SECURE CODING REVIEW

Objective: Perform a secure coding review for a chosen programming language and application, identify security vulnerabilities, and provide recommendations for secure coding practices.

Programming Language: Python

Application: Web-Based To-Do List Application

Code Review

Method: Manual Code Review

Vulnerability Identification:

1. **SQL Injection**
    a. The application constructs SQL queries by directly concatenating user input into the SQL query strings. This makes the application vulnerable to SQL injection attacks.

Vulnerable Code:

Here is a snippet of Lines 23 and 31 in python file.

```
conn.execute(f"INSERT INTO tasks (name) VALUES ('{task_name}')")
```

```
conn.execute(f"DELETE FROM tasks WHERE name = '{task_name}'")
```

2. **Cross-Site Scripting (XSS):**
    a. The *task_name* is rendered directly in the HTML output without proper escaping. This can lead to XSS attacks if a malicious script is injected into the task name.

Vulnerable Code:

Snippet of line 17 in index.html file.

```
<li>{{ task }} <a href="{{ url_for('delete_task', task_name=task) }}">Delete</a></li>
```

3. **Insecure Error Handling:**
    a. The application runs in debug mode (app.run(debug=True)), which can expose sensitive information in error messages if an exception occurs.

Vulnerable Code:

Snippet of line 36 and 37 in python file.

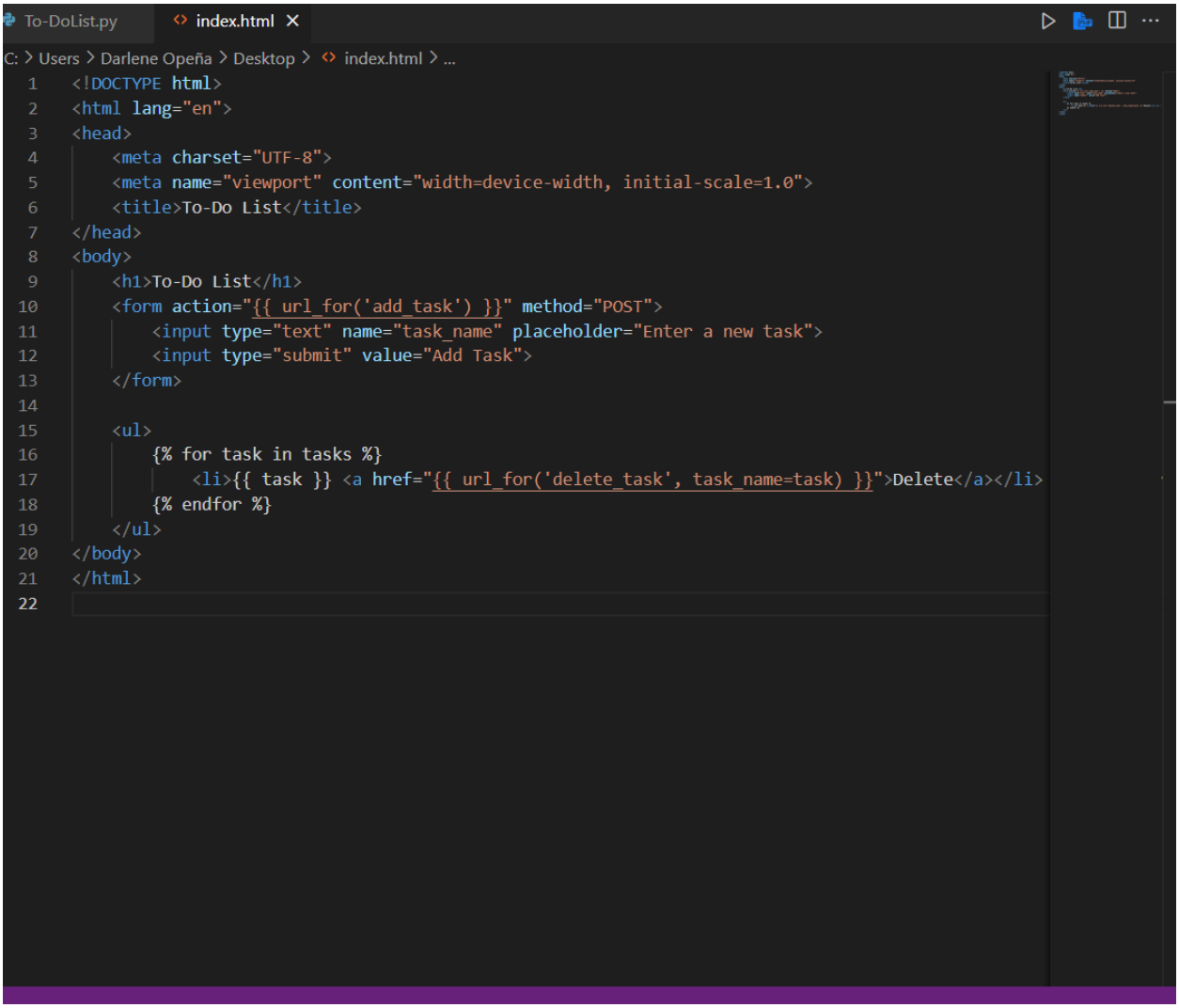
```
if __name__ == '__main__':
    app.run(debug=True)
```

Providing Recommendation

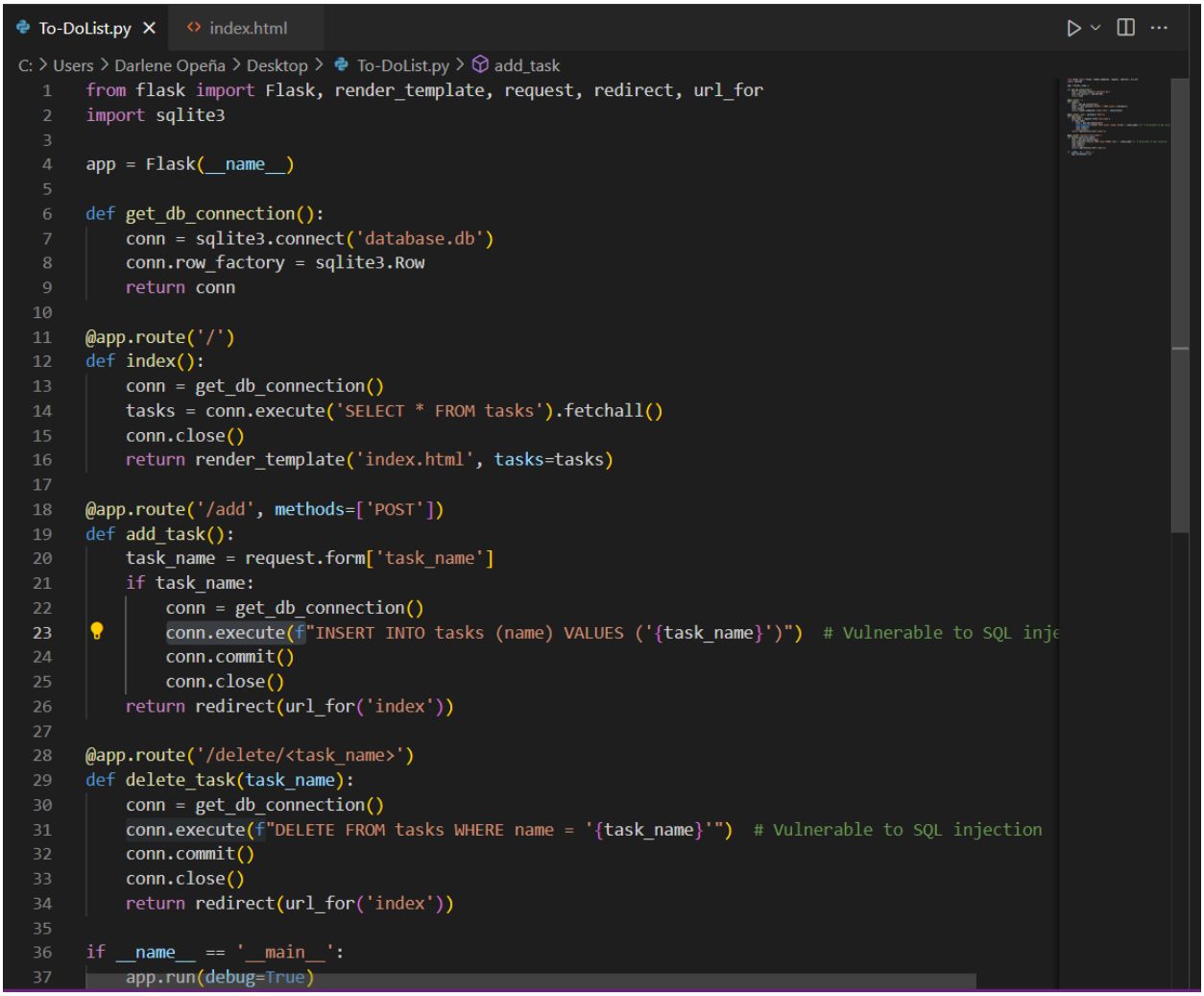
1. Use parameterized queries to avoid SQL injection vulnerabilities. Parameterized queries separate SQL code from data, preventing user input from being interpreted as executable code.
2. Ensure all user inputs are properly escaped before rendering them in the HTML output. Use Flask's built-in templating engine, Jinja2, which automatically escapes variables by default.
3. Disable debug mode in production environments to prevent the exposure of sensitive error details. Set the *debug* parameter to *False* or remove it entirely.

In conclusion, the secure coding review identified critical vulnerabilities in the application, including SQL injection, XSS, and insecure error handling. By applying the recommended secure coding practices, such as using parameterized queries, escaping user input, and disabling debug mode in production, the security of the application can be significantly improved.

TEMPLATE CODE:



APPLICATION CODE:



To-DoList.py | index.html

C: > Users > Darlene Opeña > Desktop > To-DoList.py > add_task

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

app = Flask(__name__)

def get_db_connection():
    conn = sqlite3.connect('database.db')
    conn.row_factory = sqlite3.Row
    return conn

@app.route('/')
def index():
    conn = get_db_connection()
    tasks = conn.execute('SELECT * FROM tasks').fetchall()
    conn.close()
    return render_template('index.html', tasks=tasks)

@app.route('/add', methods=['POST'])
def add_task():
    task_name = request.form['task_name']
    if task_name:
        conn = get_db_connection()
        conn.execute(f"INSERT INTO tasks (name) VALUES ('{task_name}')")  # Vulnerable to SQL inje
        conn.commit()
        conn.close()
    return redirect(url_for('index'))

@app.route('/delete/<task_name>')
def delete_task(task_name):
    conn = get_db_connection()
    conn.execute(f"DELETE FROM tasks WHERE name = '{task_name}'")  # Vulnerable to SQL injection
    conn.commit()
    conn.close()
    return redirect(url_for('index'))

if __name__ == '__main__':
    app.run(debug=True)
```