los recursos representan los elementos que pueden ser reservados por los usuarios: salas, vehículos, equipos, aulas, etc.

Los endpoints asociados permiten:

- **Crear recursos**
    - Solo accesible para administradores
    - Permite asignar una categoría opcional
    - Permite definir descripción y estado activo/inactivo
- **Listar recursos**
    - Acceso general (requiere autenticación)
    - Devuelve todos los recursos con sus categorías y campos personalizados

- **Obtener un recurso por ID**
    - Acceso general
    - Incluye categoría y campos personalizados
- **Actualizar recursos**
    - Solo administradores
    - Permite modificar nombre, descripción, categoría y estado
- **Eliminar recursos**
    - Solo administradores
    - Elimina el recurso y sus campos personalizados asociados

Validaciones aplicadas

- El recurso debe existir
- Si se especifica una categoría, esta debe existir
- Solo administradores pueden crear, modificar o eliminar recursos
- Los recursos inactivos no pueden ser reservados (regla aplicada en el módulo de reservas)

Relaciones implicadas

- Un recurso puede pertenecer a una categoría (`ResourceCategory`)
- Un recurso puede tener múltiples campos personalizados (`CustomFields`)
- Un recurso puede tener múltiples reservas (`Reservation`)

5.3.2 Gestión de categorías (ResourceCategory)

Las categorías permiten clasificar los recursos en grupos lógicos, como:

- “Salas”
- “Vehículos”
- “Equipos”
- “Aulas”

Los endpoints asociados permiten:

- **Crear categorías**
    - Solo administradores
    - Valida que el nombre no exista previamente
- **Listar categorías**
    - Acceso general

- Devuelve todas las categorías registradas

Validaciones aplicadas

- No se permite crear categorías duplicadas
- Solo administradores pueden crear nuevas categorías

Relaciones implicadas

- Una categoría puede tener múltiples recursos asociados
- La eliminación de categorías no se implementa en este punto (opcional según reglas del negocio)

5.3.3 Gestión de campos personalizados (CustomFields)

Los campos personalizados permiten extender la información de un recurso sin modificar la estructura de la base de datos. Ejemplos:

- capacidad
- ubicación
- número de serie
- marca
- modelo

Los endpoints asociados permiten:

- **Añadir un campo personalizado a un recurso**
  - Solo administradores
  - Requiere clave y valor
  - Valida que el recurso exista
- **Eliminar un campo personalizado**
  - Solo administradores
  - Valida que el campo exista y pertenezca al recurso indicado

Validaciones aplicadas

- El recurso debe existir
- El campo debe existir y pertenecer al recurso
- Solo administradores pueden gestionar campos personalizados

Relaciones implicadas

- Un recurso puede tener múltiples campos personalizados
- Los campos personalizados se eliminan automáticamente si se elimina el recurso

5.3.4 Registro de routers

Los routers correspondientes a este módulo son:

- `app/routers/resources.py`
- `app/routers/categories.py`

Ambos deben registrarse en `main.py` para que FastAPI exponga sus endpoints.

5.4 Gestión de reservas

Este apartado define la lógica completa para gestionar reservas dentro del sistema. Una reserva representa la ocupación de un recurso por parte de un usuario durante un intervalo de tiempo.

Los endpoints permiten:

- crear reservas
- listar reservas
- obtener reservas por ID
- cancelar reservas

Todo ello aplicando las reglas de negocio necesarias para garantizar la integridad del sistema.

5.4.1 Reglas de negocio aplicadas

La creación y gestión de reservas está sujeta a las siguientes reglas:

Disponibilidad del recurso

- El recurso debe existir.
- El recurso debe estar activo (`is_active = True`).
- No puede existir otra reserva que se solape en el mismo intervalo.

Validación de fechas

- `start_time` debe ser estrictamente menor que `end_time`.
- Las fechas deben ser válidas y en formato datetime.

Permisos

- Un usuario solo puede ver y cancelar sus propias reservas.
- Un administrador puede ver y cancelar cualquier reserva.

Integridad de datos

- La reserva debe estar asociada a un usuario existente.
- La reserva debe estar asociada a un recurso existente.

5.4.2 Endpoints principales

Crear reserva

Permite a un usuario reservar un recurso en un intervalo de tiempo. Incluye validación de solapamientos y disponibilidad.

Listar reservas

- Administradores: ven todas las reservas.
- Usuarios: solo ven sus propias reservas.

Obtener reserva por ID

- Administradores: acceso total.
- Usuarios: solo si la reserva les pertenece.

Cancelar reserva

- Administradores: pueden cancelar cualquier reserva.
- Usuarios: solo pueden cancelar las suyas.

5.4.3 Archivos involucrados

- `app/routers/reservations.py` Contiene todos los endpoints relacionados con reservas.
- `app/schemas/reservation.py` Define el esquema de salida para las reservas.
- `app/models/reservation.py` Modelo SQLAlchemy que representa la tabla `reservations`.
- `app/dependencies/auth.py` Gestiona permisos y autenticación para determinar qué usuario realiza la acción.
- `app/database.py` Proporciona la sesión de base de datos necesaria para operar.

5.4.4 Relaciones entre entidades

- **User 1—N Reservation** Un usuario puede tener múltiples reservas.
- **Resource 1—N Reservation** Un recurso puede tener múltiples reservas asociadas.

Estas relaciones permiten:

- filtrar reservas por usuario
- validar solapamientos por recurso
- aplicar reglas de permisos

5.5 Validaciones y restricciones

Este apartado describe de forma centralizada las validaciones y restricciones que se aplican en los distintos endpoints de la API. Aunque la lógica se implementa dentro de los routers correspondientes, es importante documentar estas reglas como parte de la arquitectura del sistema.

Las validaciones se agrupan en cuatro bloques principales:

- Validaciones de autenticación y autorización
- Validaciones de recursos
- Validaciones de reservas
- Validaciones de integridad de datos

5.5.1 Validaciones de autenticación y autorización

Estas validaciones se aplican principalmente en:

- `app/dependencies/auth.py`
- `app/routers/auth.py`
- `app/routers/users.py`
- `app/routers/resources.py`
- `app/routers/reservations.py`

Autenticación

- Los endpoints protegidos requieren un token JWT válido.
- Si el token es inválido, ha expirado o no se puede decodificar, se devuelve un error `401 Unauthorized`.
- La obtención del usuario actual se realiza mediante la dependencia `get_current_user`.

Autorización por rol

- La dependencia `get_current_admin` valida que el usuario tenga rol `admin`.
- Solo los administradores pueden:
    - crear, actualizar y eliminar recursos
    - crear categorías
    - gestionar campos personalizados
    - listar todos los usuarios
    - listar todas las reservas
    - cancelar reservas de otros usuarios

- Los usuarios con rol `user` solo pueden:
    - ver sus propios datos
    - ver recursos
    - crear reservas
    - ver sus propias reservas
    - cancelar sus propias reservas

5.5.2 Validaciones de recursos

Aplicadas principalmente en:

- `app/routers/resources.py`
- `app/routers/reservations.py`

Existencia del recurso

- Antes de crear una reserva, se valida que el recurso exista.
- Antes de actualizar o eliminar un recurso, se valida que exista.
- Si el recurso no existe, se devuelve `404 Not Found`.

Estado del recurso

- Un recurso inactivo (`is_active = False`) no puede ser reservado.
- Esta validación se aplica en la creación de reservas.
- Si el recurso está inactivo, se devuelve `400 Bad Request`.

Categorías

- Si se especifica `category_id` al crear o actualizar un recurso, se valida que la categoría exista.
- Si la categoría no existe, se devuelve `404 Not Found`.
- No se permite crear categorías duplicadas (mismo nombre).

5.5.3 Validaciones de reservas

Aplicadas principalmente en:

- `app/routers/reservations.py`

Validación de fechas

- `start_time` debe ser estrictamente menor que `end_time`.
- Si no se cumple, se devuelve `400 Bad Request`.

Solapamiento de reservas

- Antes de crear una reserva, se comprueba si ya existe otra reserva para el mismo recurso cuyo intervalo se solape.
- La condición de solapamiento es:

  > Una reserva A se solapa con B si: `A.start_time < B.end_time` y `A.end_time > B.start_time`

- Si existe solapamiento, se devuelve `409 Conflict`.

Permisos sobre reservas

- Un usuario solo puede ver y cancelar sus propias reservas.
- Un administrador puede ver y cancelar cualquier reserva.
- Si un usuario intenta acceder o cancelar una reserva que no le pertenece, se devuelve `403 Forbidden`.

5.5.4 Validaciones de integridad de datos

Estas validaciones garantizan que las relaciones entre entidades sean coherentes.

Usuario existente

- La creación de reservas siempre se hace en el contexto de un usuario autenticado (`get_current_user`).
- No se permite crear reservas sin usuario asociado.

Relaciones entre entidades

- `Reservation.user_id` debe apuntar a un usuario existente.
- `Reservation.resource_id` debe apuntar a un recurso existente.
- `CustomField.resource_id` debe apuntar a un recurso existente.

Si alguna de estas entidades no existe, se devuelve `404 Not Found`.

5.5.5 Resumen de responsabilidades

- `app/dependencies/auth.py`
  - Valida tokens
  - Obtiene el usuario actual
  - Valida rol admin
- `app/routers/resources.py`
  - Valida existencia de recursos
  - Valida existencia de categorías
  - Restringe operaciones a administradores
- `app/routers/reservations.py`
  - Valida existencia y estado del recurso
  - Valida fechas
  - Valida solapamientos
  - Valida permisos sobre reservas
- `app/routers/users.py`
  - Restringe acceso a datos de otros usuarios a administradores

5.6 Manejo de errores

El sistema implementa un conjunto de mecanismos para gestionar errores de forma consistente, clara y segura. El objetivo es garantizar que la API devuelva mensajes comprensibles y códigos HTTP adecuados ante situaciones inesperadas o inválidas.

El manejo de errores se basa en:

- excepciones estándar de FastAPI (`HTTPException`)
- validaciones dentro de los routers
- dependencias de autenticación
- un módulo opcional de excepciones reutilizables
- un manejador global opcional para errores no controlados

5.6.1 Errores estándar mediante HTTPException

FastAPI proporciona la clase `HTTPException`, que se utiliza en toda la API para devolver errores con:

- código de estado HTTP
- mensaje descriptivo
- encabezados opcionales

Ejemplos comunes:

- `404 Not Found` → entidad no encontrada
- `400 Bad Request` → datos inválidos
- `401 Unauthorized` → token inválido
- `403 Forbidden` → permisos insuficientes
- `409 Conflict` → solapamiento de reservas

Estas excepciones se utilizan en:

- `auth.py`
- `users.py`
- `resources.py`
- `categories.py`
- `reservations.py`

5.6.2 Validaciones que generan errores

Los errores no se generan de forma arbitraria, sino como resultado de validaciones explícitas:

Autenticación

- Token inválido → `401 Unauthorized`
- Usuario inexistente → `401 Unauthorized`

Autorización

- Usuario sin rol admin intentando acceder a un endpoint restringido → `403 Forbidden`

Recursos

- Recurso inexistente → `404 Not Found`
- Categoría inexistente → `404 Not Found`
- Recurso inactivo → `400 Bad Request`

Reservas

- Fechas inválidas → `400 Bad Request`
- Solapamiento de reservas → `409 Conflict`
- Reserva inexistente → `404 Not Found`
- Usuario intentando acceder a reservas ajenas → `403 Forbidden`

5.6.3 Excepciones reutilizables

El archivo `app/core/exceptions.py` define funciones que generan errores estándar reutilizables. Esto permite:

- mantener consistencia en los mensajes
- evitar duplicación de código
- facilitar futuras ampliaciones

5.6.4 Manejador global de errores

El sistema puede incluir un manejador global para capturar errores no controlados y devolver un mensaje uniforme:

- evita que se filtren trazas internas
- mejora la experiencia del cliente
- mantiene la API consistente

Este manejador no reemplaza las `HTTPException`, solo actúa como "última capa de seguridad".

5.6.5 Resumen de responsabilidades

- **Routers** → validan datos y lanzan errores específicos
- **Dependencias de autenticación** → gestionan errores de permisos
- **Excepciones personalizadas** → facilitan consistencia
- **Manejador global** → captura errores inesperados

NoelYTejerina