Database-driven Web Applications

Databases and Interfaces

Matthew Pike & Yuan Yao

University of Nottingham Ningbo China (UNNC)

Overview

This Lecture

- Finally, we bring together all the concepts we have learned so far to create a database-driven web application using Flask and SQLite
- We will look at how to:
  - Execute SQL commands using Python
  - Present the results to the user via a web interface
- We will also look at how to handle errors and make our web applications more robust

The Database Schema for this Lecture

```
CREATE TABLE Student(
    sID INTEGER PRIMARY KEY,
    firstName VARCHAR(20) NOT NULL,
    lastName VARCHAR(20) NOT NULL
);
```

```
CREATE TABLE Module(
    mCode CHAR(8) PRIMARY KEY,
    title VARCHAR(30) NOT NULL,
    credits INTEGER NOT NULL
);
```

```
CREATE TABLE Grade(
    sID INTEGER NOT NULL,
    mCode CHAR(8) NOT NULL,
    grade INTEGER NOT NULL,
    PRIMARY KEY (sID, mCode),
    FOREIGN KEY (sID)
        REFERENCES Student(sID),
    FOREIGN KEY (mCode)
        REFERENCES Module(mCode)
);
```

The Database Content for this Lecture

sID	firstName	lastName
1	John	Smith
2	Jane	Doe
3	Mary	Jones
4	David	Smith

Table 1: `Student` Table

mCode	title	credits
COMP1036	Fundamentals	20
COMP1048	Databases	10
COMP1038	Programming	20

Table 2: `Module` Table

sID	mCode	grade
1	COMP1036	35
1	COMP1048	50
2	COMP1048	65
2	COMP1038	70
3	COMP1036	35
3	COMP1038	65

Table 3: `Grade` Table

Database-driven Web Applications

- In the last lecture we looked at how to create a web application using Flask
  - We saw how to create a simple web application that adapted to user input
- This lecture advances our skills to integrate a database into our Flask web app.
- The integration of a database enables:
  - **Data Persistence**: Ability to store and retrieve data over time, beyond a single session.
  - **Enhanced Interactivity**: Creating dynamic content that evolves based on stored data and user interactions.
  - **Complex Functionalities**: Implementing features like user accounts, data analytics, and personalised user experiences.
- Understanding how to build database-driven applications is an important skill in academia and industry.

Using SQLite with Python

The SQLite Module (from the Python Standard Library)

- Before we can create a database driven web application, we need to understand how to interact with a database using Python
- Python provides a built-in module, `sqlite3`, that allows us to interact with SQLite databases
  - `import sqlite3`
- We can now use our existing SQL knowledge to interact with SQLite databases from Python/Flask

Example: Connecting to a Database and Executing SELECT

```
import sqlite3

conn = sqlite3.connect("Students.db") # Connect to the database
conn.row_factory = sqlite3.Row # Make the results easier to work with
cur = conn.cursor() # Create a cursor object
# Execute SQL commands using the cursor object and fetch the results
cur.execute("SELECT * FROM Student")

rows = cur.fetchall()
# Print the results
for row in rows:
  print(row["sID"], row["firstName"], row["lastName"])

conn.close() # Close the connection
```

Connecting and Executing SQL Commands (1/2)

1. Establishing a Connection
   - **Database Connection**: Use `sqlite3.connect('database_name.db')` to establish a connection to an SQLite database.

2. Specifying how to return results (row_factory)
   - **Row Factory**: By default, the results of a query are returned as a list of tuples. We can use `connection.row_factory = sqlite3.Row` to return results as a list of dictionaries instead - this makes the results easier to work with.

3. Creating a Cursor
   - **Creating a Cursor**: A Cursor object is used to execute SQL commands and manage transactions. We can create a Cursor object using `connection.cursor()`.
   - **Executing Queries**: The cursor is used to execute SQL statements (e.g., `cursor.execute("SELECT * FROM table_name")`).

Connecting and Executing SQL Commands (2/2)

4. Fetching Results or Committing Changes
   - **Fetching Results**: If we're executing a SELECT statement, we can use `cursor.fetchall()` to get the results of the query. Alternatively, we can use `cursor.fetchone()` to get the first result of the query.
   - **Committing Changes**: If we're executing an `INSERT`, `UPDATE` or `DELETE` statement, we may need to commit the changes to the database using `connection.commit()`.

5. Closing the Connection
   - **Closing the Connection**: Once we're finished with the database, we should close the connection using `connection.close()`.

Working with Results

When we set `row_factory = sqlite3.Row`, the results are returned as a list of dictionaries. Often, this makes the results easier to work with. For example, consider the following code:

```
import sqlite3
conn = sqlite3.connect("Students.db")
conn.row_factory = sqlite3.Row
cur = conn.cursor()
cur.execute("SELECT * FROM Student")

rows = cur.fetchall()
for row in rows:
    # We can access the values in the row using the table column names
    print(f"{row['sID']}: {row['firstName']} {row['lastName']}")

conn.close()
```

INSERT'ing Data using Python

What are `"""` strings? What are f-strings?

- We can use `"""` to create a multi-line string in Python, as shown in the example below.
- f-strings are a convenient way to insert variables into strings. For example, `f"Hello {name}"` will insert the value of the variable `name` into the string.

```
import sqlite3
conn = sqlite3.connect("Students.db")
cur = conn.cursor()
cur.execute(""" INSERT INTO Student
    VALUES (NULL, 'John', 'Smith')
""")
```

```
conn.commit() # Commit the changes to the database
```

Parameterized Queries

- Up to now, our SQL queries have incorporated fixed, or “hard-coded”, values. However, this approach is somewhat limiting.
- **Parameterized queries**, a more dynamic approach, enable the incorporation of Python variables into SQL queries. This is particularly useful for integrating user input securely.
- In these queries, placeholders like `?` are used in SQL statements where variable data is required. For instance:
  - `INSERT INTO Student VALUES (NULL, ?, ?)`
- These placeholders are then replaced by actual values provided as a tuple in the `execute()` method call. For example:
  - `cur.execute("INSERT INTO Student VALUES (NULL, ?, ?)", (firstname, lastname))`

Example: Parameterized Queries

```
import sqlite3
conn = sqlite3.connect("Students.db")
cur = conn.cursor()
firstname = "Dave" # In practice, this would be user input
lastname = "Towey" # as opposed to hard-coded values

cur.execute("""
    INSERT INTO Student
    VALUES (NULL, ?, ?)
""", (firstname, lastname))

conn.commit()
```

Developing Robust and Resilient Web Applications

SQL Injection Attacks

- Handling user input in web applications necessitates vigilance against **SQL injection attacks**.
- Such an attack transpires when malicious SQL code is inputted by a **user**, typically into a form, and subsequently **executed by the database**.
- Imagine this scenario:
  - Using `cur.execute(f"INSERT INTO Student VALUES (NULL, '{firstname}', '{lastname}')")`
    - Note here the use of f-strings rather than parameterized queries - this is bad practice! Don't do this!
- A problematic input like `John'); DROP TABLE Student; --` would result in:
  - `INSERT INTO Student VALUES (NULL, 'John'); DROP TABLE Student; --', 'Smith')`
  - This command, alarmingly, erases the `Student` table.
- To counteract SQL injection - use **parameterized queries** when handling user inputs.

A real-world issue

- In the UK, a company was established with the name: `;DROP TABLE "COMPANIES";--LTD`
- This name is in fact an SQL injection attack
  - The initial semicolon ends the preceding SQL statement
  - `DROP TABLE "COMPANIES"` commands to delete a database table named "COMPANIES".
  - The second semicolon denotes the end of the SQL statement
  - `--` comments out the rest of the SQL statement
- The company was required to change its name due to the security implications.

FILE COPY

CERTIFICATE OF INCORPORATION
OF A
PRIVATE LIMITED COMPANY

Company Number **10542519**

The Registrar of Companies for England and Wales, hereby certifies that

; DROP TABLE "COMPANIES";-- LTD

is this day incorporated under the Companies Act 2006 as a private company, that the company is limited by shares, and the situation of its registered office is in England and Wales

Given at Companies House, Cardiff, on **29th December 2016**

Figure 1: `;DROP TABLE "COMPANIES";--LTD`

XKCD: Exploits of a Mom

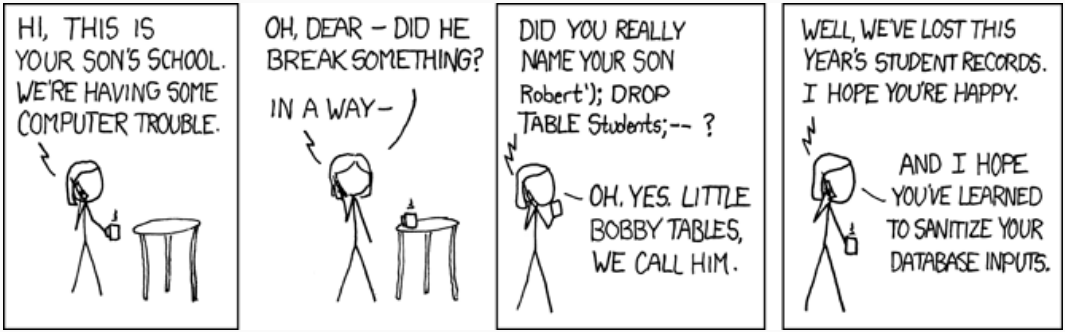

Figure 2: XKCD: Exploits of a Mom - https://xkcd.com/327/

Handling Errors

- When executing SQL commands, various errors can arise, such as:
  - Database accessibility issues (e.g., locked or unavailable)
  - Syntax errors in SQL commands
  - Violations of database constraints
- To prevent these errors from disrupting the application, it's important to manage them gracefully, avoiding unanticipated error messages to users.
- Python's `try-except` structure is instrumental in error handling. It enables us to execute a code block while being prepared to catch and handle any exceptions.
- The typical structure is:
  - `try:`
    - `# Attempt this code`
  - `except ErrorType:`
    - `# Respond to the specific error`
- Knowing which errors to handle is important. You'll need to study the documentation for the functions you're using to determine the appropriate error types.

Example: Handling Errors

```
import sqlite3
try:
    conn = sqlite3.connect("not-available.db")
    cur = conn.cursor()
    cur.execute("SELECT * FROM NotATable")
    rows = cur.fetchall()
except sqlite3.DatabaseError as e:
    print("An error occurred when connecting to the database: ", e)
except sqlite3.OperationalError as e:
    print("Operational error occurred: ", e)
finally:
    # Ensure the connection is closed even if an error occurs
    if conn:
        conn.close()
```

Using SQLite with Flask

- Our culmination point involves integrating Flask (as discussed previously) with SQLite to construct a database-driven web application.
- This process echoes our earlier practices but pivots from static to dynamic data sourcing from a database. The key steps include:
  - Establishing a database connection using - `sqlite3.connect()`.
  - Utilising a cursor object to run SQL commands - `cursor.execute()`.
  - Linking query results with web page templates - `render_template("template.html", rows=rows)`.

Example: Database Driven Web Application (1/2)

Flask (**app.py**)

```
from flask import Flask, render_template
import sqlite3
app = Flask(__name__)
@app.route("/")
def index():
    conn = sqlite3.connect("Students.db")
    conn.row_factory = sqlite3.Row
    cur = conn.cursor()
    cur.execute("SELECT * FROM Student")
    rows = cur.fetchall()
    conn.close()
    return render_template("index.html", rows=rows)
```

Example: Database Driven Web Application (2/2)

Jinja Template (**index.html**)

```
<!DOCTYPE html>
<html>
    <head><title>Students</title></head>
    <body>
        <h1>Students</h1>
        <ul>
            {% for row in rows %}
                <li> {{ row["firstname"] }} {{ row["lastname"] }} </li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Resources

- A comprehensive example of a database driven web application using Flask and SQLite is provided on Moodle
- Flask Mega-Tutorial
  - https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world
- Using Flask with SQLite
  - https://flask.palletsprojects.com/en/2.3.x/patterns/sqlite3/
- How To Use an SQLite Database in a Flask Application
  - https://www.digitalocean.com/community/tutorials/how-to-use-an-sqlite-database-in-a-flask-application