Topic 02: Initial Flask Application

Overview

This topic bridges the gap between basic database operations (Topic 01) and web applications. You'll learn how to build a **Flask web application** that provides a complete browser-based interface for database CRUD operations. This represents the first step toward building real-world database-driven web applications, where users interact with data through web forms and pages rather than running Python scripts directly.

What is Flask?

Flask is a lightweight web framework for Python that makes it easy to build web applications. It's:

- **Micro-framework**: Minimal core with extensions available as needed
- **Easy to learn**: Simple, intuitive API with minimal boilerplate
- **Flexible**: Doesn't enforce specific patterns or structures
- **Perfect for learning**: Small enough to understand completely
- **Production-ready**: Used by major companies despite its simplicity

Flask handles:

- HTTP request/response processing
- URL routing (mapping URLs to Python functions)
- Template rendering (generating HTML with dynamic data)
- Form data processing
- Redirects and error handling

Key Concepts

1. Flask Application Setup

```
from flask import Flask, render_template, request, redirect, url_for
import sqlite3

connection = sqlite3.connect("pets.db", check_same_thread=False)
app = Flask(__name__)
```

Breaking it down:

- `Flask(__name__)` creates the application instance
- `__name__` helps Flask locate templates and static files
- `connection = sqlite3.connect(...)` creates a **global database connection**
- `check_same_thread=False` allows the connection to be used across different requests

[!] Note on Global Connection: This approach works for learning but has limitations:

- Not thread-safe for production with multiple concurrent users
- Connection isn't properly closed
- No connection pooling
- Topic 03 will address these issues with better patterns

2. Routes and HTTP Methods

Routes map URLs to Python functions. Flask supports different HTTP methods for different actions.

Basic GET route:

```
@app.route("/", methods=["GET"])
def get_index():
    return "Example flask server."
```

- `@app.route()` decorator defines the URL pattern
- `methods=["GET"]` specifies this handles GET requests (default)
- Function returns a string response

Route with parameters:

```
@app.route("/hello/<name>", methods=["GET"])
def get_hello(name="world"):
    return render_template("hello.html", prof={"name": name, "title":
"Dr."})
```

- `<name>` captures URL segments as parameters
- `name="world"` provides a default value
- URL `/hello/Alice` passes "Alice" to the function

Multiple routes for one function:

```
@app.route("/hello", methods=["GET"])
@app.route("/hello/<name>", methods=["GET"])
def get_hello(name="world"):
    # Handles both /hello and /hello/Alice
```

3. Template Rendering with Jinja2

Flask uses the **Jinja2** template engine to generate dynamic HTML.

Rendering a template:

```
def get_hello(name="world"):
    data = [
```

```
        {"name": "bob", "age": 10},
        {"name": "suzy", "age": 8},
    ]
    return render_template("hello.html", data=data, prof={"name": name,
"title": "Dr."})
```

Template syntax (hello.html):

```
<html>
    <h1>Hello {{ prof["title"] }} {{ prof["name"] }}!</h1>
    <table>
        {% for item in data %}
        <tr>
            <td>{{ item["name"] }}</td><td>{{ item["age"] }}</td>
        </tr>
        {% endfor %}
    </table>
</html>
```

Jinja2 syntax:

- `{{ variable }}` - Outputs a variable value
- `{% for ... %}` - Control structure (loops, conditionals)
- Variables passed from Python are available in the template
- Supports dictionaries, lists, and complex data structures

4. Reading Data - List View

Route handler:

```
@app.route("/list", methods=["GET"])
def get_list():
    cursor = connection.cursor()
    cursor.execute("""select * from pets""")
    rows = cursor.fetchall()
    for row in rows:
        print(row)  # Debug output to console
    return render_template("list.html", rows=rows)
```

Process:

1. Create cursor from global connection
2. Execute SELECT query to get all pets
3. Fetch all results as a list of tuples
4. Pass data to template for rendering

Template (list.html):

```
<table>
    <tr>
        <th>ID</th><th>Name</th><th>Type</th><th>Age</th><th>Owner</th>
    </tr>
    {% for row in rows %}
    <tr>
        {% for item in row %}
        <td>{{ item }}</td>
        {% endfor %}
        <td><a href="/delete/{{row[0]}}">Delete</a></td>
        <td><a href="/update/{{row[0]}}">Update</a></td>
    </tr>
    {% endfor %}
</table>
```

Key features:

- Nested loops: outer loop for rows, inner loop for columns
- `row[0]` accesses the ID (first column) for action links
- Links include the ID in the URL for record-specific operations

5. Creating Data - Form Handling

Creating data requires TWO routes with the same URL but different HTTP methods:

GET route - Display the form:

```
@app.route("/create", methods=["GET"])
def get_create():
    return render_template("create.html")
```

POST route - Process form submission:

```
@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"))
```

Key concepts:

- **Form data**: `request.form` contains submitted form data
- **Type conversion**: HTML forms send everything as strings, must convert age to int
- **Error handling**: try/except prevents crashes from invalid input
- **Parameterized query**: Uses `?` placeholders for safe SQL
- **Commit**: Required to save changes to database
- **Redirect**: After POST, redirect to avoid duplicate submissions on refresh

Form template (create.html):

```
<form action="/create" method="post">
    <p>Animal Name:<input name="name"/></p>
    <p>Age:<input name="age"/></p>
    <p>Type:<input name="type"/></p>
    <p>Owner:<input name="owner"/></p>
    <button type="submit">Create</button>
    <a href="/list">Cancel</a>
</form>
```

- `action="/create"` specifies where to send form data
- `method="post"` uses POST (appropriate for data modification)
- `name` attribute on inputs becomes the key in `request.form`

6. Updating Data - Pre-populating Forms

Updating requires THREE operations:

GET route - Retrieve and display current data:

```
@app.route("/update/<id>", methods=["GET"])
def get_update(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
        }
    except:
        return "Data not found."
    return render_template("update.html", data=data)
```

Process:

1. Extract ID from URL parameter
2. Query database for that specific pet
3. Unpack tuple into individual variables
4. Convert to dictionary for easy template access
5. Handle case where pet doesn't exist

POST route - Save updated data:

```
@app.route("/update/<id>", methods=["POST"])
def post_update(id):
    data = dict(request.form)
    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()

    return redirect(url_for("get_list"))
```

Update template (update.html):

```
<form action="/update/{{data['id']}}" method="post">
    <p>Animal Name:<input name="name" value="{{data['name']}}"/></p>
    <p>Age:<input name="age" value="{{data['age']}}"/></p>
    <p>Type:<input name="type" value="{{data['type']}}"/></p>
    <p>Owner:<input name="owner" value="{{data['owner']}}"/></p>
    <button type="submit">Update</button>
    <a href="/list">Cancel</a>
</form>
```

Key difference from create:

- `value` attributes pre-populate form fields with current data
- Form action includes the ID: `/update/{{data['id']}}`
- User sees current values and can modify them

7. Deleting Data - Simple GET Handler

```
@app.route("/delete/<id>", methods=["GET"])
def get_delete(id):
```

```
    cursor = connection.cursor()
    cursor.execute(f"""delete from pets where id = ?""", (id,))
    connection.commit()
    return redirect(url_for("get_list"))
```

Why GET instead of POST?

- Simplicity: Can be triggered by a simple link
- In production, DELETE operations should use POST/DELETE methods
- This demonstrates basic concepts; Topic 03+ will improve patterns

Delete link in list template:

```
<a href="/delete/{{row[0]}}">Delete</a>
```

Clicking this link triggers the delete operation immediately.

RESTful Patterns Emerging

This application begins to follow **REST** (Representational State Transfer) principles:

Operation	HTTP Method	URL Pattern	Purpose
List all	GET	/list	Display all pets
Show form	GET	/create	Display creation form
Create	POST	/create	Process new pet
Show form	GET	/update/<id>	Display update form
Update	POST	/update/<id>	Process pet update
Delete	GET	/delete/<id>	Remove pet

REST principles demonstrated:

- URLs represent resources (pets)
- HTTP methods indicate operations (GET=read, POST=modify)
- GET operations are idempotent (safe to repeat)
- POST operations modify data

The Request-Response Cycle

Understanding web application flow:

```
1. Browser -> GET /list -> Flask
2. Flask executes get_list()
3. Queries database
4. Renders template with data
```

```
5. Flask -> HTML response -> Browser
6. User clicks "Create"
7. Browser -> GET /create -> Flask
8. Flask renders create form
9. Flask -> HTML form -> Browser
10. User fills form and submits
11. Browser -> POST /create (with form data) -> Flask
12. Flask processes data
13. Inserts into database
14. Flask -> Redirect to /list -> Browser
15. Browser -> GET /list -> Flask (repeat cycle)
```

Why redirect after POST?

- Prevents duplicate submissions if user refreshes
- Follows **Post/Redirect/Get (PRG)** pattern
- Clean URL in browser after operation

URL Building with url_for()

```
return redirect(url_for("get_list"))
```

Instead of hardcoding URLs:

```
return redirect("/list")  # Fragile
```

Use `url_for()`:

- References function name, not URL string
- Automatically generates correct URL
- If you change route URLs, links still work
- Handles URL parameters automatically

Error Handling and Validation

Type conversion with error handling:

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

- User might enter non-numeric age
- Conversion fails gracefully, defaults to 0
- Prevents server crash from invalid input

Database query error handling:

```
try:
    (id, name, xtype, age, owner) = rows[0]
    # ... create data dict ...
except:
    return "Data not found."
```

- Handles case where pet ID doesn't exist
- Returns user-friendly error message
- Prevents IndexError on empty results

[!] Limitations:

- Generic error messages (not user-friendly)
- No input validation before database operations
- Silent failures (except clause too broad)
- Later topics will demonstrate better error handling

Template Organization

Templates are stored in the `templates/` folder by default:

```
topic-02-initial-flask-app/
|-- app.py
|-- pets.db
\-- templates/
    |-- create.html
    |-- hello.html
    |-- list.html
    \-- update.html
```

Flask automatically finds templates in this folder when using `render_template()`.

Running the Application

1. Install Flask

```
pip install flask
```

2. Set Up Database

You'll need a `pets.db` file with a pets table. Create it using Topic 01's code or run:

```
sqlite3 pets.db
```

```
create table pets (
    id integer primary key autoincrement,
    name text not null,
    type text not null,
    age integer,
    owner text
);

insert into pets(name, type, age, owner) values
    ('Suzy', 'dog', 3, 'Greg'),
    ('Sandy', 'cat', 2, 'Steve');

.quit
```

3. Run Flask Application

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

Or with auto-reload for development:

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

4. Access in Browser

Open: http://localhost:5000/

Available routes:

- http://localhost:5000/ - Simple message
- http://localhost:5000/hello - Template demo
- http://localhost:5000/hello/Alice - Parameterized route
- http://localhost:5000/list - Pet listing (main CRUD interface)
- http://localhost:5000/create - Add new pet
- http://localhost:5000/bye - Simple goodbye message

Debugging Tips

Debug output:

```
print(data)  # Appears in terminal running Flask
```

Check browser developer tools:

- Network tab shows requests/responses
- Console shows JavaScript errors
- Elements tab shows rendered HTML

Enable debug mode:

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

Benefits:

- Auto-reload on code changes
- Detailed error pages with stack traces
- Interactive debugger in browser

Current Limitations

This initial implementation has several issues addressed in later topics:

1. **Global connection**: Not thread-safe, no proper lifecycle management
2. **No abstraction**: Database code mixed with web code
3. **Repetitive code**: Similar patterns repeated for each operation
4. **Limited error handling**: Generic errors, no validation
5. **SQL in web routes**: Business logic mixed with presentation
6. **No relationships**: Can't model related data properly
7. **Basic HTML**: No styling, minimal user experience

Topic 03 addresses many of these by introducing database abstraction layers.

Complete CRUD Workflow Example

Typical user journey:

1. Visit `/list` - See all pets
2. Click "Create New Pet"
3. Fill out form with pet details
4. Submit - Redirected to `/list` with new pet
5. Click "Update" on a pet
6. Modify pet's age
7. Submit - Redirected to `/list` with updated data
8. Click "Delete" on a pet
9. Pet removed, view refreshes

Key Concepts and Terminology

- **Flask**: Lightweight Python web framework
- **Route**: URL pattern mapped to a Python function
- **HTTP Methods**: GET (retrieve), POST (submit data), DELETE, etc.
- **Template**: HTML file with dynamic placeholders

- **Jinja2**: Template engine for generating HTML
- **render_template()**: Generates HTML from template and data
- **request.form**: Dictionary-like object containing form data
- **redirect()**: Sends user to a different URL
- **url_for()**: Generates URLs from function names
- **Post/Redirect/Get**: Pattern to prevent duplicate submissions

Best Practices Demonstrated

1. ☑ Separate GET and POST handlers for forms
2. ☑ Use templates for HTML generation (not string concatenation)
3. ☑ Redirect after POST operations (PRG pattern)
4. ☑ Use `url_for()` instead of hardcoded URLs
5. ☑ Parameterized queries for SQL safety
6. ☑ Try/except for type conversion and error handling
7. ☑ Organize templates in dedicated folder

Common Pitfalls to Avoid

[!] **Returning HTML strings directly:**

```
return "<html><body>Hello</body></html>"  # Unmaintainable
```

[!] **Hardcoded URLs in redirects:**

```
return redirect("/list")  # Use url_for() instead
```

[!] **Forgetting to commit:**

```
cursor.execute("insert into ...")
# Missing: connection.commit()
```

[!] **Not handling form data types:**

```
data = dict(request.form)
# age is still a string! Need to convert.
```

Connection to Future Topics

- **Topic 03**: Database abstraction - Separates database operations into dedicated module
- **Topic 04**: Keys and joins - Models relationships between entities
- **Topic 05**: ORM (Peewee) - Eliminates raw SQL with object mapping

- **Topic 06**: Dataset library - Simplifies database operations further

Key Takeaways

1. **Flask makes web development simple** with minimal boilerplate
2. **Routes map URLs to functions** that handle requests and return responses
3. **Templates separate HTML from Python** using Jinja2 syntax
4. **Forms use GET to display, POST to submit** data for processing
5. **Redirect after POST** prevents duplicate submissions
6. **url_for() generates URLs** from function names for maintainability
7. **Database operations follow the same pattern** as Topic 01, now in a web context
8. **This is just the beginning** - later topics will improve code organization and safety

This topic establishes the foundation for database-driven web applications, demonstrating how to connect browser interfaces with database operations using Flask's routing and templating capabilities.