MongoDB & Python ORM Patterns

Learning Objectives

- Set up MongoDB using Docker Compose
- Understand MongoDB's document-based data model
- Connect to MongoDB using PyMongo for raw operations
- Use MongoEngine ODM for object-document mapping
- Compare NoSQL document patterns with relational ORM
- Build practical MongoDB integration functions

Quick Review: Where We Are

Week 1: Development environment, Git, Python basics
Week 2: Type hints, code quality tools (mypy, ruff, pre-commit)
Week 3: Data structures (TypedDict, dataclasses, Pydantic, NumPy)
Week 4: Pandas mastery and data validation
Week 5: Databases, Docker, and SQLAlchemy ORM
Week 6 (Today): MongoDB and Python document patterns

Today's Focus: From relational to document-based data storage

Part 1: MongoDB with Docker

Adding MongoDB to Docker Compose

Extending our existing docker-compose.yml:

Updated docker-compose.yml:

```yaml
version: '3.8'

services:
  mariadb: ...

  mongodb:
    image: mongo:7.0
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: adminpassword
      MONGO_INITDB_DATABASE: dataeng
    ports:
      - "27017:27017"
    volumes:
      - ./data/mongo:/data/db
      - ./init-mongo.js:/docker-entrypoint-initdb.d/init-mongo.js
```

Starting the services:

```bash
# Start both databases
docker-compose up -d

# Check running services
docker-compose ps

# View MongoDB logs
docker-compose logs mongodb

# Connect to MongoDB shell
docker-compose exec mongodb mongosh -u admin -p adminpassword
```

Key differences from MariaDB:

- ✅ **No schema required** - documents can vary
- ✅ **JSON-like storage** - natural for Python dicts
- ✅ **Flexible structure** - add fields dynamically
- ❌ **No relationships** - no foreign keys or joins

MongoDB Initialization Script

init-mongo.js - Creating collections and sample data:

```
// Switch to dataeng database
db = db.getSiblingDB('dataeng');

// Create a user collection with sample data
db.users.insertMany([
  {
    username: "alice",
    email: "alice@example.com",
    profile: {
      age: 28,
      city: "Berlin",
      interests: ["python", "data science"]
    },
    created_at: new Date()
  },
  ...
]);

// Create an index on username
db.users.createIndex({ "username": 1 }, { unique: true });

console.log("MongoDB initialized with sample data");
```

Notice the document structure:

- **Nested objects** (`profile` contains multiple fields)
- **Arrays** (`interests` as list of strings)
- **Flexible schema** (documents can have different fields)
- **No foreign keys** (all data embedded or referenced by ID)

MongoDB shell verification:

```
// List all users
db.users.find().pretty()

// Find user by username
db.users.findOne({username: "alice"})

// Update user's interests
db.users.updateOne(
  {username: "alice"},
  {$push: {interests: "machine learning"}}
)
```

Part 2: Raw MongoDB Operations

Basic MongoDB Queries

Understanding MongoDB's query language:

List all users:

```
db.users.find()
```

Filter by username:

```
db.users.find({username: "alice"})
```

Filter by nested field:

```
db.users.find({"profile.city": "Berlin"})
```

Filter by array element:

```
db.users.find({interests: "python"})
```

Insert new user:

```
db.users.insertOne({
  username: "charlie",
  email: "charlie@example.com",
  profile: {
    age: 25,
    city: "Hamburg"
  },
  created_at: new Date()
})
```

Update user:

```
db.users.updateOne(
  {username: "charlie"},
  {$set: {"profile.age": 26}}
)
```

MongoDB Operators Reference

Essential MongoDB operators for data manipulation:

Operator	Purpose	Example
`$set`	Set field value	`{$set: {"age": 30}}`
`$push`	Add to array	`{$push: {"interests": "python"}}`
`$pull`	Remove from array	`{$pull: {"interests": "java"}}`
`$inc`	Increment number	`{$inc: {"age": 1}}`
`$unset`	Remove field	`{$unset: {"temp_field": ""}}`
`$addToSet`	Add unique to array	`{$addToSet: {"tags": "new"}}`
`$gt` / `$lt`	Greater/Less than	`{"age": {$gt: 18}}`
`$in` / `$nin`	In/Not in array	`{"city": {$in: ["Berlin", "Munich"]}}`
`$exists`	Field exists	`{"profile": {$exists: true}}`
`$regex`	Pattern match	`{"email": {$regex: "@gmail"}}`

The Absence of Relationships

Why MongoDB doesn't have foreign keys:

Relational approach (SQLAlchemy):

```sql
-- Users table
CREATE TABLE users (
  id INT PRIMARY KEY,
  username VARCHAR(50),
  email VARCHAR(100)
);

-- Orders table
CREATE TABLE orders (
  id INT PRIMARY KEY,
  user_id INT REFERENCES users(id),
  total DECIMAL(10,2)
);

-- Join required for user + orders
SELECT u.username, o.total
FROM users u
JOIN orders o ON u.id = o.user_id;
```

Document approach (MongoDB):

```javascript
// Option 1: Embed orders in user document
{
  username: "alice",
  email: "alice@example.com",
  orders: [
    {order_id: 1, total: 99.99, date: "2024-01-15"},
    {order_id: 2, total: 149.50, date: "2024-02-01"}
  ]
}

// Option 2: Reference by ID (manual joins)
// User document
{_id: ObjectId("..."), username: "alice"}

// Order documents
{user_id: ObjectId("..."), total: 99.99}
{user_id: ObjectId("..."), total: 149.50}
```

Trade-off: Flexibility vs. data consistency

Part 3: PyMongo - Python MongoDB Driver

Connecting with PyMongo

Basic connection setup:

```
from pymongo import MongoClient
from pymongo.errors import ConnectionFailure
import os

MONGO_URL = 'mongodb://admin:adminpassword@localhost:27017/'

def get_mongodb_client():
    client = MongoClient(MONGO_URL)
    client.admin.command('ping')
    return client

client = get_mongodb_client()
db = client.dataeng
users_collection = db.users
```

Basic operations:

```
user_data = {
    "username": "dave",
    "email": "dave@example.com",
    "profile": {
        "age": 30,
        "city": "Frankfurt"
    }
}
result = users_collection.insert_one(user_data)
print(f"Inserted ID: {result.inserted_id}")

for user in users_collection.find({"profile.city": "Berlin"}):
    print(f"User: {user['username']}")

user = users_collection.find_one({"username": "alice"})
print(f"Found: {user['email']}")

users_collection.update_one(
    {"username": "dave"},
    {"$set": {"profile.age": 31}}
)
```

Part 4: MongoEngine ODM

MongoEngine: Object-Document Mapping

Why use MongoEngine over raw PyMongo?

Raw PyMongo (verbose):

```
user_doc = {
    "username": username,
    "email": email,
    "profile": {"age": age, "city": city},
    "created_at": datetime.utcnow()
}
result = collection.insert_one(user_doc)

if not user_doc.get('email'):
    raise ValueError("Email required")

user_doc['profile']['age'] = int(age)
```

MongoEngine (clean):

```
from mongoengine import (
    Document, StringField, IntField,
    EmbeddedDocument, EmbeddedDocumentField, DateTimeField,
)
from datetime import datetime

class Profile(EmbeddedDocument):
    age = IntField(min_value=0, max_value=120)
    city = StringField(max_length=100)

class User(Document):
    username = StringField(required=True, unique=True, max_length=50)
    email = StringField(required=True)
    profile = EmbeddedDocumentField(Profile)
    created_at = DateTimeField(default=datetime.utcnow)

user = User(username="eve", email="eve@example.com")
user.profile = Profile(age=27, city="Cologne")
user.save()
```

MongoEngine Document Models

Complete User model:

```python
from mongoengine import *
import datetime

connect('dataeng', host=MONGO_URL)

class Profile(EmbeddedDocument):
    age = IntField(min_value=0, max_value=120)
    city = StringField(max_length=100)
    interests = ListField(StringField(max_length=50))

class User(Document):
    username = StringField(required=True, unique=True, max_length=50)
    email = StringField(required=True)
    profile = EmbeddedDocumentField(Profile)
    created_at = DateTimeField(default=datetime.datetime.utcnow)

    meta = {
        'collection': 'users',
        'indexes': ['username', 'email']
    }

    def __str__(self):
        return f"User({self.username}, {self.email})"
```

Using the model:

```python
user = User(
    username="frank",
    email="frank@example.com"
)
user.profile = Profile(
    age=35,
    city="Stuttgart",
    interests=["python", "mongodb", "data engineering"]
)
user.save()

berlin_users = User.objects(profile__city="Berlin")
for user in berlin_users:
    print(f"{user.username} from {user.profile.city}")

user = User.objects(username="frank").first()
user.profile.interests.append("docker")
user.save()

User.objects(username="frank").delete()
```

Why It's Not Exactly ORM

Object-Document Mapping vs Object-Relational Mapping:

Traditional ORM (SQLAlchemy):

- **Relational model**: Tables, rows, columns
- **Relationships**: Foreign keys, joins
- **Schema enforcement**: Fixed table structure
- **ACID transactions**: Strong consistency
- **SQL queries**: Structured query language

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    username = Column(String(50))
    orders = relationship("Order", back_populates="user")

users_with_orders = session.query(User).join(Order).all()
```

Document ODM (MongoEngine):

- **Document model**: Collections, documents, fields
- **Embedding**: Nested documents, no joins needed
- **Schema flexibility**: Documents can vary
- **Eventual consistency**: Flexible consistency models
- **MongoDB queries**: Document-based queries

```
class User(Document):
    username = StringField()
    orders = ListField(EmbeddedDocumentField(Order))

users_with_orders = User.objects(orders__exists=True)
```

Key difference: ODM maps objects to documents, not relational tables.

Part 5: Practical MongoDB Functions

PyMongo Functions

PyMongo implementation:

```
from pymongo import MongoClient
from bson import ObjectId
from datetime import datetime

client = MongoClient(MONGO_URL)
db = client.dataeng
users = db.users

def list_all_users() -> list[dict]:
    user_list = []
    for user in users.find():
        user['id'] = str(user['_id'])
        del user['_id']
        user_list.append(user)
    return user_list

def find_by_username(username: str) -> dict | None:
    user = users.find_one({"username": username})
    if user:
        user['id'] = str(user['_id'])
        del user['_id']
    return user
```

```
def create_user(username: str, email: str,
                age: int = None, city: str = None) -> str:
    user_doc = {
        "username": username,
        "email": email,
        "created_at": datetime.utcnow()
    }

    if age or city:
        user_doc["profile"] = {}
        if age:
            user_doc["profile"]["age"] = age
        if city:
            user_doc["profile"]["city"] = city

    result = users.insert_one(user_doc)
    return str(result.inserted_id)

def update_user(username: str, **updates) -> bool:
    update_doc = {}
    for key, value in updates.items():
        if key in ['age', 'city']:
            update_doc[f"profile.{key}"] = value
        else:
            update_doc[key] = value

    result = users.update_one(
        {"username": username},
        {"$set": update_doc}
    )
    return result.modified_count > 0
```

MongoEngine Functions

MongoEngine implementation:

```
from mongoengine import *

connect('dataeng', host=MONGO_URL)

def list_all_users() -> list[User]:
    return list(User.objects.all())

def find_by_username(username: str) -> User | None:
    try:
        return User.objects.get(username=username)
    except User.DoesNotExist:
        return None

def create_user(username: str, email: str,
                age: int = None, city: str = None) -> User:
    user = User(username=username, email=email)

    if age or city:
        profile = Profile()
        if age:
            profile.age = age
        if city:
            profile.city = city
        user.profile = profile

    user.save()
    return user
```

```
def update_user(username: str, **updates) -> bool:
    try:
        user = User.objects.get(username=username)

        profile_updates = {}
        user_updates = {}

        for key, value in updates.items():
            if key in ['age', 'city']:
                profile_updates[key] = value
            else:
                user_updates[key] = value

        if profile_updates:
            if not user.profile:
                user.profile = Profile()
            user.profile.update(**profile_updates)

        if user_updates:
            user.update(**user_updates)

        user.save()
        return True
    except User.DoesNotExist:
        return False

users_raw = list_all_users()
users_odm = list_all_users()
```

What We've Accomplished Today

MongoDB Setup:

- ✅ Extended Docker Compose with MongoDB container
- ✅ MongoDB initialization with JavaScript
- ✅ Understanding document-based data model
- ✅ Recognizing the absence of relationships

Python Integration:

- ✅ PyMongo for raw MongoDB operations
- ✅ MongoEngine ODM for object-document mapping
- ✅ Understanding ODM vs ORM differences
- ✅ Complete CRUD function implementations

Key Insights:

- ✅ When to use documents vs relational data
- ✅ Embedding vs referencing strategies
- ✅ Trade-offs between flexibility and consistency

Coming Up in Practical Session

Hands-on Implementation:

- Set up MongoDB with Docker Compose
- Create MongoDB collections and documents
- Implement user management with PyMongo
- Build the same functionality with MongoEngine
- Compare performance and code complexity
- Design document schemas for real-world data

Next Week Preview:

- Neo4j graph database integration
- Python graph operations and queries
- Relationship modeling with graphs
- Combining relational, document, and graph data

By the end: You'll understand when to choose document databases and how to integrate them effectively with Python!

Questions?

Ready to embrace the document database world!

During the practical session, we'll build a complete MongoDB integration system and see how document databases complement relational databases in modern data engineering projects.