

Departamento de Informática

Desarrollo de Aplicaciones con Bases de Datos Documentales

Juan Gualberto Gutiérrez Marín

Febrero 2025

AVISO: El texto contiene muchos enlaces embebidos que sólo funcionan si se abre el PDF desde un lector digital. Si se imprime, se perderá mucha información interesante. No obstante, en el apartado “Bibliografía” se han añadido los enlaces para aquellos lectores que deseen una copia impresa y puedan ver la información.

Este documento se encuentra bajo una licencia Creative Commons de Atribución-CompartirIgual (CC BY-SA).

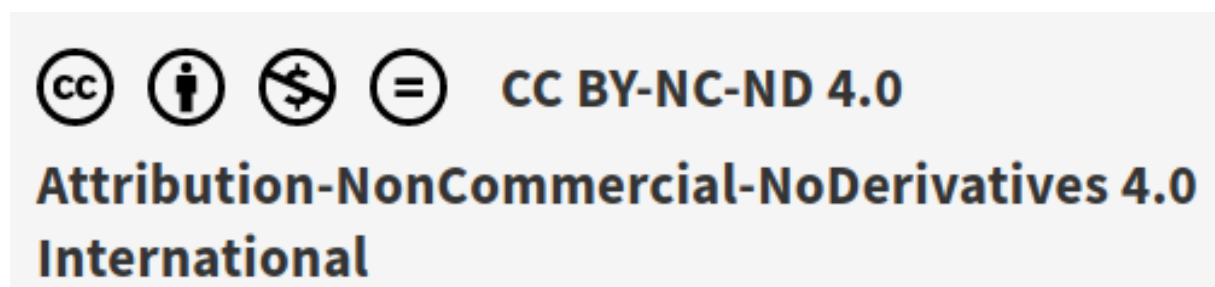


Figura 1: Atribución-CompartirIgual (CC BY-SA)

Esto significa que puedes:

- Compartir: copiar y redistribuir el material en cualquier medio o formato.
- Adaptar: remezclar, transformar y construir sobre el material para cualquier propósito, incluso comercialmente.

Bajo las siguientes condiciones:

- Atribución: debes dar crédito de manera adecuada, proporcionar un enlace a la licencia e indicar si se han realizado cambios. Puedes hacerlo de cualquier manera razonable, pero no de una manera que sugiera que el licenciante te respalda a ti o al uso que hagas del trabajo.
- Compartir igual: si remezclas, transformas o creas a partir del material, debes distribuir tus contribuciones bajo la misma licencia que el original.

Para más detalles, consulta la licencia completa.

Para una versión actualizada de este libro visita esta Web: <https://gitlab.iesvirgendelcarmen.com/juangu/adtd07-flask-mongo-praetorian-reservas>.

Índice

1	Desarrollo de Aplicaciones con Bases de Datos Documentales	4
1.1	Repaso JSON	5
1.2	JSON Schema	6
2	Instalación de MongoDB	8
2.1	Conexión interactiva a MongoDB	9
2.2	Colección de ejemplo	11
3	Segundo ejemplo completo con MongoDB	13
3.1	1. Modelo de Datos del Ejemplo	15
3.2	2. Ejemplos de Operaciones CRUD	15
3.2.1	2.1. Seleccionar la Base de Datos	15
3.2.2	2.2. Crear (Insertar Documentos)	15
3.2.3	2.3. Leer (Consultas)	16
3.2.4	2.4. Actualizar	17
3.2.5	2.5. Borrar	17
3.3	Objetos Embebidos vs. Objetos Anidados	18
3.3.1	Objetos Embebidos	18
4	Mi primera aplicación Flask	19
5	Autenticación JWT con Praetorian y MongoEngine en Flask	20
5.1	Convirtiendo de MySQL a JSON	21
5.2	Instalar dependencias necesarias	22
5.3	Configurar Flask con MongoEngine	22
5.4	Definir el modelo de usuario	23
5.5	Inicializar Praetorian con MongoEngine	23
5.6	Crear rutas de autenticación	23

1 Desarrollo de Aplicaciones con Bases de Datos Documentales

Las bases de datos documentales son un tipo de sistema de gestión de bases de datos (SGBD) que se centra en el almacenamiento y recuperación de datos en formato de documentos. En lugar de organizar la información en tablas como en las bases de datos relacionales, las bases de datos documentales utilizan documentos, que pueden ser en formatos como JSON (JavaScript Object Notation) o BSON (Binary JSON). Cada documento puede contener datos estructurados y no estructurados, lo que proporciona flexibilidad en la representación de la información.

Mientras que las bases de datos relacionales están más recomendadas a casos en los que hay mucha rotación de datos, las documentales se recomiendan para los casos donde hay pocas o nulas actualizaciones.

Características comunes de las bases de datos documentales:

1. **Documentos:** La unidad básica de almacenamiento es el documento, que puede contener datos en formato clave-valor, matrices, objetos anidados, etc.
2. **Esquema dinámico:** A diferencia de las bases de datos relacionales, las bases de datos documentales permiten esquemas dinámicos, lo que significa que cada documento en la colección puede tener diferentes campos.
3. **Flexibilidad:** Son adecuadas para datos semiestructurados y no estructurados, lo que facilita el almacenamiento de información variada y cambiante.
4. **Escalabilidad horizontal:** Muchas bases de datos documentales están diseñadas para escalar horizontalmente, distribuyendo la carga de trabajo en varios servidores para manejar grandes volúmenes de datos y tráfico.
5. **Consultas eficientes:** Permiten realizar consultas eficientes utilizando índices en los campos clave.

Ejemplos de bases de datos documentales:

1. **MongoDB:** Es una de las bases de datos documentales más populares y ampliamente utilizadas. Almacena datos en formato BSON (una representación binaria de JSON) y permite esquemas flexibles.
2. **Firebase Firestore:** Es una base de datos documental en la nube proporcionada por Firebase, que es parte de Google Cloud Platform. Almacena datos en formato JSON y es especialmente popular para aplicaciones web y móviles.

Tipos de BBDD NoSQL:

- Basados en columnas

- Permite más facilidad para hacer medias, varianza, etc.
- Clave-valor
 - como en la base de datos Redis o JSON
- Tripletes de
 - el objeto que describes
 - relaciones con otros objetos
 - valor
- Documentos
 - JSON
 - XML
 - BLOB
 - * texto

MongoDB

Mongo es una base de datos documental NoSQL (Not Only SQL) organizada en bases de datos como las bases de datos de MySQL o algo similar a los Schemas de Oracle. A su vez, las bases de datos se organizan en colecciones, lo que en relacional serían las tablas.

Mongo está muy relacionado con el mundo JavaScript y NodeJS porque la información que almacena y gestiona está en formato JSON.

1.1 Repaso JSON

Recordamos que JSON (JavaScript Object Notation) es un objeto JavaScript con el formato:

```
1 { clave: "valor" }
```

Ejemplo: Imagina que quieres almacenar una “persona”, pues un ejemplo de este objeto sería así:

```
1 {  
2   "Nombre": "John",  
3   "Apellidos": "Doe",  
4   "DNI": "12345678Z",  
5   "FechaNac": "01-01-1980",  
6   "Sexo": "V"  
7 }
```

1.2 JSON Schema

Si podemos crear una base de datos con documentos en cualquier formato y cualquier esquema, la programación se nos puede complicar.

Por suerte disponemos de herramientas como JSON Schema que nos permite “moldear” los documentos. Para el ejemplo anterior la sintaxis de un JSON Schema define los diferentes campos que una persona tendría sería así:

```
1 {
2   "$schema": "http://json-schema.org/draft-04/schema#",
3   "description": "A representation of a person, company, organization
4     , or place",
5   "type": "object",
6   "required": ["familyName", "givenName"],
7   "properties": {
8     "fn": {
9       "description": "Formatted Name",
10      "type": "string"
11    },
12    "familyName": { "type": "string" },
13    "givenName": { "type": "string" },
14    "additionalName": { "type": "array", "items": { "type": "string" } },
15    "honorificPrefix": { "type": "array", "items": { "type": "string" } },
16    "honorificSuffix": { "type": "array", "items": { "type": "string" } },
17    "nickname": { "type": "string" },
18    "url": { "type": "string", "format": "uri" },
19    "email": {
20      "type": "object",
21      "properties": {
22        "type": { "type": "string" },
23        "value": { "type": "string", "format": "email" }
24      }
25    },
26    "tel": {
27      "type": "object",
28      "properties": {
29        "type": { "type": "string" },
30        "value": { "type": "string", "format": "phone" }
31      }
32    },
33    "adr": { "$ref": "http://json-schema.org/address" },
34    "geo": { "$ref": "http://json-schema.org/geo" },
35    "tz": { "type": "string" },
36    "photo": { "type": "string" },
37    "logo": { "type": "string" },
38    "sound": { "type": "string" },
```

```
38     "bday": { "type": "string", "format": "date" },
39     "title": { "type": "string" },
40     "role": { "type": "string" },
41     "org": {
42         "type": "object",
43         "properties": {
44             "organizationName": { "type": "string" },
45             "organizationUnit": { "type": "string" }
46         }
47     }
48 }
49 }
```

Así, cuando en MongoDB veas referencia a un [schema](#) ya sabes de donde viene el tema. Cuando insertamos objetos, para optimizar la búsqueda e inserción de datos, Mongo va creando y actualizando los esquemas.

2 Instalación de MongoDB

De la imagen oficial de Mongo, adaptamos el Docker Compose:

```
1 services:
2
3   mongo:
4     image: mongo
5     restart: "no"
6     ports:
7       - 27017:27017
8     environment:
9       MONGO_INITDB_ROOT_USERNAME: root
10      MONGO_INITDB_ROOT_PASSWORD: 78agsbjha7834aSDFjhd73
11
12   mongo-express:
13     image: mongo-express
14     restart: "no"
15     ports:
16       - 8081:8081
17     environment:
18       ME_CONFIG_MONGODB_ADMINUSERNAME: root
19       ME_CONFIG_MONGODB_ADMINPASSWORD: 78agsbjha7834aSDFjhd73
20       ME_CONFIG_MONGODB_URL: mongodb://root:78agsbjha7834aSDFjhd73@mongo:27017/
21       ME_CONFIG_BASICAUTH: "false"
```

Para comprobar que ha funcionado, abrimos Mongo Express en local: <http://localhost:8081/>.

Dejamos propuesto como ejercicio crear en la carpeta del stack un fichero **.env** donde almacenar las credenciales y usar este archivo tanto para el contenedor como para los scripts de Python que haremos después. Aunque nosotros para que funcione el proyecto lo estamos incluyendo en el proyecto, **recuerda que es muy mala idea incluir cualquier tipo de credenciales en un repositorio de software**.

Mongo organiza los documentos en bases de datos, las bases de datos en colecciones y las colecciones tienen documentos. Para entender mejor cómo funciona te presentamos esta tabla de equivalencias:

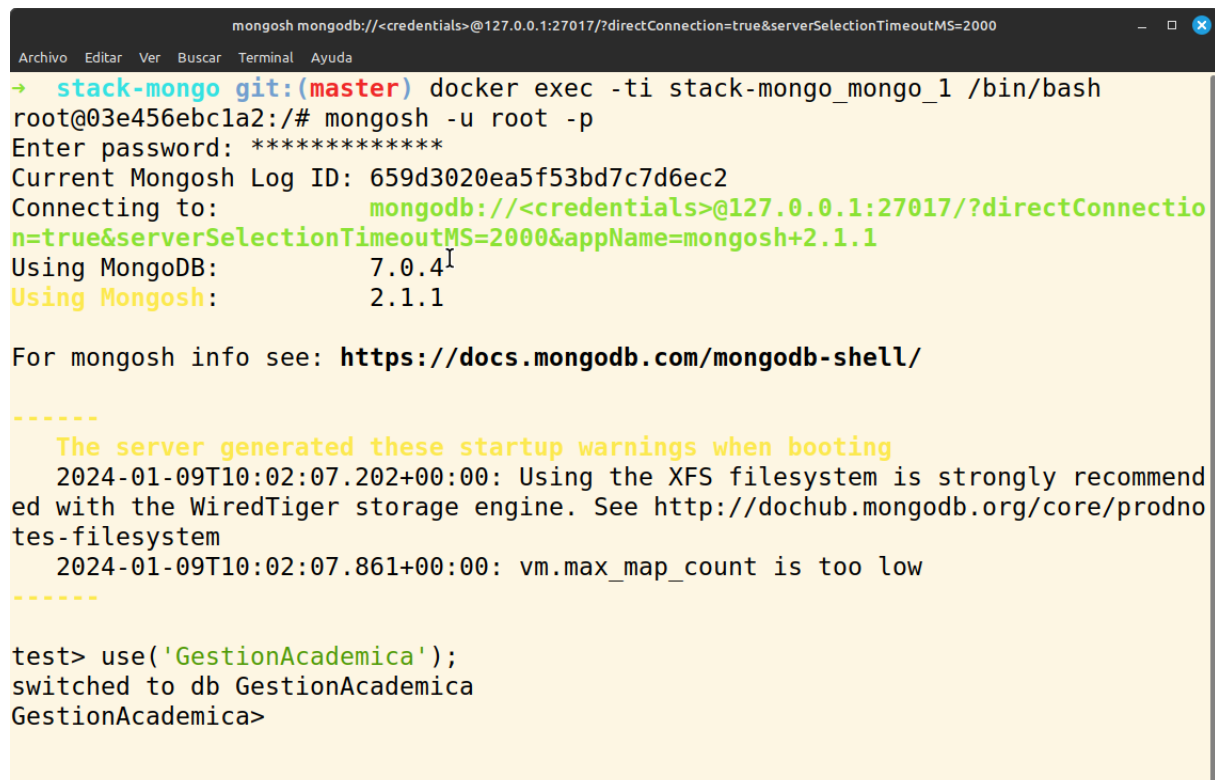
MySQL	MongoDB
Base de datos	Base de Datos
Tablas	Colecciones
Filas o tuplas	Documentos

Sin colecciones no puedo tener bases de datos en Mongo. Siempre, como mínimo estará la colección

“delete_me”.

2.1 Conexión interactiva a MongoDB

Aunque por lo general no usaremos MongoDB en modo interactivo, vamos a ver algunos ejemplos de cómo interactuar con la shell de mongo (mongosh):



```
mongosh mongodb://<credentials>@127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
→ stack-mongo git:(master) docker exec -ti stack-mongo_mongo_1 /bin/bash
root@03e456ebc1a2:/# mongosh -u root -p
Enter password: *****
Current Mongosh Log ID: 659d3020ea5f53bd7c7d6ec2
Connecting to:      mongodb://<credentials>@127.0.0.1:27017/?directConnectio
n=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.1.1
Using MongoDB:      7.0.4
Using Mongosh:      2.1.1

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2024-01-09T10:02:07.202+00:00: Using the XFS filesystem is strongly recommend
ed with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodno
tes-filesystem
2024-01-09T10:02:07.861+00:00: vm.max_map_count is too low
-----

test> use('GestionAcademica');
switched to db GestionAcademica
GestionAcademica>
```

Figura 2: Ejemplo de conexión en modo interactivo.

1. **Iniciar el shell interactivo:** Abre tu terminal y ejecuta los siguientes comandos:
 1. `docker exec -ti stack-mongo_mongo_1 /bin/bash`: para abrir una terminal interactiva en el contenedor de nuestro servicio **mongo**.
 2. `mongosh -u root -p`: para ingresar al shell interactivo de MongoDB, la contraseña es `83uddjfp0cmMD`, como fijamos en el docker-compose.
 3. `use('GestionAcademica')`: desde la shell de mongo, indicamos qué base de datos queremos usar de esta manera.
2. **Crear un documento (Create):** Para insertar un nuevo documento en una colección llamada `alumnos`, puedes utilizar el siguiente comando:

```
1 db.alumnos.insertOne({ nombre: "Juan Perez", edad: 20, carrera: "Ingeniería Informática" })
```

3. **Leer documentos (Read):** Para recuperar todos los documentos de la colección `alumnos`, puedes usar el comando `find()`:

```
1 db.alumnos.find()
```

Esto mostrará todos los documentos que representan a los alumnos.

4. **Actualizar un documento (Update):** Para actualizar un documento, puedes utilizar el comando `updateOne()`. Supongamos que Juan Pérez cambió su carrera a “Ingeniería Eléctrica”:

```
1 db.alumnos.updateOne({ nombre: "Juan Perez" }, { $set: { carrera: "Ingeniería Eléctrica" } })
```

Esto actualiza el documento de Juan Pérez con la nueva información sobre su carrera.

5. **Eliminar un documento (Delete):** Para eliminar un documento, puedes utilizar el comando `deleteOne()`. Supongamos que Juan Pérez ya no es alumno:

```
1 db.alumnos.deleteOne({ nombre: "Juan Perez" })
```

Esto eliminará el documento que representa a Juan Pérez de la colección.

Recuerda, para buscar un objeto en una colección usamos el método `find`. Así el formato para consultar la colección `profesor` sería:

```
1 db.profesor.find( { "nombre":"Juan" }).pretty();
```

Pero ¿y si quiero buscar profesore con asignaturas con más de 5 horas (inclusive)?

```
1 db.profesor.find({
2   asignaturas:
3     {
4       $elemMatch: {
5         horas: { $gt: 4}
6       }
7     }
8 });
```

Tienes más ejemplos sobre la sintaxis de consulta en la Web de MongoDB: <https://www.mongodb.com/docs/manual/tutorial/query-documents/>.

2.2 Colección de ejemplo

Vamos a utilizar una colección ficticia llamada `estudiantes` que contiene información sobre estudiantes en una universidad. Aquí tienes algunos documentos de ejemplo en esta colección:

```
1 db.estudiantes.insertMany([
2   { nombre: "Ana García", edad: 22, carrera: "Biología", semestre: 5,
3     promedio: 8.5 },
4   { nombre: "Carlos López", edad: 21, carrera: "Ingeniería Civil",
5     semestre: 4, promedio: 7.2 },
6   { nombre: "María Torres", edad: 20, carrera: "Psicología", semestre:
7     3, promedio: 9.0 },
8   { nombre: "Juan Rodríguez", edad: 23, carrera: "Historia", semestre:
9     6, promedio: 7.8 },
10  { nombre: "Elena Pérez", edad: 19, carrera: "Matemáticas", semestre:
11    2, promedio: 9.5 }
12 ])
```

Ahora, puedes realizar diversas consultas utilizando el método `find()` de MongoDB. Aquí tienes algunos ejemplos:

1. Recuperar todos los estudiantes:

```
1 db.estudiantes.find({})
```

2. Filtrar estudiantes por carrera:

```
1 db.estudiantes.find({ carrera: "Biología" })
```

3. Estudiantes mayores de 21 años:

```
1 db.estudiantes.find({ edad: { $gt: 21 } })
```

4. Estudiantes con promedio mayor o igual a 8.0:

```
1 db.estudiantes.find({ promedio: { $gte: 8.0 } })
```

5. Estudiantes de Psicología en el tercer semestre:

```
1 db.estudiantes.find({ carrera: "Psicología", semestre: 3 })
```

6. Ordenar estudiantes por promedio en orden descendente:

```
1 db.estudiantes.find().sort({ promedio: -1 })
```

7. Limitar la cantidad de resultados a 3:

```
1 db.estudiantes.find().limit(3)
```

Estos son solo ejemplos básicos de consultas utilizando el método `find()` en MongoDB. La sintaxis puede variar según las necesidades específicas de tu aplicación, y MongoDB ofrece una amplia variedad de operadores y opciones para realizar consultas más avanzadas. Puedes consultar la documentación oficial de MongoDB para obtener más detalles sobre la sintaxis y los operadores de consulta: [MongoDB Query Documents](#).

3 Segundo ejemplo completo con MongoDB

Para abrir una terminal interactiva con el contenedor, lo podemos hacer desde el plugin correspondiente del IDE, desde las propias utilidades de Docker Desktop o bien desde terminal así:

```
1 docker exec -ti [nombre_contenedor_mongostack] /bin/bash
```

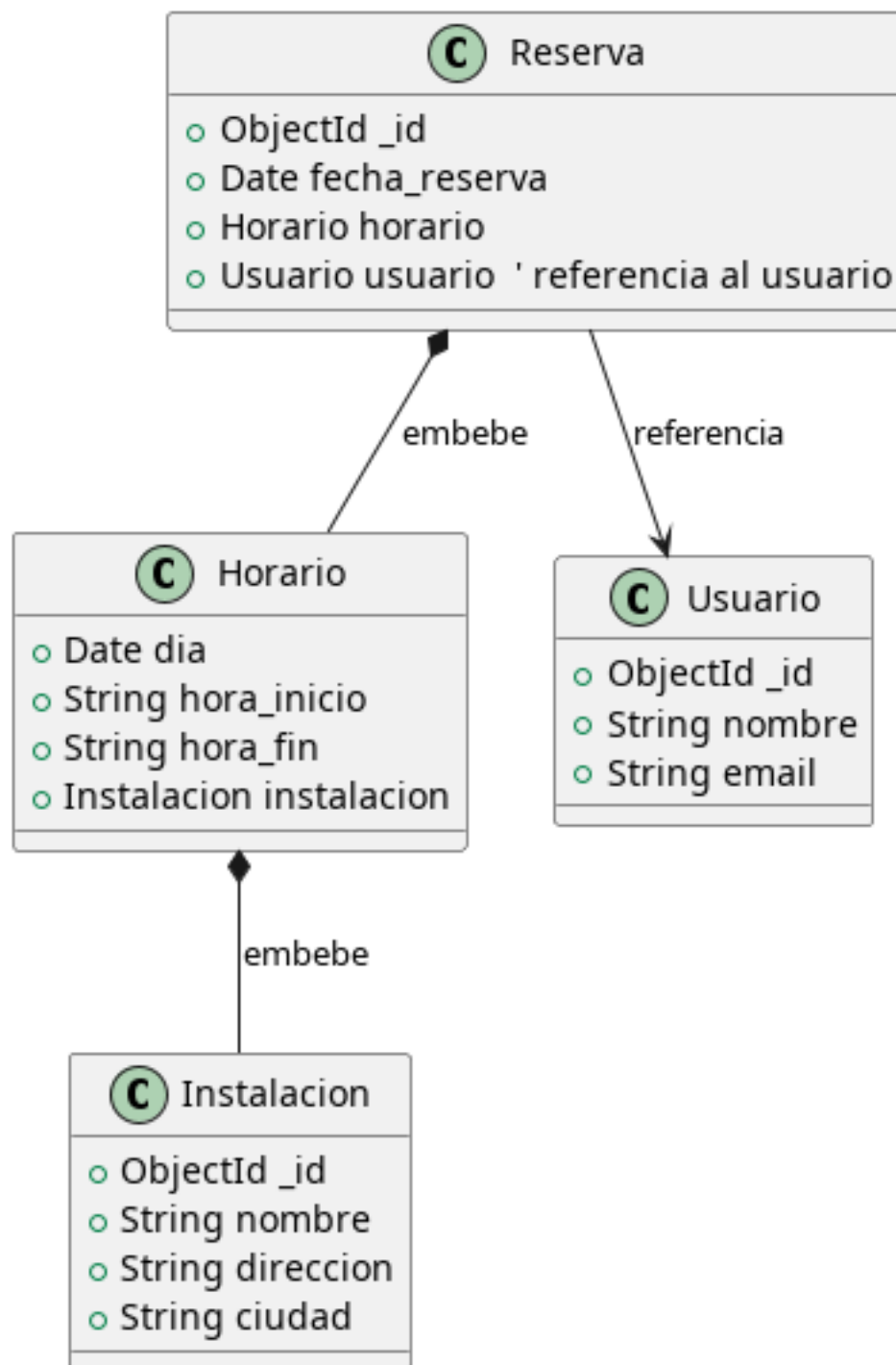
Ahora entramos en la shell de Mongo con:

```
1 mongosh -u root
```

Cuando nos pida la contraseña recuerda usar la que has puesto en el archivo `docker-compose.yml`.

A continuación vamos a revisar nuestro caso de reservas de pistas deportivas. Verás ejemplos de operaciones CRUD (Crear, Leer, Actualizar y Borrar) y algunas consultas útiles.

Nuestra base de datos sigue este esquema:

**Figura 3:** Diagrama PlantUML

3.1 1. Modelo de Datos del Ejemplo

Para nuestro ejemplo, tenemos las siguientes entidades:

- **Instalaciones:** Datos de las pistas o instalaciones deportivas.
- **Horarios:** Disponibilidad de cada instalación. Cada horario lleva *embebida* la información de la instalación (para tener un snapshot de la instalación cuando se crea el horario).
- **Usuarios:** Información de los usuarios que realizan reservas.
- **Reservas:** Cada reserva incluye:
 - La fecha de reserva.
 - Una referencia al usuario (no embebemos el usuario, sino que solo guardamos su `_id`).
 - El horario embebido, con la información del día, hora y la instalación.

3.2 2. Ejemplos de Operaciones CRUD

3.2.1 2.1. Seleccionar la Base de Datos

```
1 use reservas_db;
```

3.2.2 2.2. Crear (Insertar Documentos)

3.2.2.1 Insertar una Instalación

Insertar una instalación nueva:

```
1 db.instalaciones.insertOne({
2   nombre: "Pista de Tenis",
3   direccion: "Calle A 123",
4   ciudad: "Madrid"
5 });
```

3.2.2.2 Insertar un Horario con la Instalación Embebida

Supongamos que ya tenemos la instalación insertada y conocemos su `_id` (por ejemplo, `ObjectId("60f1b2c3d4e5f67890123456")`):

```
1 db.horarios.insertOne({
2   dia: "2025-03-10",
3   hora_inicio: "10:00",
4   hora_fin: "11:00",
5   instalacion: {
6     _id: ObjectId("60f1b2c3d4e5f67890123456"),
7     nombre: "Pista de Tenis",
8     direccion: "Calle A 123"
9   }
10 });
```



```
9    }  
10   });
```

3.2.2.3 Insertar un Usuario Insertar un Usuario:

```
1 db.usuarios.insertOne({  
2   nombre: "Ana García",  
3   email: "ana.garcia@example.com"  
4 });
```

3.2.2.4 Insertar una Reserva Imaginemos que:

- El usuario insertado tiene `_id: ObjectId("60f1b2c3d4e5f67890123457")`
- El horario (con la instalación embebida) se usó para crear la reserva.

```
1 db.reservas.insertOne({  
2   fecha_reserva: "2025-03-01",  
3   usuario_id: ObjectId("60f1b2c3d4e5f67890123457"),  
4   horario: {  
5     dia: "2025-03-10",  
6     hora_inicio: "10:00",  
7     hora_fin: "11:00",  
8     instalacion: {  
9       _id: ObjectId("60f1b2c3d4e5f67890123456"),  
10      nombre: "Pista de Tenis",  
11      direccion: "Calle A 123"  
12    }  
13  }  
14 });
```

3.2.3 2.3. Leer (Consultas)

3.2.3.1 Consultar Todas las Instalaciones Listar todas las Instalaciones

```
1 db.instalaciones.find().pretty();
```

3.2.3.2 Buscar Horarios para un Día Específico Buscar Horarios para un día específico:

```
1 db.horarios.find({ dia: "2025-03-10" }).pretty();
```

3.2.3.3 Buscar Reservas de un Usuario Específico Reservas de un Usuario específico (por ID):

```
1 db.reservas.find({ usuario_id: ObjectId("60f1b2c3d4e5f67890123457") }).pretty();
```

3.2.3.4 Buscar Reservas para una Instalación Reservas para una Instalación (usando **dot notation** en el horario):

```
1 db.reservas.find({ "horario.instalacion.nombre": "Pista de Tennis" }).pretty();
```

3.2.4 2.4. Actualizar

3.2.4.1 Actualizar la Dirección de una Instalación Actualizar la Dirección de una Instalación:

```
1 db.instalaciones.updateOne(  
2   { _id: ObjectId("60f1b2c3d4e5f67890123456") },  
3   { $set: { direccion: "Calle Nueva 456" } }  
4 );
```

Nota: Si la dirección de la instalación cambia y se desea que las modificaciones se reflejen en los horarios o reservas, habría que actualizar esos documentos por separado. Esto es una de las consideraciones al embedir datos.

3.2.4.2 Actualizar el Horario Embebido en una Reserva Por ejemplo, modificar la hora de inicio y fin de un horario dentro de una reserva:

```
1 db.reservas.updateOne(  
2   { _id: ObjectId("60f1b2c3d4e5f67890123458") },  
3   { $set: { "horario.hora_inicio": "11:00", "horario.hora_fin": "12:00" } }  
4 );
```

3.2.5 2.5. Borrar

3.2.5.1 Eliminar un Usuario Eliminar un Usuario por ID:

```
1 db.usuarios.deleteOne({ _id: ObjectId("60f1b2c3d4e5f67890123457") });
```

3.2.5.2 Eliminar una Reserva Eliminar una reserva por ID:

```
1 db.reservas.deleteOne({ _id: ObjectId("60f1b2c3d4e5f67890123458") });
```

3.3 Objetos Embebidos vs. Objetos Anidados

3.3.1 Objetos Embebidos

- **Definición:** Son documentos completos que se insertan directamente dentro de otro documento.
- **Ventajas:**
 - **Lectura Rápida y Sencilla:** Al estar en un mismo documento, se evitan costosas operaciones de “join”.
 - **Atomicidad:** Las operaciones de escritura en el documento completo son atómicas.
- **Desventajas:**
 - **Duplicación de Datos:** Si el objeto embebido es usado en varios documentos, se duplica la información.
 - **Actualizaciones Complejas:** Si la información embebida necesita actualizarse de forma global, se debe actualizar en cada documento donde aparezca.
- **Ejemplo en Nuestro Caso:**

En la colección [horarios](#) se embebe la información de la instalación. Esto es útil si el horario necesita conservar un snapshot de la instalación en el momento de creación, sin preocuparse por cambios futuros en la instalación.

4 Mi primera aplicación Flask

Nuestra referencia es el manual de Flask.

Si no tenemos python instalado, lo instalamos:

```
1 apt install python3.12 python3.12-venv
```

En Windows sería similar pero con `winget`.

Empezamos la instalación.

```
1 mkdir tuto-flask
2 cd tuto-flask
3 python -m venv venv
4 . ./venv/bin/activate
```

Creamos un archivo `.gitignore`, le añadimos la carpeta `venv`.

Inicializamos el repositorio:

```
1 git init
```

Añadimos Flask al proyecto:

```
1 pip install Flask
```

Cada vez que añadimos una dependencia hay que ejecutar esto:

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

Creamos un archivo `hola.py` con este contenido:

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route("/")
6 def hello_world():
7     return "<p>Hola Mundo!</p>"
```

Me aseguro que el virtual environment está activo y ejecuto la APP en el puerto 8080:

```
1 . ./venv/bin/activate
2 flask --app hola run --port 8080
```

5 Autenticación JWT con Praetorian y MongoEngine en Flask

Vamos a hacer una API REST con algunas rutas protegidas y a las que vamos a poder acceder con autenticación Bearer (token JWT).

MongoEngine es un **Document-Relational Mapper (DRM)** para Python que facilita la interacción con bases de datos NoSQL basadas en MongoDB. Su funcionamiento es similar al de los **Object-Relational Mappers (ORMs)** en bases de datos relacionales, como SQLAlchemy en Python o Hibernate en Java. De hecho, si se busca una analogía en el ecosistema de Java, MongoEngine cumple un papel equivalente al de **Spring Data JPA**, pero aplicado a una base de datos documental en lugar de relacional.

Al igual que **Spring Data JPA** permite definir entidades Java como clases con anotaciones que representan tablas y relaciones, **MongoEngine** usa clases de Python para modelar documentos y sus estructuras dentro de MongoDB. Por ejemplo, en Spring Data JPA, se usaría `@Entity` para definir una clase persistente en una base de datos relacional, mientras que en MongoEngine se define un modelo heredando de `Document` y se especifican los campos con tipos como `StringField`, `IntField`, o `ReferenceField`.

Otra similitud clave es la manera en que ambos frameworks abstraen la interacción con la base de datos. Mientras que Spring Data JPA proporciona repositorios (`JpaRepository`) para facilitar operaciones CRUD sin necesidad de escribir consultas SQL manualmente, MongoEngine ofrece métodos como `.save()`, `.objects()` y `.delete()`, que permiten interactuar con la base de datos sin necesidad de escribir queries en BSON o JSON. Esta capa de abstracción simplifica enormemente el desarrollo, permitiendo a los programadores centrarse en la lógica de negocio en lugar de en los detalles de almacenamiento y recuperación de datos.

Flask-Praetorian es un **framework de autenticación basado en JWT** para Flask que simplifica la gestión de usuarios, roles y protección de rutas en aplicaciones web. Si buscamos una analogía en el ecosistema de Java, Flask-Praetorian cumple un papel similar a **Spring Security**, que es la solución estándar para gestionar autenticación y autorización en aplicaciones Spring Boot.

Al igual que **Spring Security** permite definir un sistema de autenticación y autorización basado en tokens o sesiones, **Flask-Praetorian** proporciona una integración sencilla para manejar autenticación con **JWT (JSON Web Tokens)**. En Spring Security, un usuario se representa generalmente con una entidad que implementa `UserDetails`, mientras que en Flask-Praetorian se define una clase de usuario que cumple con ciertos métodos requeridos, como `lookup()`, `identify()`, e `is_valid()`.

En cuanto a la protección de rutas, Spring Security usa anotaciones como `@PreAuthorize("hasRole('ADMIN')")` para restringir el acceso según los roles del usuario. En Flask-Praetorian, esto se logra con decoradores como `@guard.auth_required` o `@guard.roles_required('admin')`, que permiten restringir endpoints a usuarios autenticados o con ciertos permisos.

Ambos frameworks también facilitan el almacenamiento seguro de contraseñas: Spring Security usa **BCrypt** para cifrar contraseñas por defecto, mientras que Flask-Praetorian también admite BCrypt y permite su integración de manera sencilla.

5.1 Convirtiendo de MySQL a JSON

Imagina que queremos exportar las tuplas de la tabla usuario a documentos JSON. Para ello nos conectamos al contenedor de MYSQL con algo parecido a esto (donde el parámetro `ti` quiere decir terminal interactivo):

```
1 docker exec -ti pilapistas_db_1 sh
2 mysql -u root -p
```

La contraseña es la que fijamos en el docker-compose (recuerda el tema anterior). Ahora con este comando podemos exportar los datos a JSON para **usuario**:

```
1 use deporte;
2 show tables;
3 describe usuario;
4 select JSON_OBJECT(
5     'enabled', enabled,
6     'id', id,
7     'username', username,
8     'email', email,
9     'password', password,
10    'tipo', tipo
11 ) from usuario;
```

Lo que nos daría como salida:

```
1 {"id": 2, "tipo": "OPERARIO", "email": "pepe@gmail.com", "enabled": "
  base64:type16:AQ==", "password": "$2a$10$zLD33q.JAxrRPsUGYGY7tedH/
  dQUn2MmlxQzj07Y.oqK6r0jJdueq", "username": "pepe"}
2 {"id": 5, "tipo": "ADMIN", "email": "admin@correo.com", "enabled": "
  base64:type16:AQ==", "password": "$2a$10$krLxeZI8Xm.n1fNz7v81Y.
  yzsHtoMoCnDCsStEAPeGkE9BU0Bkwn2", "username": "admin"}
3 {"id": 7, "tipo": "USUARIO", "email": "darksideside@starwars.com", "enabled": "
  base64:type16:AQ==", "password": "$2a$10$.EJQbCFZtHW1pavBGmMkw.
  VxOn2or6AL2oPP.8RVvCSqXQA/zwUom", "username": "obijuan"}
4 {"id": 13, "tipo": "ADMIN", "email": "gerencia@vdc.com", "enabled": "
  base64:type16:AQ==", "password": "$2a$10$hWkDEd0V0QgmiffgPcScoe1.
  OMq5ew.wl70FBMqii5XkfxtIwzZ92", "username": "gerente"}
```

Ejercicio: Haz lo mismo para instalacion, horario y reserva.

5.2 Instalar dependencias necesarias

Primero, asegúrate que tienes las dependencias o librerías necesarias:

```
1 pip install flask mongoengine flask-praetorian flask-cors flask-bcrypt
```

Una vez más recuerda salvar esta configuración para poder clonar el repositorio y empezar de cero.

Para guardar las dependencias hacemos:

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

Para instalar y recuperar el entorno virtual (en Linux/Mac):

```
1 python -m venv .
2 . ./venv/bin/activate
3 pip -r requirements.txt
```

5.3 Configurar Flask con MongoEngine

Creamos o modificamos el archivo `app.py` con la configuración de MongoDB y Flask.

```
1 from flask import Flask, jsonify, request
2 from flask_cors import CORS
3 from flask_praetorian import Praetorian
4 from flask_bcrypt import Bcrypt
5 from mongoengine import connect, Document, StringField, BooleanField
6
7 app = Flask(__name__)
8 CORS(app)
9 bcrypt = Bcrypt(app)
10
11 # Configuración de MongoDB
12 app.config["MONGODB_SETTINGS"] = {
13     "db": "gestion",
14     "host": "mongodb://root:78agsbjha7834aSDFjhd73@mongo:27017"
15 }
16 connect(**app.config["MONGODB_SETTINGS"])
17
18 # Configuración de Flask-Praetorian
19 app.config["SECRET_KEY"] = "supersecretkey"
20 app.config["JWT_ACCESS_LIFESPAN"] = {"hours": 24}
21
22 # Inicializar Praetorian
23 guard = Praetorian()
```

5.4 Definir el modelo de usuario

MongoEngine no usa SQLAlchemy, así que definimos un **modelo de usuario** con las funciones necesarias para Flask-Praetorian.

```
1 class User(Document):
2     username = StringField(required=True, unique=True)
3     password = StringField(required=True)
4     roles = StringField(default="user") # Praetorian usa esto para
        roles
5     is_active = BooleanField(default=True)
6
7     @classmethod
8     def lookup(cls, username):
9         return cls.objects(username=username).first()
10
11     @classmethod
12     def identify(cls, user_id):
13         return cls.objects(id=user_id).first()
14
15     def is_valid(self):
16         return self.is_active
```

Nota: `lookup()` y `identify()` son métodos que necesita Flask-Praetorian para buscar usuarios.

5.5 Inicializar Praetorian con MongoEngine

Agregamos esta línea en `app.py` después de definir el modelo:

```
1 guard.init_app(app, User)
```

5.6 Crear rutas de autenticación

Ahora, agregamos **registro de usuarios, login y ruta protegida**.

```
1 @app.route("/register", methods=["POST"])
2 def register():
3     data = request.get_json()
4     username = data.get("username")
5     password = data.get("password")
6
7     if User.objects(username=username).first():
8         return jsonify({"error": "Usuario ya existe"}), 400
9
10    hashed_password = guard.hash_password(password)
11    user = User(username=username, password=hashed_password).save()
```



```
12
13     return jsonify({"message": "Usuario registrado"}), 201
14
15
16 @app.route("/login", methods=["POST"])
17 def login():
18     data = request.get_json()
19     username = data.get("username")
20     password = data.get("password")
21
22     user = User.objects(username=username).first()
23
24     if user and guard.authenticate(username, password):
25         token = guard.encode_jwt_token(user)
26         return jsonify({"access_token": token}), 200
27     else:
28         return jsonify({"error": "Credenciales incorrectas"}), 401
29
30
31 @app.route("/protected", methods=["GET"])
32 @guard.auth_required
33 def protected():
34     user = guard.current_user()
35     return jsonify({"message": f"Bienvenido, {user.username}."}), 200
```

Para probar los endpoints:

Registro de usuario

```
1 curl -X POST http://127.0.0.1:5000/register -H "Content-Type: application/json" -d '{"username": "admin", "password": "1234"}'
```

Login y obtención del token

```
1 curl -X POST http://127.0.0.1:5000/login -H "Content-Type: application/json" -d '{"username": "admin", "password": "1234"}'
```

Acceso a una ruta protegida (reemplaza <TOKEN> por el token obtenido)

```
1 curl -X GET http://127.0.0.1:5000/protected -H "Authorization: Bearer <TOKEN>"
```