# Topic 03: Database Abstraction Layer

## Overview

This topic introduces the critical software engineering practice of **separation of concerns** by extracting database operations into a dedicated module. Building on Topic 02's Flask application, we now separate the "what" (application logic) from the "how" (database implementation). This abstraction makes code more maintainable, testable, and reusable.

## What Changed from Topic 02?

### Before (Topic 02): Database Code in Routes

```
@app.route("/list", methods=["GET"])
def get_list():
    cursor = connection.cursor()
    cursor.execute("""select * from pets""")
    rows = cursor.fetchall()
    return render_template("list.html", rows=rows)
```

### After (Topic 03): Clean Abstraction

```
@app.route("/list", methods=["GET"])
def get_list():
    pets = database.get_pets()
    return render_template("list.html", pets=pets)
```

**Key improvements:**

- No SQL in the route handler
- No cursor management in web code
- Single line replaces multiple database operations
- Route handler focuses on HTTP concerns, not database details

## The Abstraction Layer: database.py

The new `database.py` module encapsulates all database operations, providing a clean API for the application to use.

### 1. Centralized Connection Management

```
connection = None

def initialize(database_file):
    global connection
    connection = sqlite3.connect(database_file, check_same_thread=False)
    connection.row_factory = sqlite3.Row
```

**Key concepts:**

- **Module-level connection**: Single connection shared across all functions
- **Initialize function**: Explicit setup called once at application startup
- **Configurable database**: Can specify different files (e.g., test vs production)
- **Row factory**: Critical improvement that changes how results are returned

### 2. Row Factory - The Game Changer

```
connection.row_factory = sqlite3.Row
```

This single line transforms database access:

**Without row_factory (Topic 02):**

```
rows = cursor.fetchall()
# Returns: [(1, 'Suzy', 'dog', 3, 'Greg'), ...]
# Access: row[0], row[1], row[2]... (positional, fragile)
```

**With row_factory (Topic 03):**

```
rows = cursor.fetchall()
# Returns: [<sqlite3.Row>, ...]
# Access: row['id'], row['name'], row['type']... (named, robust)
```

**Benefits:**

- **Named access**: `pet['name']` instead of `pet[1]`
- **Self-documenting**: Code shows what data means
- **Resilient to schema changes**: Adding columns doesn't break code
- **Dictionary conversion**: Can easily convert to dict for templates

### 3. Converting Rows to Dictionaries

```
def get_pets():
    cursor = connection.cursor()
    cursor.execute("""select * from pets""")
    pets = cursor.fetchall()
    pets = [dict(pet) for pet in pets]  # Convert Row objects to dicts
    return pets
```

**Why convert to dictionaries?**

- **Template compatibility**: Jinja2 works naturally with dicts
- **JSON serialization**: Easy to convert to JSON for APIs
- **Standard Python**: No SQLite-specific types in application code
- **Explicit contract**: Function clearly returns list of dicts

**List comprehension breakdown:**

```
pets = [dict(pet) for pet in pets]
# Equivalent to:
result = []
for pet in pets:
    result.append(dict(pet))
pets = result
```

### 4. Read Operations - Returning Data

**Get all pets:**

```
def get_pets():
    cursor = connection.cursor()
    cursor.execute("""select * from pets""")
    pets = cursor.fetchall()
    pets = [dict(pet) for pet in pets]
    return pets
```

**Get single pet by ID:**

```
def get_pet(id):
    cursor = connection.cursor()
    cursor.execute(f"""select * from pets where id = ?""", (id,))
    rows = cursor.fetchall()
    try:
        (id, name, xtype, age, owner) = rows[0]
        data = {"id": id, "name": name, "type": xtype, "age": age,
"owner": owner}
        return data
    except:
        return "Data not found."
```

**Note**: This function still uses tuple unpacking rather than the Row factory pattern. Could be improved:

```
def get_pet(id):
    cursor = connection.cursor()
    cursor.execute("""select * from pets where id = ?""", (id,))
    pet = cursor.fetchone()
    if pet:
        return dict(pet)
    return None
```

## 5. Write Operations - Modifying Data

**Create (Insert):**

```
def create_pet(data):
    try:
        data["age"] = int(data["age"])
    except:
        data["age"] = 0

    cursor = connection.cursor()
    cursor.execute(
        """insert into pets(name, age, type, owner) values (?,?,?,?)""",
        (data["name"], data["age"], data["type"], data["owner"])
    )
    connection.commit()
```

**Key features:**

- Validates/converts age to integer
- Uses parameterized query for safety
- Commits transaction immediately
- Takes a dictionary, returns nothing (could be improved to return new ID)

**Update:**

```
def update_pet(id, data):
    try:
        data["age"] = int(data["age"])
    except:
        data["age"] = 0

    cursor = connection.cursor()
    cursor.execute(
        """update pets set name=?, age=?, type=?, owner=? where id=?""",
        (data["name"], data["age"], data["type"], data["owner"], id)
    )
    connection.commit()
```

**Delete:**

```
def delete_pet(id):
    cursor = connection.cursor()
    cursor.execute("""delete from pets where id = ?""", (id,))
    connection.commit()
```

**Common pattern:**

- Accept simple parameters (ID, data dictionary)
- Handle database operations internally
- Commit changes before returning
- Return nothing (void functions) for write operations

## Simplified Application Code

With the abstraction layer, `app.py` becomes dramatically cleaner:

### Before Abstraction (Topic 02):

```
@app.route("/create", methods=["POST"])
def post_create():
    data = dict(request.form)
    try:
        data["age"] = int(data["age"])
    except:
        data["age"] = 0
    cursor = connection.cursor()
    cursor.execute("""insert into pets(name, age, type, owner) values
(?,?,?,?)""",
        (data["name"], data["age"], data["type"], data["owner"]))
    connection.commit()
    return redirect(url_for("get_list"))
```

### After Abstraction (Topic 03):

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

**Benefits:**

- **Simpler routes**: Focus on HTTP request/response
- **No SQL in view layer**: Separation of concerns

- **Easier to read**: Intent is clear at a glance
- **Less duplication**: Validation logic centralized in database module

## Testing Infrastructure

The abstraction layer enables proper testing:

### Test Database Setup

```
def setup_test_database():
    initialize("test_pets.db")
    cursor = connection.cursor()
    cursor.execute("""
        create table if not exists pets (
            id integer primary key autoincrement,
            name text not null,
            type text not null,
            age integer,
            owner text
        )
    """)
    connection.commit()

    pets = [
        {"name": "dorothy", "type": "dog", "age": 9, "owner": "greg"},
        {"name": "suzy", "type": "mouse", "age": 9, "owner": "greg"},
        {"name": "casey", "type": "dog", "age": 9, "owner": "greg"},
        {"name": "heidi", "type": "cat", "age": 15, "owner": "david"},
    ]
    for pet in pets:
        create_pet(pet)

    pets = get_pets()
    assert len(pets) == 4
```

**Testing principles:**

- **Separate test database**: `test_pets.db` instead of `pets.db`
- **Reproducible setup**: Creates fresh database with known data
- **Uses same functions**: Tests the actual database module code
- **Assertions**: Verifies expected behavior

### Unit Test Example

```
def test_get_pets():
    print("testing get_pets")
    pets = get_pets()

    # Test structure
```

```python
    assert type(pets) is list
    assert len(pets) > 0
    assert type(pets[0]) is dict

    # Test content
    pet = pets[0]
    for field in ["id", "name", "type", "age", "owner"]:
        assert field in pet, f"Field {field} missing from {pet}"

    # Test types
    assert type(pet["id"]) is int
    assert type(pet["name"]) is str
```

**What this tests:**

- Return type is correct (list of dicts)
- Data structure is correct (has expected fields)
- Field types are correct (id is int, name is str)
- Function actually returns data

**Running tests:**

```
python database.py
```

The `if __name__ == "__main__"` block runs tests when the module is executed directly.

## Benefits of This Abstraction

### 1. Separation of Concerns

**Web layer (app.py):**

- Handles HTTP requests and responses
- Manages routing and URL patterns
- Renders templates
- Processes form data
- Redirects users

**Data layer (database.py):**

- Manages database connections
- Executes SQL queries
- Handles transactions
- Converts data formats
- Validates input

Each layer has a clear responsibility and doesn't need to know implementation details of the other.

### 2. Testability

With the abstraction layer, you can:

- Test database operations independently of Flask
- Use a test database without affecting production data
- Write unit tests for each database function
- Verify data structure and types
- Run tests automatically

```
# Test just the database module
python database.py

# Test the web application
flask --app app run
```

### 3. Reusability

Database functions can be used from:

- Flask web applications
- Command-line scripts
- Other web frameworks
- Background jobs
- API endpoints
- Admin tools

The same `get_pets()` function works everywhere.

### 4. Maintainability

**Changing database schema?** Update only `database.py`:

```
# Add a new field
cursor.execute("""
    insert into pets(name, age, type, owner, breed)
    values (?,?,?,?,?)
""", (data["name"], data["age"], data["type"], data["owner"],
data["breed"]))
```

Web routes don't need to change if the function interface stays the same.

**Changing database system?** Swap out SQLite for PostgreSQL in `database.py`, application code unchanged.

### 5. Debugging

When there's a database problem:

- Look in `database.py` (single source of truth)

- Add debug print statements in one place
- Fix bugs once, benefits entire application

**Debug example:**

```python
def get_pets():
    cursor = connection.cursor()
    cursor.execute("""select * from pets""")
    pets = cursor.fetchall()
    pets = [dict(pet) for pet in pets]
    for pet in pets:
        print(pet)  # Debug: see what's returned
    return pets
```

## Comparing the Two Architectures

### Topic 02: Monolithic Routes

```
/--------------------------------------\
|            app.py                    |
|  /-------------------------------\  |
|  | Route Handler                  |  |
|  |  - Parse HTTP request          |  |
|  |  - Create cursor               |  |
|  |  - Execute SQL                 |  |
|  |  - Fetch results               |  |
|  |  - Commit transaction          |  |
|  |  - Render template             |  |
|  |  - Return HTTP response        |  |
|  \-------------------------------/  |
\--------------------------------------/
```

### Topic 03: Layered Architecture

```
/--------------------------------------\
|            app.py                    |
|  /-------------------------------\  |
|  | Route Handler                  |  |
|  |  - Parse HTTP request          |  |
|  |  - Call database.get_pets()    |  |
|  |  - Render template             |  |
|  |  - Return HTTP response        |  |
|  \-------------------------------/  |
\--------------------------------------/
                  |
/--------------------------------------\
|          database.py                 |
|  /-------------------------------\  |
|  | Database Function              |  |
|  |  - Create cursor               |  |
|  |  - Execute SQL                 |  |
|  |  - Fetch results               |  |
|  |  - Convert to dict             |  |
|  |  - Commit transaction          |  |
|  |  - Return data                 |  |
|  \-------------------------------/  |
\-------------------------------------/
```

## Application Initialization

```
# app.py
import database

database.initialize("pets.db")

app = Flask(__name__)
```

**Initialization flow:**

1. Import database module
2. Call `initialize()` with database filename
3. Database module creates connection with Row factory
4. Application routes can now use database functions

This explicit initialization pattern is clearer than Topic 02's global connection creation.

## What This Abstraction Doesn't Solve

While this is a significant improvement, some issues remain:

1. **Connection lifecycle**: Still no proper connection closing
2. **Connection pooling**: Still using single global connection
3. **Error handling**: Generic exceptions, not user-friendly
4. **Validation**: Minimal input validation
5. **Transactions**: Auto-commit after each operation (no multi-operation transactions)
6. **SQL duplication**: Still writing repetitive SQL statements
7. **Relationships**: No support for related entities (foreign keys, joins)

**Topics 04-05** will address these remaining concerns.

## Running the Application

### 1. Ensure Database Exists

Use Topic 01's notebook or create manually:

```
sqlite3 pets.db < schema.sql
```

### 2. Run Application

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

### 3. Run Tests

```
python database.py
```

Creates `test_pets.db` and runs unit tests.

## Key Concepts and Terminology

- **Abstraction Layer**: Module that hides implementation details
- **Separation of Concerns**: Different responsibilities in different modules
- **Row Factory**: SQLite feature for named column access
- **sqlite3.Row**: Row object that supports dict-like access
- **List Comprehension**: Concise syntax for transforming lists
- **Unit Testing**: Testing individual functions in isolation
- **Test Database**: Separate database for testing without affecting production

## Design Patterns Applied

### 1. Module Pattern

All related functions grouped in `database.py` module.

### 2. Facade Pattern

Database module provides simplified interface to complex SQL operations.

### 3. Data Transfer Object (DTO)

Functions return dictionaries (simple data structures) rather than complex objects.

## Best Practices Demonstrated

1. ✅ Separate business logic from presentation
2. ✅ Use Row factory for named access
3. ✅ Convert database results to standard Python types
4. ✅ Centralize connection management
5. ✅ Make database configurable (different files for test/prod)
6. ✅ Write unit tests for data layer

7. ✅ Document function behavior with tests
8. ✅ Use explicit initialization pattern

## Migration Path from Topic 02

If you have Topic 02 code, migrate to this pattern:

1. Create `database.py` module
2. Move connection creation to `initialize()` function
3. Extract each database operation into a function
4. Add Row factory to connection
5. Convert results to dictionaries
6. Update app.py to use database functions
7. Write tests for each database function

## Key Takeaways

1. **Abstraction layers separate concerns** - web code handles HTTP, data code handles databases
2. **Row factory enables named access** - more robust than positional access
3. **Dictionary conversion** creates clean interface between layers
4. **Testing becomes possible** when database operations are in their own module
5. **Code becomes more maintainable** with single source of truth for data operations
6. **Functions provide clean API** - clear inputs and outputs
7. **Explicit initialization** is better than hidden globals
8. **This pattern scales** to larger applications and teams

This abstraction layer is a fundamental architectural pattern that appears in professional applications. While we'll introduce ORMs (Object-Relational Mappers) in later topics, understanding this manual abstraction helps you appreciate what ORMs do automatically and gives you the skills to work without them when needed.