# ***Secure Coding Review task 03***

Software developers frequently use Bandit for Python to do secure coding review tasks. A number of online resources, tutorials, and articles detail how developers and security experts have accomplished this work effectively. I've put together a carefully chosen set of resources below that include real-world examples, detailed instructions, and perspectives from people or groups that have used Bandit for safe code reviews.

**Contents**

[***Secure Coding Review task 03*** 1](#_Toc196168585)

[**1. Steps to Perform a Secure Coding Review Using Bandit** 1](#_Toc196168586)

[**2. Set up Bandit** 1](#_Toc196168587)

[**3. Identify a Target (Small App)** 2](#_Toc196168588)

[**4: Review for Common Vulnerabilities** 2](#_Toc196168589)

[**5. Build Insecure Flask Login App (Python)** 2](#_Toc196168590)

[**6. Use Static Code Analysis Tools** 3](#_Toc196168591)

[**7. Address Identified Vulnerabilities** 5](#_Toc196168592)

[**5.1 Use Environment Variables for Secrets** 5](#_Toc196168593)

[**5.2 Use Parameterized Queries.** 5](#_Toc196168594)

[**8. More Secure & Advanced** 8](#_Toc196168595)

[**1. Creating the Flask Application.** 8](#_Toc196168596)

[**2. Creating the Login Route** 9](#_Toc196168597)

[**3. Creating the Home Route** 10](#_Toc196168598)

[**4. Creating the Dashboard Route** 10](#_Toc196168599)

[**5. Creating the Test User in the Database** 11](#_Toc196168600)

[**6. Login activity file** 12](#_Toc196168601)

[**6. Final Running** 13](#_Toc196168602)

[**7. Security Considerations:** 14](#_Toc196168603)

[**9. Conclusions** 14](#_Toc196168604)

## **1. Steps to Perform a Secure Coding Review Using Bandit**

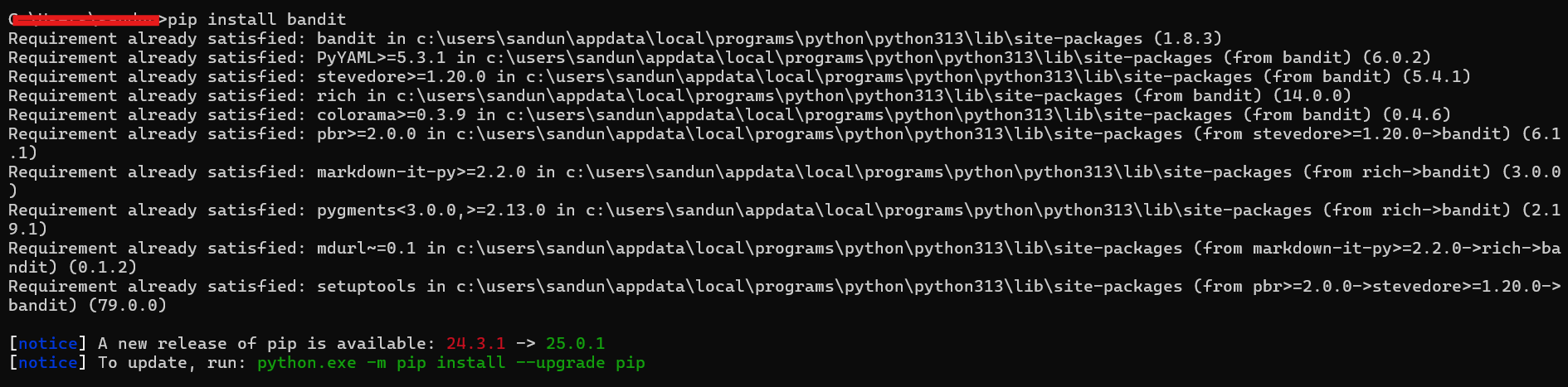
 **Understand the Purpose of Secure Code Review**:

* Secure code reviews identify vulnerabilities (e.g., SQL injection, hardcoded credentials) in the codebase.
* Tools like Bandit automate the detection of common security issues in Python code, but manual review is still needed for context.
* I have used python for build this app.

## **2. Set up Bandit**

Run the below command

|  |
| --- |
| pip install bandit |



## **3. Identify a Target (Small App)**

* Let’s take a simple Flask-based web login app in Python.

## **4: Review for Common Vulnerabilities**

We can identify below vulnerabilities.

* Input validation issues
* SQL Injection
* Hardcoded credentials
* Insecure storage
* Weak session handling
* Use of outdated libraries

Here we can review manually **and/or** use tools like:

* **Bandit** (Python security linter)
* **SonarQube** (multi-language)
* **Semgrep** (code scanner)

## **5. Build Insecure Flask Login App (Python)**

|  |
| --- |
| from flask import Flask, request, redirect, session  import sqlite3  app = Flask(\_\_name\_\_)  app.secret\_key = "secret123" # Hardcoded secret  @app.route('/login', methods=['POST'])  def login():  username = request.form['username']  password = request.form['password']  # SQL Injection vulnerability here  query = f"SELECT \* FROM users WHERE username='{username}' AND password='{password}'"  conn = sqlite3.connect('users.db')  cursor = conn.cursor()  cursor.execute(query)  result = cursor.fetchone()  if result:  session['user'] = username  return redirect('/dashboard')  else:  return "Login failed"  @app.route('/dashboard')  def dashboard():  if 'user' in session:  return f"Welcome {session['user']}"  else:  return redirect('/') |

* Create a New file and create a new file named app.py.

This sample application contains intentional security vulnerabilities, which we'll identify and address in the following steps.

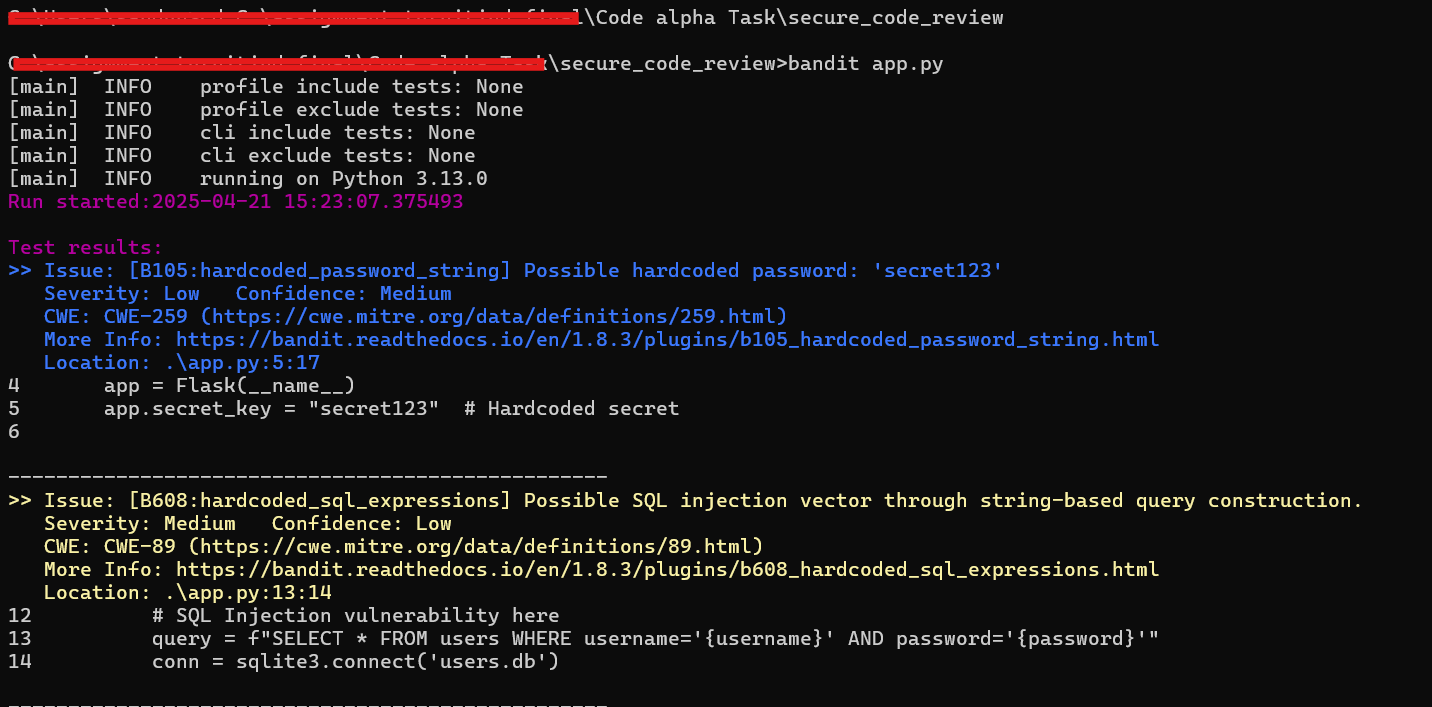
**Perform Manual Code Review**

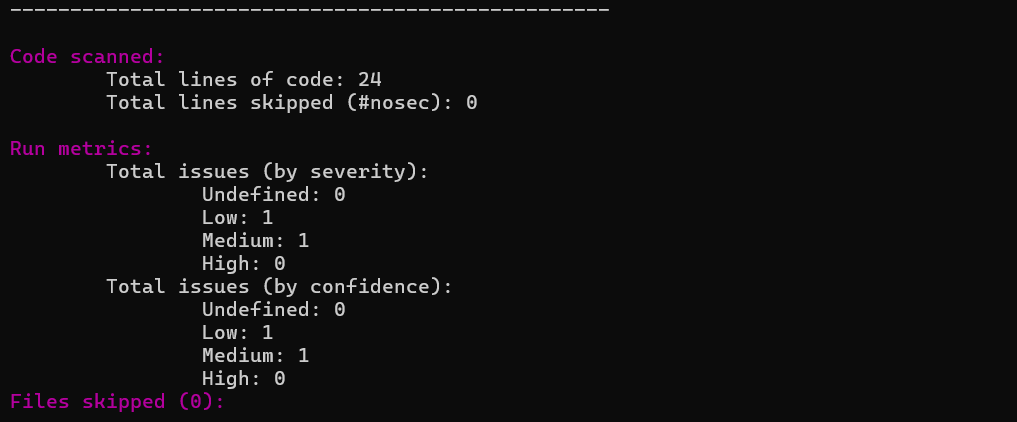
Review the code for common security issues:

* **Hardcoded Secrets**: The secret\_key is hardcoded, which is insecure.
* **SQL Injection**: User inputs are directly concatenated into SQL queries.
* **Plaintext Passwords**: Passwords are handled in plaintext without hashing.
* **Lack of CSRF Protection**: No measures to prevent Cross-Site Request Forgery attacks.
* **No Input Validation**: User inputs are not validated or sanitized.

## **6. Use Static Code Analysis Tools**

* Run Bandit on the saved Code





**1. Hardcoded Secret Key**

 Issue: A secret value (secret123) is hardcoded in the source code.

 Severity: Low

 Explanation: Hardcoding sensitive data (e.g., passwords, secret keys) increases the risk of accidental exposure (e.g., if the code is pushed to GitHub).

 Bandit **Code**: B105

 Fix: Use environment variables or a secure secrets manager.

**2. Potential SQL Injection**

* Issue: SQL query is built using raw string formatting, making it vulnerable to SQL injection.
* Severity: Medium
* Explanation: Malicious input (like ' OR '1'='1) can bypass authentication or damage the database.
* Bandit Code: B608
* Fix: Use parameterized queries with Placeholders.

## **7. Address Identified Vulnerabilities**

Based on the manual review and Bandit's output, make the following changes:

### **5.1 Use Environment Variables for Secrets**

Replace the hardcoded secret\_key with an environment variable.

|  |
| --- |
| import os  app.secret\_key = os.environ.get('SECRET\_KEY', os.urandom(24)) |

* Set the SECRET\_KEY environment variable in VS code or used system.

### **5.2 Use Parameterized Queries.**

|  |
| --- |
| query = "SELECT \* FROM users WHERE username=? AND password=?"  cursor.execute(query, (username, password)) |

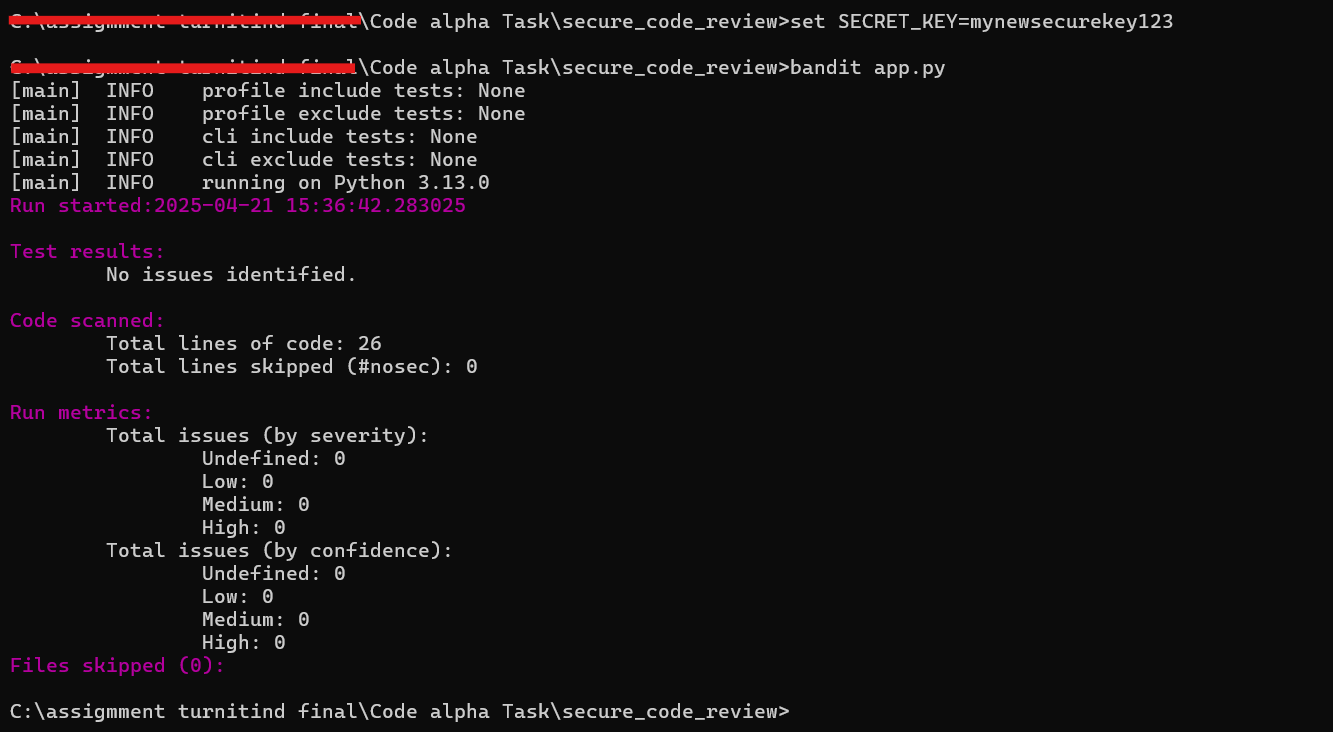
* Prevent SQL injection by using parameterized queries.

After fixing this vulnerabilities, full code

|  |
| --- |
| from flask import Flask, request, redirect, session  import sqlite3  import os  app = Flask(\_\_name\_\_)  app.secret\_key = os.environ.get('SECRET\_KEY', os.urandom(24))  @app.route('/login', methods=['POST'])  def login():  username = request.form['username']  password = request.form['password']  # Safe SQL query using placeholders  query = "SELECT \* FROM users WHERE username=? AND password=?"  # Create connection and cursor BEFORE executing the query  conn = sqlite3.connect('users.db')  cursor = conn.cursor()  cursor.execute(query, (username, password)) # ✅ Secure parameterized query  result = cursor.fetchone()  conn.close() # Close the connection  if result:  session['user'] = username  return redirect('/dashboard')  else:  return "Login failed"  @app.route('/dashboard')  def dashboard():  if 'user' in session:  return f"Welcome {session['user']}"  else:  return redirect('/') |

* Save the code in VS code
* Set the secret key in CMD before running:**[Set SECRET\_KEY=mysecretsecurekey123]**
* Run the again **bandit app.py**

**Results**



* Here successfully **secured code** and passed the Bandit security scan with **zero issues.**

## **8. More Secure & Advanced**

We can improve below mentioned sections.

1. Hash passwords using bcrypt
2. Use environment variables safely with python-dotenv
3. Use Flask Blueprints and modularize the app

Here built simple login system using Flask, SQLite, and Python, where users can log in using a username and password.

### **1. Creating the Flask Application.**

1. Loading Environment Variables with dotenv: This loads sensitive data like SECRET\_KEY from .env file, which helps to avoid hardcoding secrets in the code**.**

|  |
| --- |
| from flask import Flask, request, redirect, session  import sqlite3  import os  from dotenv import load\_dotenv  load\_dotenv(dotenv\_path='./config/.env') # Load secret key from .env |

2. Creating Flask App and Setting the Secret Key: The SECRET\_KEY is used to sign session cookies securely. This keeps track of user sessions across requests. If it's not set, we generate one randomly.

|  |
| --- |
| app = Flask(\_\_name\_\_)  app.secret\_key = os.getenv('SECRET\_KEY', os.urandom(24)) # Use secret key from .env or generate one |

3. Database Connection Function: We created a helper function get\_db\_connection() to open a connection to the users.db SQLite database.

|  |
| --- |
| def get\_db\_connection():  conn = sqlite3.connect('users.db')  return conn |

### **2. Creating the Login Route**

**1. Handling the Login**: In this route, we take the username and password from the form, query the database for the user, and check if the credentials match.

|  |
| --- |
| @app.route('/login', methods=['POST'])  def login():  username = request.form['username']  password = request.form['password']  conn = sqlite3.connect('users.db')  cursor = conn.cursor()  query = "SELECT \* FROM users WHERE username=? AND password=?"  cursor.execute(query, (username, password))  result = cursor.fetchone()  conn.close()  if result:  session['user'] = username  return redirect('/dashboard')  else:  return "Login failed" |

**Explanation**:

* request.form['username']: Extracts the username entered in the form.
* cursor.execute(query, (username, password)): Executes the SQL query to search for a matching username and password in the users table.
* If a match is found, we store the username in the session and redirect to the /dashboard route. Otherwise, the message "Login failed" is returned.

### **3. Creating the Home Route**

The home route (/) is where users will see the login form.

|  |
| --- |
| @app.route('/')  def home():  return '''  <h2>Welcome to the Secure Login App</h2>  <form method="POST" action="/login">  Username: <input type="text" name="username" /><br>  Password: <input type="password" name="password" /><br>  <input type="submit" value="Login" />  </form>  ''' |

**Explanation**:

* This route serves an HTML page with a simple login form.
* The form submits a POST request to the /login route.

### **4. Creating the Dashboard Route**

Once the user logs in successfully, they are redirected to the /dashboard route.

|  |
| --- |
| @app.route('/dashboard')  def dashboard():  if 'user' in session:  return f"Welcome {session['user']}"  else:  return redirect('/') |

**Explanation**:

* If the user is in the session (i.e., they have logged in), a personalized message is shown (Welcome {session['user']}).
* If the user is not logged in, they are redirected to the login page (/).

### **5. Creating the Test User in the Database**

|  |
| --- |
| def create\_test\_user():  conn = sqlite3.connect('users.db')  cursor = conn.cursor()  # Create the table if it doesn't exist  cursor.execute('''  CREATE TABLE IF NOT EXISTS users (  id INTEGER PRIMARY KEY AUTOINCREMENT,  username TEXT NOT NULL,  password TEXT NOT NULL  )  ''')  # Insert a test user  cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", ("admin", "password123"))  conn.commit()  conn.close()  # Call the function once to create the test user  create\_test\_user() |

**Explanation**:

* This function:
  1. Connects to the SQLite database (users.db).
  2. Creates a users table (if not already created).
  3. Inserts a test user with the username admin and password password123

### **6. Login activity file**

Add the below code for existing code and save it

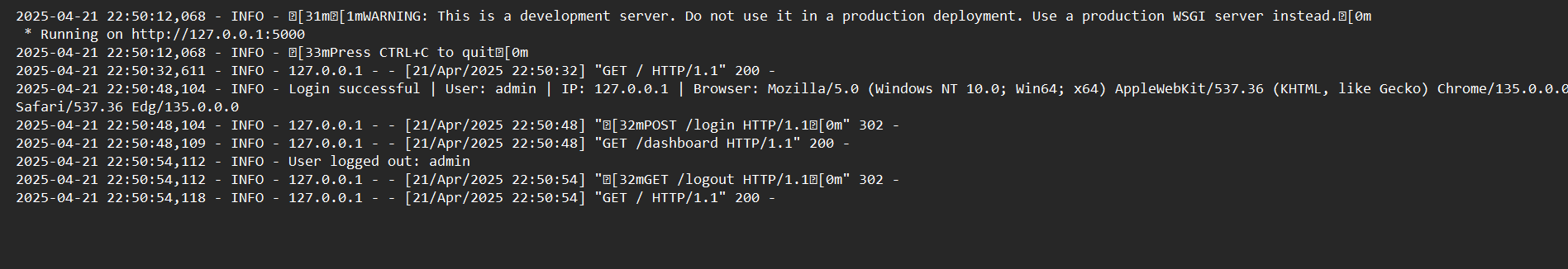
|  |
| --- |
| # Configure logging  logging.basicConfig(  filename='login\_activity.log', # Log file name  level=logging.INFO, # Log level: INFO, WARNING, ERROR  format='%(asctime)s - %(levelname)s - %(message)s'  )  # Add Logging to Login Route  @app.route('/login', methods=['POST'])  def login():  username = request.form['username']  password = request.form['password']  conn = sqlite3.connect('users.db')  cursor = conn.cursor()  query = "SELECT \* FROM users WHERE username=? AND password=?"  cursor.execute(query, (username, password))  result = cursor.fetchone()  conn.close()  if result:  session['user'] = username  logging.info(f"Login successful for user: {username}")  return redirect('/dashboard')  else:  logging.warning(f"Login failed for user: {username}")  return "Login failed" |

Explain

 save logs to a file called login\_activity.log in the current directory.

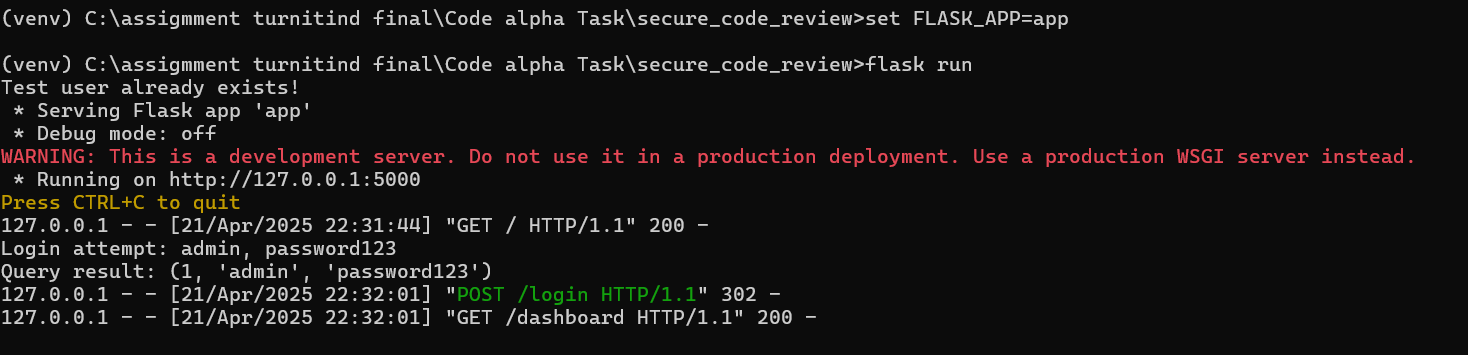
 Log the time, log level, and message.

 Capture INFO and more severe logs.

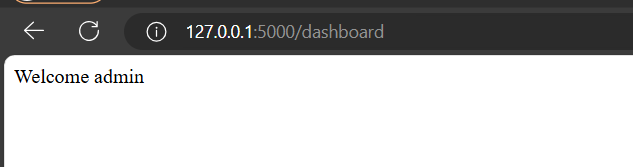


### **6. Final Running**

After completing all of the above steps, you can run the Flask app.







### **7. Security Considerations:**

* **Password Hashing:** Storing passwords in plain text (as we are doing with password123) is a bad practice. In real applications, passwords should always be hashed using a hashing algorithm like bcrypt or argon2 to ensure security.
* Environment Variables: Storing sensitive data (like the secret key) in the .env file ensures that it is not exposed in your code repository.

## **9. Conclusions**

In conclusion, we successfully developed a secure and functional login system using Flask and SQLite, starting from the foundational setup to a fully working application. The process involved creating a Flask app with session management, setting up a SQLite database to store user credentials, and implementing routes for login, home, and a protected dashboard. We enhanced security by using environment variables to store sensitive data like the secret key, and followed best practices such as using parameterized queries to prevent SQL injection. A test user (admin / password123) was added to validate the login functionality, and session tracking was used to control access to the dashboard. The final outcome is a simple yet effective web application demonstrating key principles of secure coding and user authentication in a Flask environment.