Database-driven Web Applications

COMP1048: Databases and Interfaces (2024-2025)

Matthew Pike and Yuan Yao

Overview

This Lecture

Examples on Moodle

The examples and code snippets used in this lecture are available on Moodle.

- Finally, we will integrate all the concepts learned so far to build a database-driven web application using Flask and SQLite.
- This will include:
    - Executing SQL commands through Python.
    - Displaying the results to users via a web interface.
- Additionally, we will explore how to handle errors effectively and enhance the robustness of our web applications.

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 previous lecture, we explored how to create a web application using Flask.
  - We learned how to develop a simple web application that responded to user input.
- This lecture builds on those skills by integrating a database into our Flask web application.
- Adding a database enables:
  - **Data Persistence**: Storing and retrieving data over time, beyond the scope of a single session.
  - **Enhanced Interactivity**: Generating dynamic content that evolves based on stored data and user interactions.
  - **Complex Functionalities**: Supporting features such as user accounts, data analytics, and personalised user experiences.
- Developing database-driven applications is a valuable skill, both in academia and industry.

Using SQLite with Python

The SQLite Module (from the Python Standard Library)

- Before creating a database-driven web application, we need to understand how to interact with a database using Python.
- Python includes a built-in module, `sqlite3`, which allows interaction with SQLite databases.
    - To use it, simply: `import sqlite3`
- With this module, we can apply our existing SQL knowledge to interact with SQLite databases within Python and Flask applications.

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, query results are returned as a list of tuples.
        - To make results easier to work with, set `connection.row_factory = sqlite3.Row` to return results as a list of dictionaries. This allows us to access the values using the column names e.g. `row['sID']`.
3. Creating a Cursor
    - **Creating a Cursor**: A Cursor object is required to execute SQL commands and manage transactions.
        - Create one using `connection.cursor()`.
    - **Executing Queries**: Use the cursor to execute SQL statements, such as `cursor.execute("SELECT * FROM table_name")`.

Connecting and Executing SQL Commands (2/2)

4. Fetching Results or Committing Changes

    - Fetching Results:
        - For `SELECT` statements, use `cursor.fetchall()` to retrieve all query results.
        - Alternatively, use `cursor.fetchone()` to fetch the first result of the query.
    - Committing Changes:
        - For `INSERT`, `UPDATE`, or `DELETE` statements, commit the changes to the database using `connection.commit()`.

5. Closing the Connection

    - Closing the Connection:
        - Once finished with the database, close the connection using `connection.close()` to free resources.

Working with Results

- By setting `row_factory = sqlite3.Row`, the results from SQLite queries are returned as a list of dictionaries.
- This approach often simplifies working with the results, as you can access columns using dictionary-style keys. 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 multi-line strings in Python, as shown in the example below.
- f-strings provide a convenient way to embed variables into strings. For example, `f"Hello {name}"` will insert the value of the variable `name` into the string.

```python
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

- So far, our SQL queries have used static, or “hard-coded”, values. While functional, this approach has its limitations.
- **Parameterized queries** offer a more dynamic and secure way to incorporate Python variables into SQL queries, especially useful when handling user input.
- In parameterized queries, placeholders like `?` are used within the SQL statement to represent variable data. For example:
  - `INSERT INTO Student VALUES (NULL, ?, ?)`
- These placeholders are replaced with actual values provided as a tuple in the `execute()` method. For instance:
  - `cur.execute("INSERT INTO Student VALUES (NULL, ?, ?)", (firstname, lastname))`

Example: Parameterized Queries

```python
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()
```

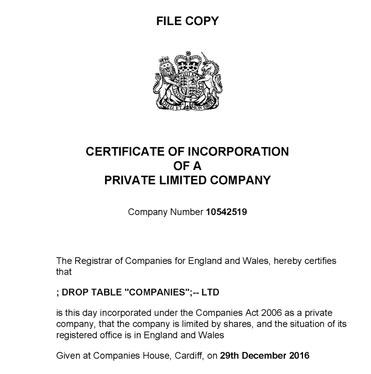
Developing Robust and Resilient Web Applications

SQL Injection Attacks

- Handling user input in web applications requires vigilance against **SQL injection attacks**.
- An SQL injection attack occurs when malicious SQL code is provided by a **user**, often through a form, and is subsequently **executed by the database**.
- Consider this scenario:
  - Using `cur.execute(f"INSERT INTO Student VALUES (NULL, '{firstname}', '{lastname}')")`
    - Note the use of f-strings instead of parameterized queries—this is bad practice! Avoid doing this!
- A malicious input like `John'); DROP TABLE Student; --` would result in:
  - `INSERT INTO Student VALUES (NULL, 'John'); DROP TABLE Student; --', 'Smith')`
  - This command, alarmingly, deletes the `Student` table.
- To prevent SQL injection, always use **parameterized queries** when handling user inputs.

A Real-World Issue

- In the UK, a company was registered with the name: `;DROP TABLE "COMPANIES";--LTD`.
- This name is a deliberate example of an SQL injection attack:
    - The initial semicolon (`;`) ends the preceding SQL statement.
    - `DROP TABLE "COMPANIES"` instructs the database to delete a table named “COMPANIES”.
    - The second semicolon (`;`) marks the end of the SQL command.
    - `--` comments out the rest of the SQL statement, preventing errors.
- Due to the security implications, the company was required to change its name.



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` Certificate of Incorporation.

XKCD: Exploits of a Mom

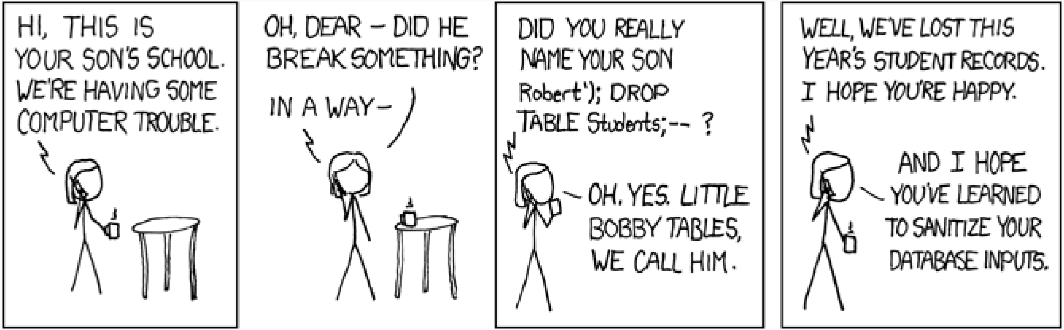

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

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

Handling Errors

- SQL commands can fail due to issues like:
  - Database inaccessibility (e.g., locked or unavailable).
  - SQL syntax errors.
  - Database constraint violations.
- To prevent disruptions, handle errors gracefully and avoid exposing raw error messages to users.
- Use Python's `try-except` structure for error handling:
  - `try:`
    - `# Attempt code execution`
  - `except ErrorType:`
    - `# Handle specific errors`
- Refer to function documentation to identify which errors to manage.

Example: Handling Errors

```python
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

Developing Dynamic Web Pages with Flask and SQLite

- Our final objective is to integrate Flask (as previously discussed) with SQLite to build a database-driven web application.
- This approach shifts from static to dynamic web pages, where data is sourced directly from a database. The key steps include:
  - Establishing a database connection using `sqlite3.connect()`.
  - Using a cursor object to execute SQL commands with `cursor.execute()`.
  - Linking query results to web page templates via `render_template("template.html", rows=rows)`.

Example: Database Driven Web Application (1/2)

Flask (`app.py`)

```python
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`**)

```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