Topic 10: MongoDB with Flask Applications

Overview

Building on **Topic 09's interactive exploration** of MongoDB concepts, this topic develops a **complete Flask web application** using Mongita. While Topic 09 introduced basic CRUD operations in the Python REPL, Topic 10 applies those concepts to build a structured application with:

- Web interface for managing pets and kinds
- Manual joins between collections (like SQL relationships)
- Application-level referential integrity
- Complete CRUD operations through HTTP routes

This demonstrates how MongoDB fits into web application architecture, paralleling Topics 02-06 but with document-oriented database design.

From Interactive to Application

Topic 09 (Interactive Session):

```python
# Direct commands in Python REPL
mongoose_collection.insert_one({'name': 'Meercat', 'weight': 2})
list(mongoose_collection.find({'weight': {'$gt': 1}}))
```

Topic 10 (Flask Application):

```python
# Structured functions with web interface
@app.route("/create", methods=["POST"])
def post_create():
    data = dict(request.form)
    database.create_pet(data)
    return redirect(url_for("get_list"))
```

Why Continue with Mongita?

As established in Topic 09, Mongita provides the ideal environment for learning MongoDB:

- **No server required**: Runs entirely in Python, no installation or server management needed
- **MongoDB-compatible API**: Code written for Mongita translates directly to MongoDB
- **Disk-based persistence**: Data survives between sessions, stored in local files
- **Lightweight**: Perfect for development and learning environments
- **Pure Python**: Easy to install via pip with no system dependencies

Code written in this topic will work with real MongoDB (Topics 10-11) with minimal changes.

Key MongoDB Concepts Applied

1. Document-Oriented Data Model

Unlike relational databases with fixed schemas and tables, MongoDB stores data as **documents** - flexible, JSON-like objects that can contain nested data structures.

Example from our code:

1. Document-Oriented Data Model (from Topic 09)

As explored in Topic 09, MongoDB stores data as **documents** - flexible, JSON-like objects that can contain nested data structures.

Example from our code:

```
pet_document = {
    "name": "Suzy",
    "age": 3,
    "owner": "Greg",
    "kind_id": ObjectId("...")  # Reference to another document
}
```

Recall from Topic 09:

- Documents in the same collection can have different fields
- No schema enforcement - flexibility over structure
- Self-contained records with nested data

2. Collections Instead of Tables (from Topic 09)

In MongoDB, **collections** are analogous to tables in relational databases, but without enforced schemas. Our application uses two collections:

- `pets_collection` - Stores individual pet records
- `kind_collection` - Stores pet kind/species information (Dog, Cat, Fish)

Creating and accessing collections (Topic 09 pattern):

```
pets_db = client.pets_db  # Database (auto-created)
pets_collection = pets_db.pets_collection  # Collection (auto-created)
kind_collection = pets_db.kind_collection  # Another collection
```

No CREATE TABLE needed - collections created on first use, as demonstrated in Topic 09.

3. ObjectId and Document Identity (from Topic 09)

As introduced in Topic 09, MongoDB automatically generates a unique `_id` field for each document using **ObjectId**.

Key principles (Topic 09 recap):

- Every document must have an `_id` field
- If not provided, MongoDB generates an ObjectId automatically
- ObjectIds are used for references between documents (like foreign keys in SQL)
- ObjectIds can be converted to strings for display and back for queries

Usage in our code:

```
from bson.objectid import ObjectId

# Converting string ID to ObjectId for queries
id = ObjectId(id_string)
pet = pets_collection.find_one({"_id": id})

# Converting ObjectId to string for display
pet["id"] = str(pet["_id"])
```

4. Document References (Manual Joins) - NEW IN TOPIC 10

MongoDB doesn't have built-in JOIN operations like SQL databases. Instead, we use **manual references** - storing the `_id` of one document in another document.

This extends Topic 09's single-collection operations to **multi-collection relationships**.

Our reference pattern:

```
# Each pet stores a reference to its kind
pet = {
    "name": "Suzy",
    "kind_id": ObjectId("...")  # References a document in kind_collection
}

# To display complete information, we manually "join" the data
kind = kind_collection.find_one({"_id": pet["kind_id"]})
pet["kind_name"] = kind["kind_name"]
pet["food"] = kind["food"]
pet["noise"] = kind["noise"]
```

This approach gives us flexibility but requires explicit code to combine related data.

Project Structure

Core Files

`create-database.py`

Initializes the database with sample data. This script:

- Drops existing collections (fresh start)
- Creates kind documents (Dog, Cat, Fish) with their properties
- Creates pet documents with references to kinds
- Demonstrates how to establish document references

Key learning points:

- Using `insert_many()` for bulk inserts
- Establishing references between documents
- Database initialization patterns

`database.py`

Contains all database operations, separated into two logical groups:

Pet Operations:

- `retrieve_pets()` - Gets all pets with their kind information (performs manual join)
- `retrieve_pet(id)` - Gets a single pet by ID
- `create_pet(data)` - Inserts a new pet document
- `update_pet(id, data)` - Updates an existing pet document
- `delete_pet(id)` - Removes a pet document

Kind Operations:

- `retrieve_kinds()` - Gets all pet kinds
- `retrieve_kind(id)` - Gets a single kind by ID
- `create_kind(data)` - Inserts a new kind document
- `update_kind(id, data)` - Updates an existing kind document
- `delete_kind(id)` - Removes a kind document

Key patterns:

- Converting between ObjectId and string representations
- Manual joins for displaying related data
- Using `$set` operator for updates

`app.py`

Flask web application providing a complete CRUD interface. Implements RESTful routing patterns:

Pet Routes:

- `GET /list` - Display all pets
- `GET /create` - Show pet creation form
- `POST /create` - Process new pet submission
- `GET /update/<id>` - Show pet update form

- `POST /update/<id>` - Process pet update
- `GET /delete/<id>` - Delete a pet

Kind Routes:

- `GET /kind/list` - Display all kinds
- `GET /kind/create` - Show kind creation form
- `POST /kind/create` - Process new kind submission
- `GET /kind/update/<id>` - Show kind update form
- `POST /kind/update/<id>` - Process kind update
- `GET /kind/delete/<id>` - Delete a kind

CRUD Operations Explained

Building on Topic 09's interactive CRUD operations, we now implement them in a structured application context.

Create (Insert) - Applied from Topic 09

Single document (Topic 09 -> Topic 09):

Topic 09 (interactive):

```
mongoose_collection.insert_one({'name': 'Meercat', 'weight': 2})
```

Topic 10 (application):

```
def create_pet(data):
    pets_collection = pets_db.pets_collection
    data["kind_id"] = ObjectId(data["kind_id"])  # Convert string to 
ObjectId
    pets_collection.insert_one(data)
```

Multiple documents:

```
kind_collection.insert_many([
    {"kind_name": "Dog", "food": "Dog food", "noise": "Bark"},
    {"kind_name": "Cat", "food": "Cat food", "noise": "Meow"}
])
```

Read (Query) - Applied from Topic 09

Find all documents (Topic 09 concept -> Topic 09 function):

Topic 09:

```
list(mongoose_collection.find())
```

Topic 09:

```python
def retrieve_pets():
    pets = list(pets_collection.find())
    # ... process and return
    return pets
```

Find one document by ID:

```python
pet = pets_collection.find_one({"_id": ObjectId(id)})
```

Find with query conditions (Topic 09 operator knowledge):

```python
# Using $gt operator from Topic 09
dogs = list(pets_collection.find({"kind_name": "Dog"}))
```

Update - Applied from Topic 09

MongoDB uses **update operators** like `$set` to modify documents:

```python
def update_pet(id, data):
    pets_collection.update_one(
        {"_id": ObjectId(id)},      # Query to find document
        {"$set": data}               # Update operation
    )
```

The `$set` operator updates specified fields without affecting others. This is safer than replacing the entire document.

Delete

```python
def delete_pet(id):
    pets_collection.delete_one({"_id": ObjectId(id)})
```

Data Integrity Considerations

Unlike relational databases with foreign key constraints, MongoDB doesn't automatically enforce referential integrity. In this application:

- Deleting a kind doesn't automatically delete or update pets referencing it
- The application handles integrity at the application layer
- The `delete_kind()` function includes error handling for this scenario

Example pattern (referenced but not fully implemented here):

```python
def delete_kind(id):
    # Check if any pets reference this kind
    pets_with_kind = pets_collection.find({"kind_id": ObjectId(id)})
    if list(pets_with_kind):
        return "Cannot delete kind: pets still reference it"
    kind_collection.delete_one({"_id": ObjectId(id)})
```

Testing Pattern

The `database.py` file includes comprehensive unit tests for each operation:

```python
def test_retrieve_pets():
    print("test retrieve_pets")
    pets = retrieve_pets()
    assert type(pets) is list
    assert type(pets[0]) is dict
    # More assertions...
```

Testing principles demonstrated:

- Each CRUD operation has a corresponding test
- Tests verify data types and structure
- Tests check actual values match expectations
- Tests clean up after themselves (create then delete)

Run tests:

```
python database.py
```

Web Interface

The Flask application provides a complete web interface with HTML templates:

- **list.html** - Table view of all pets with their kind information
- **create.html** - Form with dropdown to select pet kind
- **update.html** - Pre-populated form for editing pets
- **kind_list.html** - Table view of all kinds
- **kind_create.html** - Form for adding new kinds
- **kind_update.html** - Form for editing kinds

Form Handling Pattern

```
@app.route("/create", methods=["GET", "POST"])
def get_post_create():
    if request.method == "GET":
        # Display form with available kinds
        kinds = database.retrieve_kinds()
        return render_template("create.html", kinds=kinds)

    if request.method == "POST":
        # Process form submission
        data = dict(request.form)
        data["age"] = int(data["age"])  # Type conversion
        database.create_pet(data)
        return redirect(url_for("get_list"))
```

Getting Started

1. Install Dependencies

```
pip install flask mongita
```

2. Initialize the Database

```
python create-database.py
```

This creates the `pets_db` directory containing Mongita's data files.

3. Run Tests (Optional)

```
python database.py
```

Verify all CRUD operations work correctly.

4. Start the Flask Application

```
flask --app app run
```

Visit http://localhost:5000/ to interact with the application.

Transition to MongoDB

The code in this topic can be easily adapted to use real MongoDB:

```
# Mongita (current)
from mongita import MongitaClientDisk
client = MongitaClientDisk()

# MongoDB (future)
from pymongo import MongoClient
client = MongoClient("mongodb://localhost:27017/")
```

All other code remains the same! This demonstrates Mongita's value as a learning tool - everything you learn applies directly to production MongoDB.

Key Takeaways

1. **Documents vs. Tables**: MongoDB stores flexible JSON-like documents instead of fixed-schema rows
2. **Collections vs. Tables**: Collections hold documents but don't enforce schemas
3. **ObjectId**: Unique identifiers automatically generated for each document
4. **Manual References**: Application-level joins by storing ObjectIds across documents
5. **Flexible Schema**: Different documents in the same collection can have different fields
6. **CRUD Operations**: MongoDB provides simple methods for insert, find, update, and delete
7. **No Built-in Constraints**: Referential integrity must be handled at the application layer
8. **Mongita for Learning**: Perfect way to learn MongoDB concepts without infrastructure overhead

Next Steps

After mastering these concepts with Mongita:

- **Topic 11**: Learn to connect to MongoDB Atlas (cloud-hosted MongoDB)
- **Topic 12**: Set up and manage a local MongoDB server
- **Topic 13**: Explore advanced querying techniques including aggregation pipelines
- **Topic 14**: Model complex relationships like social media data