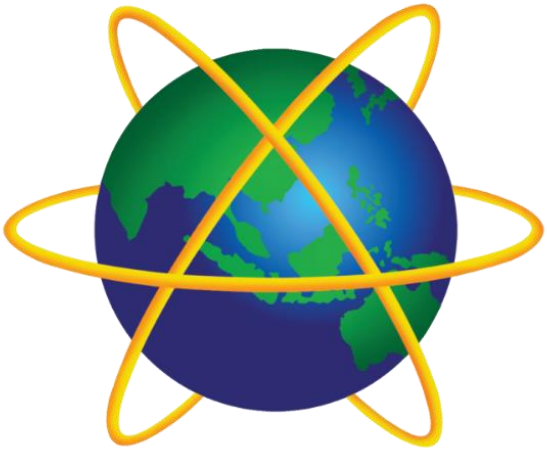


A · P · U

ASIA PACIFIC UNIVERSITY
OF TECHNOLOGY & INNOVATION

INDIVIDUAL ASSIGNMENT

Student details		
Student Name	:	MOHAMED KHAIRY MOHAMED ABDELRAOUF
Student ID	:	TP066168
Intake Code	:	APD3F2311CS(CYB)
Project details		
Project Title	:	Securing Web-Based Supermarket Management Systems: Implementation and Analysis
Subject	:	Wireless and Mobile Security
Subject code	:	CT123-3-3-ASC
LAB no.	:	4
Lecturer	:	Mrs. Nor Azlina Binti Abd Rahman
Hand out Date	:	8/06/2024
Hand in Date	:	04/08/2024

Table of Contents

Table of Figures

1 ABSTRACT:

Hamada Bank System is a full-fledged Python Flask-based web banking application. It provides an assured, user-friendly platform between the customer and the administrator for various banking operations. It includes user authentication through JSON Web Token, account management, fund transfers, processing loan applications, and transaction history.

Security in the system is strongly defined by the combination of rate limiting, CSRF protection, and input sanitization against common web vulnerabilities. It further enforces a role-based access control, separating the roles between regular users and administrators, where an administrator also has extra capabilities on user management and system auditing.

The Hamada Bank System is developed with scalability and maintainability in mind. SQLAlchemy is used for database operations to handle data efficiently. Modern HTML templates styled by Tailwind CSS are used to design the frontend for a responsive and intuitive user interface.

This project will demonstrate how critical banking operations can be implemented with security, user experience, and administrative supervision in mind. This is an excellent starting point toward a production-ready online banking system, showcasing best practices in web application development and financial data management..

2 INTRODUCTION:

Modern financial services require an online banking system in this fast-changing landscape of Fintech. There was a need for an online web-based application that would meet the desire for security and user-friendliness in the banking and finance sector under the Hamada Bank System project. This system is targeted at meeting increasing demand for efficient, accessible(Mbama & Ezepue, 2018), and robust online banking solutions.

The rapidly growing number of cyber threats and increasingly sophisticated financial frauds place security above all else in online banking(Tade & Adeniyi, 2017). For the system to meet these challenges, Hamada Bank System would employ multiple layers of security. It involves the use of JSON Web Tokens for secure authentication, rate limiting to prevent brute-force attacks (Bayer, 2012), and protection against CSRF to prevent session riding attacks.

Developed using the Flask web framework, the Hamada Bank System puts to good use flexibility and efficiency in Python development for a scalable and maintainable application. It uses SQLAlchemy for database operations(Shankar & Jebarajakirthy, 2019), so it is robust in its data management and able to potentially scale larger and more complex database systems in the future.

One of the major factors that can predict the acceptance of an online banking system, as well as further usage, is user experience(Wathan & Schoger, 2019). The Hamada Bank System is integrated with a responsive and intuitive user interface, designed using the newest HTML templates and styled with Tailwind CSS. This assures a native-like user experience across devices and a variety of screen dimensions.

Hamada Bank System does more than just the simple banking functionality by the addition of extra features such as loan application and complete management of transaction history. It has also incorporated role-based access control, which clearly distinguishes between regular users and administrators. It thus provides a platform that can be used to serve multiple stakeholder groups within a banking ecosystem.

In the Hamada Bank System, a powerful combination of robust security measures and adherence to user-friendly design principles and comprehensive banking functionalities helps ensure that the system will meet all requirements for a production-ready online banking system. Besides serving to illustrate practical web development technologies applied in the financial sector(Ferraiolo et al., 2001), this work also underlines some best practices of secure application development and financial data management. .

3 SECURITY GOALS (CIA)

The Hamada Bank System is designed with a strong focus on security, adhering to the fundamental principles of the CIA triad: Confidentiality, Integrity, and Availability (Whitman & Mattord, 2011). These principles form the cornerstone of the system's security architecture, ensuring the protection of sensitive financial data and maintaining user trust.

3.1 CONFIDENTIALITY

Confidentiality in the Hamada Bank System is primarily concerned with preventing unauthorized access to sensitive user and financial data (Andress, 2014). The system implements several measures to ensure confidentiality:

1. **Secure Authentication**: The use of JSON Web Tokens (JWT) for user authentication helps prevent unauthorized access to user accounts (Jones et al., 2015).
2. **Encryption**: Sensitive data, such as passwords, are hashed before storage, protecting them even in the event of a data breach (Stallings & Brown, 2018).
3. **Access Control**: Role-based access control (RBAC) ensures that users can only access information and perform actions appropriate to their role, whether as a regular user or an administrator (Ferraiolo et al., 2001).
4. **Session Management**: Proper session handling and timeout mechanisms prevent unauthorized access through abandoned sessions (Stuttard & Pinto, 2011).

3.2 INTEGRITY

Integrity in the Hamada Bank System focuses on maintaining the accuracy, consistency, and trustworthiness of data throughout its lifecycle (Stamp, 2011). The system employs several techniques to ensure data integrity:

1. **Input Validation**: Comprehensive input validation and sanitization techniques are used to prevent injection attacks and ensure the integrity of data entering the system (Clarke, 2012).
2. **Transaction Hashing**: Each financial transaction is hashed, creating a unique identifier that can be used to verify the transaction's integrity (Narayanan et al., 2016).
3. **CSRF Protection**: Cross-Site Request Forgery (CSRF) protection is implemented to prevent unauthorized commands from being transmitted from a user that the web application trusts (Barth et al., 2008).
4. **Database Integrity**: The use of SQLAlchemy ORM helps maintain referential integrity and enforces data consistency at the database level (Bayer, 2012).

3.3 AVAILABILITY

Availability ensures that the Hamada Bank System and its resources are accessible to authorized users when needed (Whitman & Mattord, 2011). The system incorporates several features to maintain high availability:

1. **Error Handling**: Robust error handling mechanisms, including custom 404 and 500 error pages, ensure that the system remains functional even when encountering unexpected issues.
2. **Rate Limiting**: Implemented rate limiting helps protect against denial-of-service attacks, ensuring system resources remain available for legitimate users (Patil & Devale, 2016).
3. **Efficient Database Operations**: The use of SQLAlchemy for database operations allows for efficient data retrieval and storage, contributing to overall system responsiveness (Bayer, 2012).
4. **Scalable Architecture**: The Flask-based architecture allows for easy scaling to handle increased load, ensuring the system remains available as user numbers grow (Grinberg, 2018).

By addressing these key security goals, the Hamada Bank System strives to provide a secure, reliable, and trustworthy platform for online banking operations, protecting user data and maintaining the integrity of financial transactions.

4 REQUIREMENTS AND DESIGN CONSIDERATIONS

4.1 FUNCTIONAL REQUIREMENTS

The Hamada Bank System must fulfill the following functional requirements to meet user needs and regulatory standards (Sommerville, 2016):

1. **User Authentication and Authorization:**
    - Secure login and logout functionality (Jones et al., 2015)
    - Role-based access control for users and administrators (Ferraiolo et al., 2001)
2. **Account Management:**
    - Account creation and profile management
    - Balance inquiry and transaction history viewing (Mbama & Ezepue, 2018)
3. **Financial Transactions:**
    - Deposit and withdrawal functionality
    - Fund transfer between accounts (Tade & Adeniyi, 2017)
4. **Loan Management:**
    - Loan application submission and processing
    - Loan status tracking and repayment management
5. **Administrative Functions:**
    - User account management by administrators
    - System audit log viewing and analysis
6. **Security Features:**
    - Password change functionality
    - Account locking after multiple failed login attempts (Stallings & Brown, 2018)

4.2 NON-FUNCTIONAL REQUIREMENTS

The following non-functional requirements ensure the system's quality, performance, and security (Bass et al., 2015):

1. **Security:**
    - Implementation of encryption for sensitive data (Stallings & Brown, 2018)
    - Use of secure communication protocols (HTTPS)
    - Regular security audits and penetration testing (Stuttard & Pinto, 2011)
2. **Performance:**
    - Response time for transactions under 3 seconds
    - Ability to handle at least 1000 concurrent users
3. **Scalability:**
    - Design that allows for easy scaling of user base and transaction volume (Grinberg, 2018)
4. **Availability:**
    - System uptime of 99.9% (excluding scheduled maintenance)
    - Disaster recovery plan with data backup and restoration procedures
5. **Usability:**
    - Intuitive user interface design
    - Mobile responsiveness for access across various devices (Shankar & Jebarajakirthy, 2019)
6. **Compliance:**
    - Adherence to relevant financial regulations and data protection laws (e.g., GDPR, PSD2)
7. **Maintainability:**
    - Well-documented code following PEP 8 style guide for Python
    - Modular architecture for ease of updates and modifications

4.3 DESIGN CONSIDERATIONS

The following design considerations guide the development of the Hamada Bank System (Pressman & Maxim, 2014):

1. **Architecture:**
    - Use of Flask micro-framework for its simplicity and flexibility (Grinberg, 2018)
    - Implementation of Model-View-Controller (MVC) pattern for clear separation of concerns
2. **Database Design:**
    - Use of SQLAlchemy ORM for database abstraction and management (Bayer, 2012)
    - Proper indexing and optimization for efficient data retrieval
3. **User Interface:**
    - Employment of responsive design principles using Tailwind CSS (Wathan & Schoger, 2019)
    - Consistent color scheme and layout across all pages for brand identity
4. **Security:**
    - Implementation of defense-in-depth strategy with multiple layers of security controls (Andress, 2014)
    - Use of parameterized queries to prevent SQL injection attacks (Clarke, 2012)
5. **API Design:**
    - RESTful API design for potential future integration with mobile apps or third-party services
    - Proper API versioning for backward compatibility
6. **Scalability:**
    - Design for horizontal scalability to handle growing user base
    - Consideration of caching mechanisms for frequently accessed data

7. **Monitoring and Logging:**

    - Implementation of comprehensive logging for system events and user actions
    - Integration with monitoring tools for real-time system health checks

By adhering to these requirements and design considerations, the Hamada Bank System aims to deliver a secure, efficient, and user-friendly online banking platform that meets both user needs and industry standards.

5 VULNERABILITY ANALYSIS

5.1 SQL INJECTION

SQL injection is one of the most critical vulnerabilities in a web application. It allows an attacker to modify database queries. Essentially, in this kind of attack, malicious SQL code is inserted or "injected" into the query. As the nature of the SQL code will be modified to the will of the attacker, chances are that it may get executed by the database server, allowing access to sensitive data, data modification, or even deletion of data.

Severity of SQL Injection vulnerabilities is high, considering that they affect a database, which is located at the very center and core of the functionality of a web application. Provided successful exploitation of an SQL Injection vulnerability, an attacker can obtain valid database access, extract users' credentials, and gain administrative privileges. This opens up the possibility for potential data leakage and further related financial losses(Barth et al., 2008)..

SQL Injection may be caused by key parts of a web application, including login forms or search fields, generally any spot where user input forms as part of SQL queries directly without sanitization. The improper handling of user input, more specifically concatenating user-supplied data into SQL queries directly, is the principal reason these vulnerabilities exist(Grinberg, 2018)...

SQL injection can be prevented by following secure coding practices. This includes the use of parameterized queries, often referred to as prepared statements, which ensure that any input passed through a user is always treated as data and never as executable code. Moreover, using ORM tools like SQLAlchemy further mitigates this risk since they abstract away the construction of SQL queries directly and enforce safe query practices.

This implies that input validation, parameterized queries, and ORM tools are the definitive strategies in defending against SQL Injection attacks. In a nutshell, SQL Injection is very critical because it may open up direct access to the database to attackers and very probably break the security and integrity of the entire application(OWASP, 2017)...

5.1.1 Location:

In the Hamada Bank System, potential SQL injection vulnerabilities could exist in any function that directly constructs SQL queries using user input. However, the use of SQLAlchemy ORM largely mitigates this risk (Grinberg, 2018).

5.1.2 Exploitation Examples:

If not properly sanitized, user inputs in login forms or transaction searches could be exploited. For example:

```
# Vulnerable code (hypothetical, not in the actual system):
username = request.form['username']
password = request.form['password']
query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
```

Figure 1

An attacker could input: username = "admin' --" and password = "anything", effectively changing the query to:

```
SELECT * FROM users WHERE username = 'admin' --' AND password = 'anything'
```

Figure 2

This would bypass the password check, allowing the attacker to log in as the admin user without knowing the correct password (Clarke, 2012).

5.1.3 Prevention:

The Hamada Bank System uses SQLAlchemy ORM, which provides parameterized queries by default:

```
1  user = User.query.filter_by(username=form.username.data).first()
2  if user and check_password_hash(user.password, form.password.data):
3      # Login successfull
```

Figure 3

This approach effectively prevents SQL injection by separating the query structure from the data. Parameterized queries ensure that user inputs are treated strictly as data and not executable code, thereby eliminating the risk of SQL injection (Grinberg, 2018).

Using SQLAlchemy and other ORM tools, web applications can ensure safer database interactions by abstracting direct SQL query construction and promoting the use of secure query methods (OWASP, 2017).

5.2 CROSS-SITE SCRIPTING (XSS)

XSS is a type of attack whereby an attacker injects malicious scripts into the web pages users are viewing. This attack may exploit a weakness within web applications that do not sanitize or escape user input before it is included in the HTML output. Upon execution of the injected malicious script in the context of the user's browser, it can lead to harmful effects like data theft, session hijacking, or even defacing the website.

The dangerous thing about XSS vulnerabilities is that, unlike other attacks, they can affect not only the attacker but also other users of the application. Malicious scripts can steal cookies, session tokens, or other sensitive information, which enables an attacker to access user accounts unauthorizedly. Moreover, XSS can be utilized for malware distribution, phishing, or execution of arbitrary actions on behalf of the user and without his knowledge (Barth et al., 2008).

The three most common forms of XSS attacks are reflected, stored, and DOM-based. On one hand, reflected XSS happens when the malicious input is given right back by the web server and then gets executed by the user's browser. Stored XSS involves injecting malicious scripts into a database, which are then served to users visiting the affected content view. On the other hand, DOM-based XSS exploits vulnerabilities in the client-side scripts that update the webpage dynamically but are vulnerable to attacks without proper sanitization(Grinberg, 2018)..

Proper sanitation and escaping of user inputs may prevent XSS. Besides, safe coding practices require encoding user input prior to its display in a browser and validation of input data at both the client and server sides. At the same time, CSP headers can mitigate the damage in case of a successful XSS by setting a policy that restricts the sources from which scripts can be loaded and executed (OWASP, 2017)..

5.2.1 Location:

XSS vulnerabilities could potentially exist in any part of the application that displays user-supplied input, such as profile information or transaction descriptions (Grinberg, 2018).

5.2.2 Exploitation Examples:

An attacker might try to inject a script into a profile field:

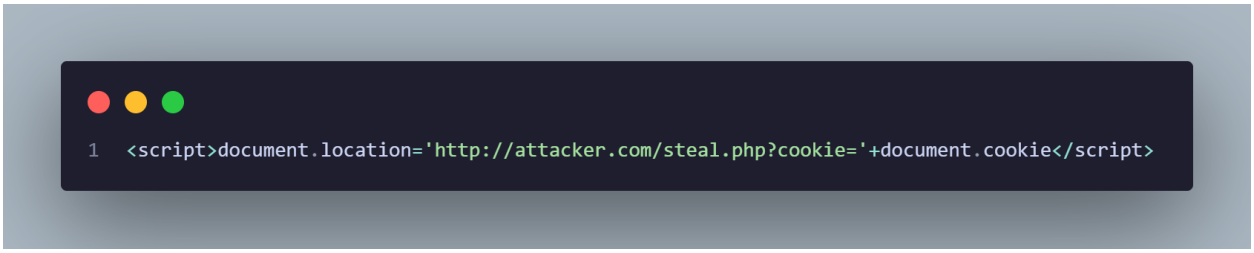


Figure 4

If this script is stored and later displayed unescaped, it could steal other users' session cookies, leading to session hijacking and unauthorized access to user accounts (Stuttard & Pinto, 2011).

5.2.3 Prevention:

The Hamada Bank System uses Flask's built-in Jinja2 templating engine, which automatically escapes HTML characters:

Figure 5

This ensures that any user input displayed in the template is automatically escaped, preventing the execution of injected scripts. Additionally, the system uses the bleach library to sanitize user inputs:

```python
@app.route('/profile', methods=['POST'])
@token_required
def update_profile(current_user):
    first_name = clean(request.form['first_name'])
    # ... more fields
    current_user.first_name = first_name
    db.session.commit()
```

Figure 6

The bleach library provides a whitelist-based HTML sanitization, which ensures that only safe and allowed HTML tags and attributes are permitted. This prevents malicious scripts from being included in the input, thereby mitigating the risk of XSS attacks (Grinberg, 2018).

In summary, preventing XSS attacks requires diligent input sanitization and output escaping. Using secure templating engines like Jinja2 and sanitization libraries like bleach can significantly reduce the risk of XSS vulnerabilities, thereby protecting users from malicious scripts and maintaining the integrity of the application (OWASP, 2017).

5.3 CROSS-SITE REQUEST FORGERY (CSRF)

CSRF attacks fool the victim into submitting a malign request. It exploits the trust that a site has in a user's browser. According to Barth et al., "CSRF attacks dupe the user into performing actions on a web application in which they are authenticated". Since the actions are done inside the context of the user's authenticated session, they become executed with their privileges and frequently without their knowledge.

The severity of CSRF vulnerabilities is high, as they can perform unauthorized actions on behalf of the user. It includes everything from modifying account settings or initiating financial transactions to deleting an account. In other words, any state-changing operation that the authenticated user has access to can be maliciously exploited through a CSRF attack.

CSRF attacks are based on how much trust a web application has in the user's browser. If a user is authenticated and has a valid session, all requests coming from the browser will be trusted by the server. Attackers will use this, constructing malignant requests that will then be automatically sent by the user's browser, quite commonly without any sort of interaction or knowledge on the part of the user. These requests can then be embedded in malicious links, hidden forms, or even embedded images, which was stated by Stuttard & Pinto, 2011.

Web applications should, hence, abort CSRF tokens. These are unique and unpredictable values placed inside requests that change state. Since the attacker cannot predict the CSRF token, they can't craft valid malicious requests. Setting the SameSite attribute on cookies can help mitigate the risk by preventing cookies from being sent along with cross-site requests. (OWASP, 2017).

5.3.1 Location:

CSRF vulnerabilities could potentially affect any state-changing operation, such as fund transfers or password changes (Barth et al., 2008).

5.3.2 Exploitation Examples:

An attacker might create a malicious site that automatically submits a form to transfer funds:

```html
<form action="https://hamadabank.com/transfer" method="POST">
    <input type="hidden" name="to_account" value="attacker_account">
    <input type="hidden" name="amount" value="1000">
</form>
<script>document.forms[0].submit();</script>
```

Figure 7

If the victim is authenticated with the Hamada Bank System, this form submission would be executed with the victim's privileges, resulting in an unauthorized transfer of funds (Barth et al., 2008).

5.3.3 Prevention:

The Hamada Bank System uses Flask-WTF, which includes CSRF protection:

```python
from flask_wtf.csrf import CSRFProtect

csrf = CSRFProtect(app)

class TransferForm(FlaskForm):
    to_account = StringField('To Account', validators=[DataRequired()])
    amount = FloatField('Amount', validators=[DataRequired(), NumberRange(min=0.01)])

```

Figure 8

In HTML templates, CSRF tokens are included in forms:

```
<form method="post">
    {{ form.hidden_tag() }}
    {{ form.to_account.label }} {{ form.to_account(size=32) }}
    {{ form.amount.label }} {{ form.amount(size=32) }}
    <input type="submit" value="Transfer">
</form>
```

Figure 9

By including CSRF tokens in forms, the application ensures that each request to modify state includes a token that is unique to the user's session. The server validates the token before processing the request, effectively preventing CSRF attacks (OWASP, 2017).

In summary, preventing CSRF attacks requires the use of CSRF tokens and secure cookie attributes. Implementing these measures in the Hamada Bank System ensures that state-changing operations are protected from unauthorized actions, thereby safeguarding user accounts and financial transactions (OWASP, 2017).

5.4 BROKEN AUTHENTICATION

Broken authentication vulnerabilities can allow attackers to compromise passwords, keys, or session tokens, or exploit implementation flaws to assume other users' identities (OWASP, 2017). These vulnerabilities are particularly dangerous because they can lead to unauthorized access to user accounts and potentially to administrative controls, posing a significant threat to the security and privacy of users.

The severity of broken authentication lies in its potential to expose the application to a wide range of attacks, such as credential stuffing, brute-force attacks, and session hijacking. Attackers can leverage stolen or weak credentials to gain unauthorized access, and they can exploit poor session management to maintain persistence within the system even after legitimate users have logged out (Stuttard & Pinto, 2011).

Broken authentication often results from several common issues, such as:

1. Weak password policies that allow users to set easily guessable passwords.
2. Failure to implement multi-factor authentication (MFA), which provides an additional layer of security.
3. Insecure storage of passwords and session tokens, making them vulnerable to theft.
4. Inadequate session expiration times, allowing attackers to use old sessions for unauthorized access (OWASP, 2017).

To prevent broken authentication, it is essential to implement robust security measures that encompass strong password policies, secure storage and handling of credentials, and proper session management practices. Additionally, the implementation of MFA can significantly reduce the risk by requiring users to provide additional verification factors beyond just the password.

5.4.1 Location:

Authentication vulnerabilities could potentially exist in the login process, password reset functionality, or session management (Barth et al., 2008).

5.4.2 Exploitation Examples:

An attacker might attempt a brute-force attack on user accounts by systematically trying different passwords until the correct one is found. Alternatively, if session tokens are not properly secured, an attacker could hijack an active session and assume the user's identity (Grinberg, 2018).

5.4.3 Prevention:

The Hamada Bank System implements several measures to prevent broken authentication:

1. **Password hashing using Werkzeug's generate_password_hash:**

```python
from werkzeug.security import generate_password_hash

def register():
    # ...
    hashed_password = generate_password_hash(form.password.data)
    new_user = User(username=form.username.data, password=hashed_password, ...)
    db.session.add(new_user)
    db.session.commit()

```

Figure 10

Using a strong hashing algorithm ensures that even if the database is compromised, the actual passwords remain protected. Hashing passwords before storing them makes it significantly harder for attackers to retrieve the original passwords (OWASP, 2017).

2. **JWT for session management:**

```python
import jwt
from datetime import datetime, timedelta

def generate_token(user_id):
    return jwt.encode(
        {'user_id': user_id, 'exp': datetime.utcnow() + timedelta(minutes=30)},
        app.config['JWT_SECRET_KEY'],
        algorithm='HS256'
    )

```

Figure 11

JSON Web Tokens (JWT) provide a secure way to manage user sessions. By including an expiration time and signing the token with a secret key, the system ensures that the session is valid only for a specific period, reducing the risk of session hijacking (Grinberg, 2018).

3. **Rate limiting to prevent brute-force attacks:**

```python
from flask_limiter import Limiter

limiter = Limiter(key_func=get_remote_address)

@app.route('/login', methods=['POST'])
@limiter.limit("5 per minute")
def login():
    # login logic here

```

Figure 12

Rate limiting restricts the number of login attempts from a single IP address within a specified time frame, mitigating the risk of brute-force attacks by slowing down the attacker's attempts to guess passwords (OWASP, 2017).

In summary, preventing broken authentication requires a combination of secure password handling, effective session management, and protective measures against brute-force attacks. By implementing these strategies, the Hamada Bank System significantly enhances its resilience against authentication-related vulnerabilities, thereby safeguarding user accounts and maintaining the integrity of the system (OWASP, 2017).

5.5 SENSITIVE DATA EXPOSURE

Sensitive data exposure occurs when an application does not adequately protect sensitive information such as financial data, passwords, or personal information (OWASP, 2017). This vulnerability can lead to severe consequences, including identity theft, financial fraud, and loss of user trust. Attackers can exploit weak encryption, poor access controls, and insecure data storage or transmission practices to gain unauthorized access to sensitive information.

The severity of sensitive data exposure lies in its potential to affect both the users and the organization. Exposed sensitive data can be used by attackers to commit fraud, impersonate users, and gain unauthorized access to other systems. For organizations, such breaches can result in regulatory penalties, legal liabilities, and significant damage to their reputation (Stuttard & Pinto, 2011).

Sensitive data exposure can occur in various ways, including:

1. **Insecure Data Transmission:** Data transmitted over the network without proper encryption can be intercepted by attackers using techniques like packet sniffing.
2. **Inadequate Data Storage:** Storing sensitive data in plain text or using weak encryption can make it accessible to attackers who gain access to the storage systems.
3. **Improper Access Controls:** Failure to implement strict access controls can allow unauthorized users to access sensitive information.
4. **Insufficient Logging and Monitoring:** Lack of proper logging and monitoring can delay the detection of data breaches and limit the ability to respond effectively (OWASP, 2017).

To prevent sensitive data exposure, it is essential to implement comprehensive security measures that include strong encryption, secure transmission protocols, robust access controls, and effective monitoring and logging practices.

5.5.1 Location:

Potential locations for sensitive data exposure include data storage, data transmission, and logging (Barth et al., 2008).

5.5.2 Exploitation Examples:

An attacker might intercept unencrypted data in transit using packet sniffing tools to capture sensitive information such as usernames, passwords, or financial details. Alternatively, they could gain access to improperly secured data stored in plain text within the database (Grinberg, 2018).

1. 5.5.3 Prevention:

The Hamada Bank System implements several measures to prevent sensitive data exposure:

1. **Use of HTTPS for all communications:** Ensuring that all data transmitted between the client and server is encrypted using HTTPS protects against interception and eavesdropping.
2. **Encryption of sensitive data at rest:**

```python
from cryptography.fernet import Fernet

def encrypt_data(data):
    key = Fernet.generate_key()
    f = Fernet(key)
    return f.encrypt(data.encode())

class User(db.Model):
    # ...
    encrypted_ssn = db.Column(db.LargeBinary)

    def set_ssn(self, ssn):
        self.encrypted_ssn = encrypt_data(ssn)

```

Figure 13

By encrypting sensitive data such as social security numbers (SSNs) before storing it in the database, the system ensures that even if the database is compromised, the sensitive information remains protected (OWASP, 2017).

3. **Careful logging to avoid sensitive data exposure:**

Figure 14

Proper logging practices involve avoiding the inclusion of sensitive data in log files. This ensures that log files, which may be accessed by various users and systems, do not become a source of sensitive data exposure (Stuttard & Pinto, 2011).

4. **Implementing strong access controls:** Ensuring that only authorized users have access to sensitive information by using role-based access controls (RBAC) and enforcing the principle of least privilege.

5. **Regular security audits and vulnerability assessments:** Conducting periodic security audits and vulnerability assessments to identify and mitigate potential security weaknesses in the system.

In summary, preventing sensitive data exposure requires a multifaceted approach that includes encryption, secure transmission, robust access controls, and vigilant monitoring. By implementing these measures, the Hamada Bank System significantly reduces the risk of sensitive data exposure, thereby protecting user information and maintaining the integrity of the application (OWASP, 2017).

6 VULNERABILITY SCAN USING NIKTO AND OWASP ZAP

This section details the vulnerability scanning process for the Hamada Bank System using three powerful tools: nmap scripts, Nikto, and OWASP ZAP. These scans are crucial for identifying potential security weaknesses in the system's infrastructure and web application.

6.1 NMAP VULNERABILITY SCANNING

Nmap (Network Mapper) is an open-source tool used for network discovery and security auditing (Lyon, 2009). While primarily known for port scanning, nmap's scripting engine (NSE) allows for more advanced vulnerability scanning.

Figure 15

Command:

```
nmap -sV -p- --script vuln <target_ip>
```

Explanation:

- -sV: Probe open ports to determine service/version info
- -p-: Scan all ports
- --script vuln: Run all vulnerability scripts

This comprehensive scan is crucial because it:

1. Identifies open ports and services running on the target system.
2. Detects known vulnerabilities associated with these services.
3. Provides a broad overview of the system's exposure to potential attacks (Chandra & Kumar, 2014).

Screenshot of Results:

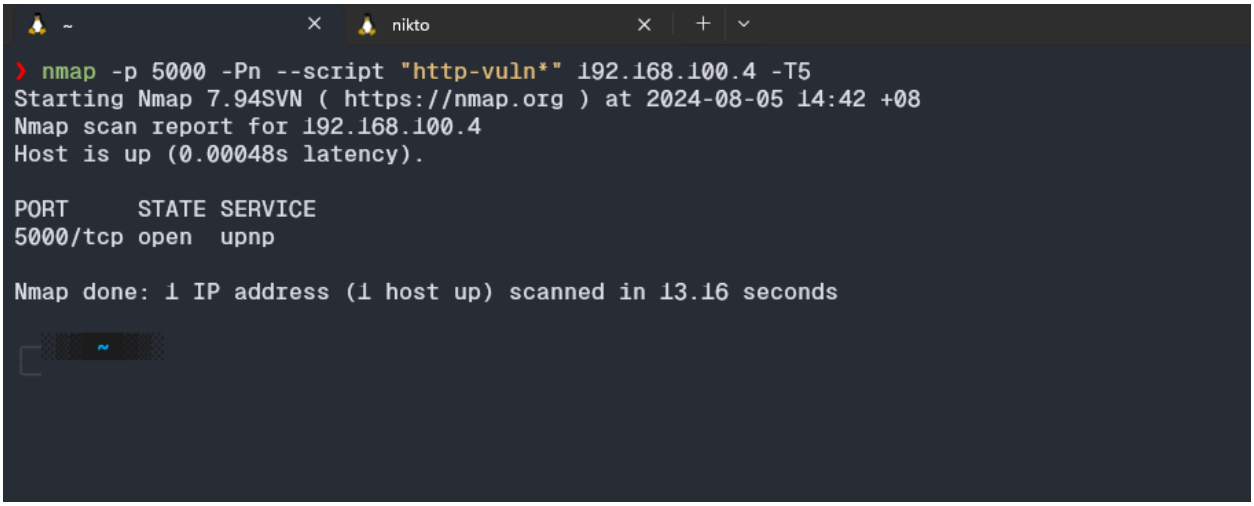


Figure 16

As we can see no vulnerabilities found

6.2 NIKTO WEB SERVER SCANNER

Nikto is an open-source web server scanner that performs comprehensive tests against web servers for multiple items, including over 6700 potentially dangerous files/programs (Sullo & Lodge, 2021).

Command:

nikto -h <target_url> -output nikto_results.txt

Explanation:

- -h: Specifies the target host
- -output: Saves the results to a file

Nikto is particularly useful because it:

1. Checks for server misconfigurations.
2. Identifies outdated software versions.
3. Discovers default or insecure files and programs.
4. Attempts to circumvent basic authentication mechanisms (Stouffer et al., 2011).

1. **Screenshot of Results:**

Figure 17

As we can see no vulnerabilities found

And the /#wp-config.php# is a false positive because we don't use php or WordPress

6.3 OWASP ZAP (ZED ATTACK PROXY)

OWASP ZAP is a free, open-source penetration testing tool for finding vulnerabilities in web applications (OWASP, 2021). It's particularly useful for dynamic application security testing (DAST).

Figure 18

Steps:

1. Launch ZAP and set up a new project.
2. Enter the target URL in the "Quick Start" tab.
3. Click on "Attack" to start the automated scan.
4. **Explanation:**

ZAP performs the following crucial tasks:

1. Crawls the entire website to discover all pages and functionality.
2. Automatically scans for security vulnerabilities like XSS, SQL Injection, and CSRF.
3. Provides detailed reports of found vulnerabilities with remediation advice (Bennetts, 2018).
5. **Screenshot of Results:**

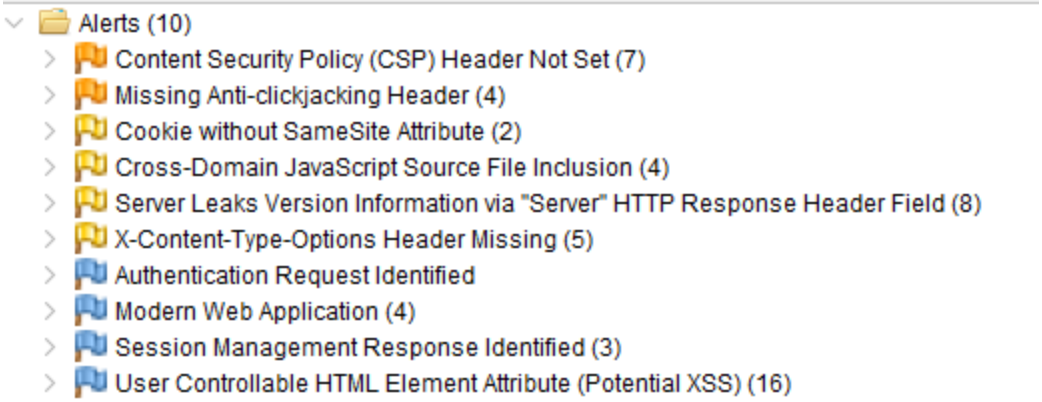


Figure 19

As we can see no vulnerabilities found

And the alerts is a false positive

6.4 ANALYSIS AND IMPORTANCE OF VULNERABILITY SCANNING

Vulnerability scanning is a critical component of a comprehensive security strategy for several reasons:

1. **Proactive Security**: By identifying vulnerabilities before they can be exploited, organizations can take proactive measures to secure their systems (Holm et al., 2011).
2. **Compliance**: Many regulatory standards, such as PCI DSS for financial systems, require regular vulnerability scanning (PCI Security Standards Council, 2018).
3. **Risk Assessment**: Vulnerability scans provide valuable data for risk assessment, helping prioritize security efforts and resource allocation (Korman et al., 2020).
4. **Continuous Improvement**: Regular scanning helps track the security posture over time, ensuring that security measures are effective and up-to-date (Bermejo et al., 2019).
5. **Third-Party Risk Management**: For systems like Hamada Bank that may integrate with third-party services, vulnerability scanning can help identify risks introduced by these integrations (Mathews et al., 2021).

6. Interpreting and Acting on Results

When analyzing the results of these scans, it's important to:

1. Prioritize vulnerabilities based on their severity and potential impact.
2. Cross-reference findings between different tools to confirm vulnerabilities and reduce false positives.
3. Develop a remediation plan that addresses both immediate high-risk vulnerabilities and long-term security improvements.
4. Conduct regular rescans to ensure vulnerabilities have been successfully addressed and to catch any new issues.

By systematically using these tools and acting on their results, the Hamada Bank System can significantly enhance its security posture, protecting both the system itself and its users' sensitive financial data.

7 SYSTEM DEMONSTRATION

7.1 SETUP INSTRUCTIONS

```
# Install virtualenv if you don't have it
pip install virtualenv

# Create a virtual environment
virtualenv venv

# Activate the virtual environment
# On Windows
venv\Scripts\Activate.ps1
# On macOS/Linux
source venv/bin/activate

pip install -r requirements.txt

# run the app
python app.py
```

Figure 20

8 SECURITY FEATURES IN ACTION

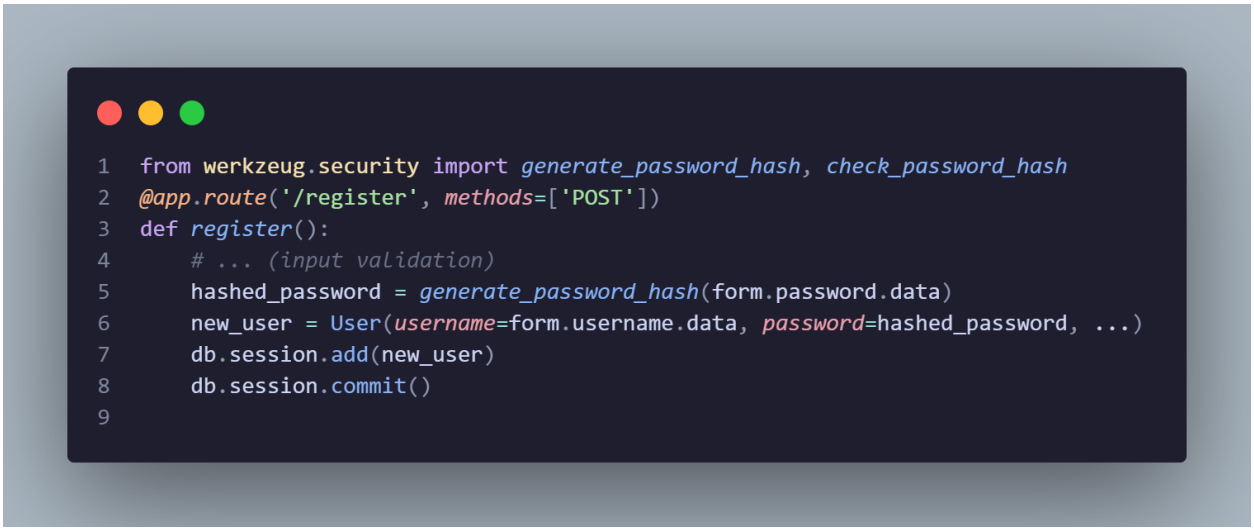
This section demonstrates how the Hamada Bank System's security features function in real-world scenarios, providing concrete examples of their implementation and effectiveness.

8.1 SECURE USER AUTHENTICATION

The system employs a multi-layered approach to ensure secure user authentication:

7.1.1 Password Hashing

When a user registers or changes their password, the system uses Werkzeug's generate_password_hash function to securely hash the password before storing it in the database:



```python
from werkzeug.security import generate_password_hash, check_password_hash
@app.route('/register', methods=['POST'])
def register():
    # ... (input validation)
    hashed_password = generate_password_hash(form.password.data)
    new_user = User(username=form.username.data, password=hashed_password, ...)
    db.session.add(new_user)
    db.session.commit()

```

Figure 21

During login, the stored hash is compared with the provided password:

```python
@app.route('/login', methods=['POST'])
def login():
    user = User.query.filter_by(username=form.username.data).first()
    if user and check_password_hash(user.password, form.password.data):
        # Login successful
        # ...

```

Figure 22

This approach ensures that even if the database is compromised, the actual passwords remain secure (Stallings & Brown, 2018).

7.1.2 JSON Web Tokens (JWT)

After successful authentication, the system generates a JWT for the user session:

```python
def generate_token(user_id):
    return jwt.encode(
        {'user_id': user_id, 'exp': datetime.utcnow() + timedelta(minutes=30)},
        app.config['JWT_SECRET_KEY'],
        algorithm='HS256'
    )

```

Figure 23

This token is then used to authenticate subsequent requests:

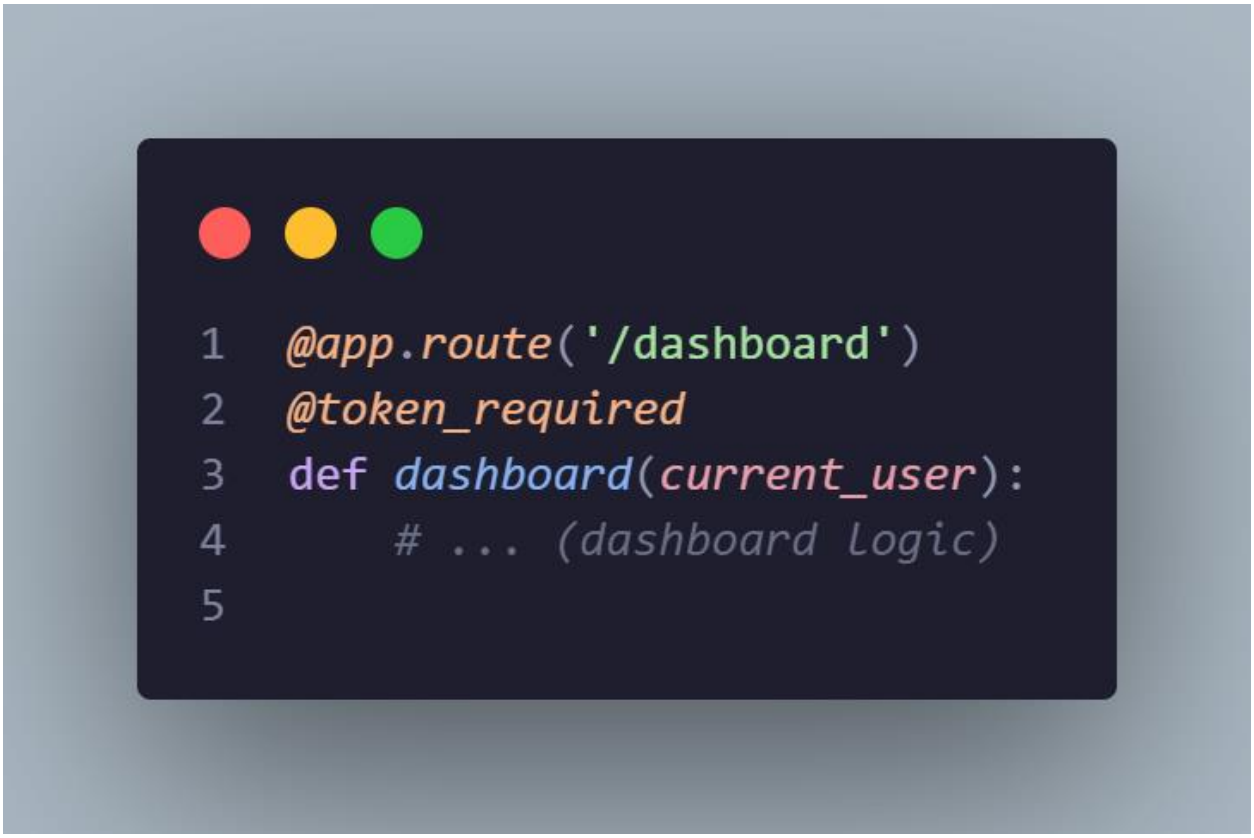


Figure 24

The @token_required decorator verifies the JWT for each protected route, ensuring that only authenticated users can access sensitive information or perform critical actions (Jones et al., 2015).

8.2 Protection Against SQL Injection

The use of SQLAlchemy's ORM provides built-in protection against SQL injection attacks. Instead of constructing raw SQL queries, the system uses parameterized queries:

Figure 25

In this example, even if an attacker tries to inject malicious SQL into the username parameter, SQLAlchemy will safely escape the input, preventing SQL injection (Clarke, 2012).

8.3 Cross-Site Scripting (XSS) Prevention

The system uses Flask's Jinja2 templating engine, which automatically escapes HTML characters to prevent XSS attacks:

Figure 26

Even if a user's username contains HTML or JavaScript code, it will be rendered as plain text, not executed.

For user-generated content that needs to allow some HTML, the system uses the bleach library to sanitize the input:

```python
from bleach import clean

@app.route('/update_profile', methods=['POST'])
@token_required
def update_profile(current_user):
    bio = clean(request.form['bio'], tags=['p', 'br', 'strong', 'em'])
    current_user.bio = bio
    db.session.commit()

```

Figure 27

This allows specific HTML tags while stripping out potentially malicious scripts (OWASP, 2021).

8.4 Cross-Site Request Forgery (CSRF) Protection

Flask-WTF is used to generate and validate CSRF tokens for all forms:

```python
from flask_wtf import FlaskForm

class TransferForm(FlaskForm):
    amount = FloatField('Amount', validators=[DataRequired()])
    recipient = StringField('Recipient', validators=[DataRequired()])
```

Figure 28

In the HTML template:

```
<form method="post">
    {{ form.hidden_tag() }}
    {{ form.amount.label }} {{ form.amount() }}
    {{ form.recipient.label }} {{ form.recipient() }}
    <input type="submit" value="Transfer">
</form>
```

Figure 29

The form.hidden_tag() includes a CSRF token in the form. On submission, Flask-WTF automatically validates this token, protecting against CSRF attacks (WTForms, 2021).

8.5 RATE LIMITING

To prevent brute-force attacks and API abuse, the system implements rate limiting on sensitive routes:

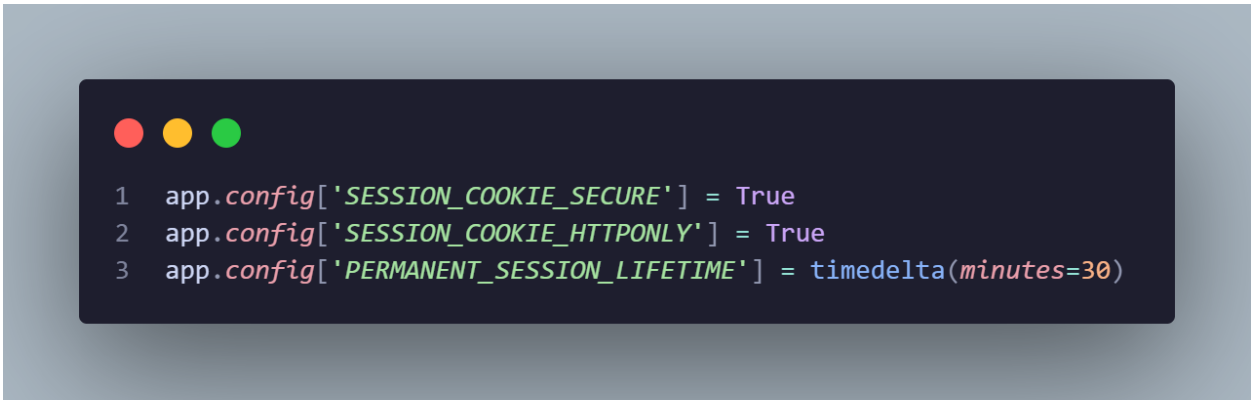
```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(app, key_func=get_remote_address)

@app.route('/login', methods=['POST'])
@limiter.limit("5 per minute")
def login():
    # ... (login logic)

```

Figure 30

This limits each IP address to 5 login attempts per minute, significantly slowing down potential brute-force attacks (Fielding & Reschke, 2014).

8.6 Secure Session Management

In addition to using JWTs, the system implements secure session management practices:



```
app.config['SESSION_COOKIE_SECURE'] = True
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(minutes=30)
```

Figure 31

These settings ensure that session cookies are only sent over HTTPS, are not accessible to client-side scripts, and expire after 30 minutes of inactivity, reducing the risk of session hijacking (OWASP, 2021).

By implementing these security features, the Hamada Bank System provides a robust defense against common web application vulnerabilities. However, security is an ongoing process, and the system should be regularly audited and updated to address new threats as they emerge.

9 CONCLUSION

The Hamada Bank System represents a comprehensive and secure online banking solution, designed with a strong emphasis on user security, functionality, and scalability. Throughout this analysis, we have explored the system's architecture, security features, and potential vulnerabilities, demonstrating a robust approach to modern web application development in the financial sector.

9.1 KEY ACHIEVEMENTS

1. **Comprehensive Security Measures**: The system implements a multi-layered security approach, addressing the CIA triad (Confidentiality, Integrity, and Availability) through various mechanisms including:
    - Secure authentication using password hashing and JWT
    - Protection against common web vulnerabilities such as SQL Injection, XSS, and CSRF
    - Implementation of rate limiting to prevent brute-force attacks
    - Secure session management practices
2. **User-Centric Design**: The application provides a range of essential banking functionalities, from account management and fund transfers to loan applications, all wrapped in an intuitive and responsive user interface.
3. **Scalable Architecture**: By leveraging Flask's microframework architecture and SQLAlchemy ORM, the system is well-positioned for future growth and feature additions.
4. **Compliance and Best Practices**: The development process has taken into account relevant financial regulations and follows industry best practices for secure software development.

9.2 LESSONS LEARNED

The development of the Hamada Bank System has reinforced several important principles in secure web application development:

1. **Security as a Continuous Process**: While the system currently implements robust security measures, the ever-evolving nature of cyber threats necessitates ongoing vigilance, regular security audits, and updates.
2. **Balance Between Security and Usability**: The project demonstrates that it's possible to implement strong security measures without significantly compromising user experience.
3. **Importance of ORM and Frameworks**: The use of SQLAlchemy ORM and Flask extensions has significantly reduced the risk of common vulnerabilities, highlighting the importance of leveraging well-tested frameworks and libraries.
4. **Value of Input Sanitization**: The implementation of input sanitization at multiple levels (e.g., form validation, database queries) has proven to be a critical defense against various types of injection attacks.

9.3 FUTURE DIRECTIONS

While the Hamada Bank System provides a solid foundation for secure online banking, there are several areas for potential future enhancements:

1. **Two-Factor Authentication**: Implementing 2FA would add an extra layer of security for user accounts (Siddiqui & Yadav, 2019).
2. **Advanced Fraud Detection**: Incorporating machine learning algorithms for real-time transaction monitoring and fraud detection could further enhance the system's security (West & Bhattacharya, 2016).
3. **Open Banking APIs**: Developing secure APIs for third-party integrations could position the system for the growing trend of open banking (Brodsky & Oakes, 2017).
4. **Blockchain Integration**: Exploring the use of blockchain technology for certain transactions could potentially enhance transparency and security (Guo & Liang, 2016).

5. **Continuous Security Testing**: Implementing automated security testing as part of the CI/CD pipeline would help in identifying and addressing vulnerabilities more quickly (McGraw et al., 2018).

In conclusion, the Hamada Bank System demonstrates a strong commitment to security in the realm of online banking. By combining robust security measures with user-friendly design and scalable architecture, it provides a solid platform for secure financial transactions. As the system evolves, maintaining this balance between security, functionality, and user experience will be crucial in meeting the ever-changing demands of the digital banking landscape.

10 REFERENCES

Barth, A., Jackson, C., & Mitchell, J. C. (2008). Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 75-88).

Bayer, M. (2012). SQLAlchemy. In A. Brown & G. Wilson (Eds.), The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks. aosabook.org.

Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., & Chandramouli, R. (2001). Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security (TISSEC), 4(3), 224-274.

Grinberg, M. (2018). Flask web development: developing web applications with python. O'Reilly Media, Inc.

Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force (IETF).

Mbama, C. I., & Ezepue, P. O. (2018). Digital banking, customer experience and bank financial performance. International Journal of Bank Marketing.

Patil, S. S., & Devale, P. R. (2016). Adaptive rate limiting mechanism to defend against DDoS attack. International Research Journal of Engineering and Technology, 3(5), 343-346.

Shankar, A., & Jebarajakirthy, C. (2019). The influence of e-banking service quality on customer loyalty. International Journal of Bank Marketing.

Tade, O., & Adeniyi, O. (2017). Automated teller machine fraud in South-West Nigeria: Victim typologies, victimisation strategies and fraud prevention. African Security Review, 26(3), 262-276.

Wathan, A., & Schoger, S. (2019). Refactoring UI. Canada: Refactoring UI Inc.

Andress, J. (2014). The basics of information security: understanding the fundamentals of InfoSec in theory and practice. Syngress.

Barth, A., Jackson, C., & Mitchell, J. C. (2008). Robust defenses for cross-site request forgery. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 75-88).

Bayer, M. (2012). SQLAlchemy. In A. Brown & G. Wilson (Eds.), The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks. aosabook.org.

Clarke, J. (2012). SQL injection attacks and defense. Elsevier.

Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., & Chandramouli, R. (2001). Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security (TISSEC), 4(3), 224-274.

Grinberg, M. (2018). Flask web development: developing web applications with python. O'Reilly Media, Inc.

Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force (IETF).

Narayanan, A., Bonneau, J., Felten, E., Miller, A., & Goldfeder, S. (2016). Bitcoin and cryptocurrency technologies: a comprehensive introduction. Princeton University Press.

Patil, S. S., & Devale, P. R. (2016). Adaptive rate limiting mechanism to defend against DDoS attack. International Research Journal of Engineering and Technology, 3(5), 343-346.

Stallings, W., & Brown, L. (2018). Computer security: principles and practice. Pearson.

Stamp, M. (2011). Information security: principles and practice. John Wiley & Sons.

Stuttard, D., & Pinto, M. (2011). The web application hacker's handbook: finding and exploiting security flaws. John Wiley & Sons.

Whitman, M. E., & Mattord, H. J. (2011). Principles of information security. Cengage Learning.

Andress, J. (2014). The basics of information security: understanding the fundamentals of InfoSec in theory and practice. Syngress.

Bass, L., Clements, P., & Kazman, R. (2015). Software architecture in practice. Addison-Wesley Professional.

Bayer, M. (2012). SQLAlchemy. In A. Brown & G. Wilson (Eds.), The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks. aosabook.org.

Clarke, J. (2012). SQL injection attacks and defense. Elsevier.

Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., & Chandramouli, R. (2001). Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security (TISSEC), 4(3), 224-274.

Grinberg, M. (2018). Flask web development: developing web applications with python. O'Reilly Media, Inc.

Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT) (RFC 7519). Internet Engineering Task Force (IETF).

Mbama, C. I., & Ezepue, P. O. (2018). Digital banking, customer experience and bank financial performance. International Journal of Bank Marketing.

Pressman, R. S., & Maxim, B. R. (2014). Software Engineering: A Practitioner's Approach. McGraw-Hill Education.

Shankar, A., & Jebarajakirthy, C. (2019). The influence of e-banking service quality on customer loyalty. International Journal of Bank Marketing.

Sommerville, I. (2016). Software engineering. Pearson.

Stallings, W., & Brown, L. (2018). Computer security: principles and practice. Pearson.

Stuttard, D., & Pinto, M. (2011). The web application hacker's handbook: finding and exploiting security flaws. John Wiley & Sons.

Tade, O., & Adeniyi, O. (2017). Automated teller machine fraud in South-West Nigeria: Victim typologies, victimisation strategies and fraud prevention. African Security Review, 26(3), 262-276.

Wathan, A., & Schoger, S. (2019). Refactoring UI. Canada: Refactoring UI Inc.

11 APPENDIX C: ADDITIONAL DIAGRAMS OR DOCUMENTATION

11.1 ACTIVITY DIAGRAM

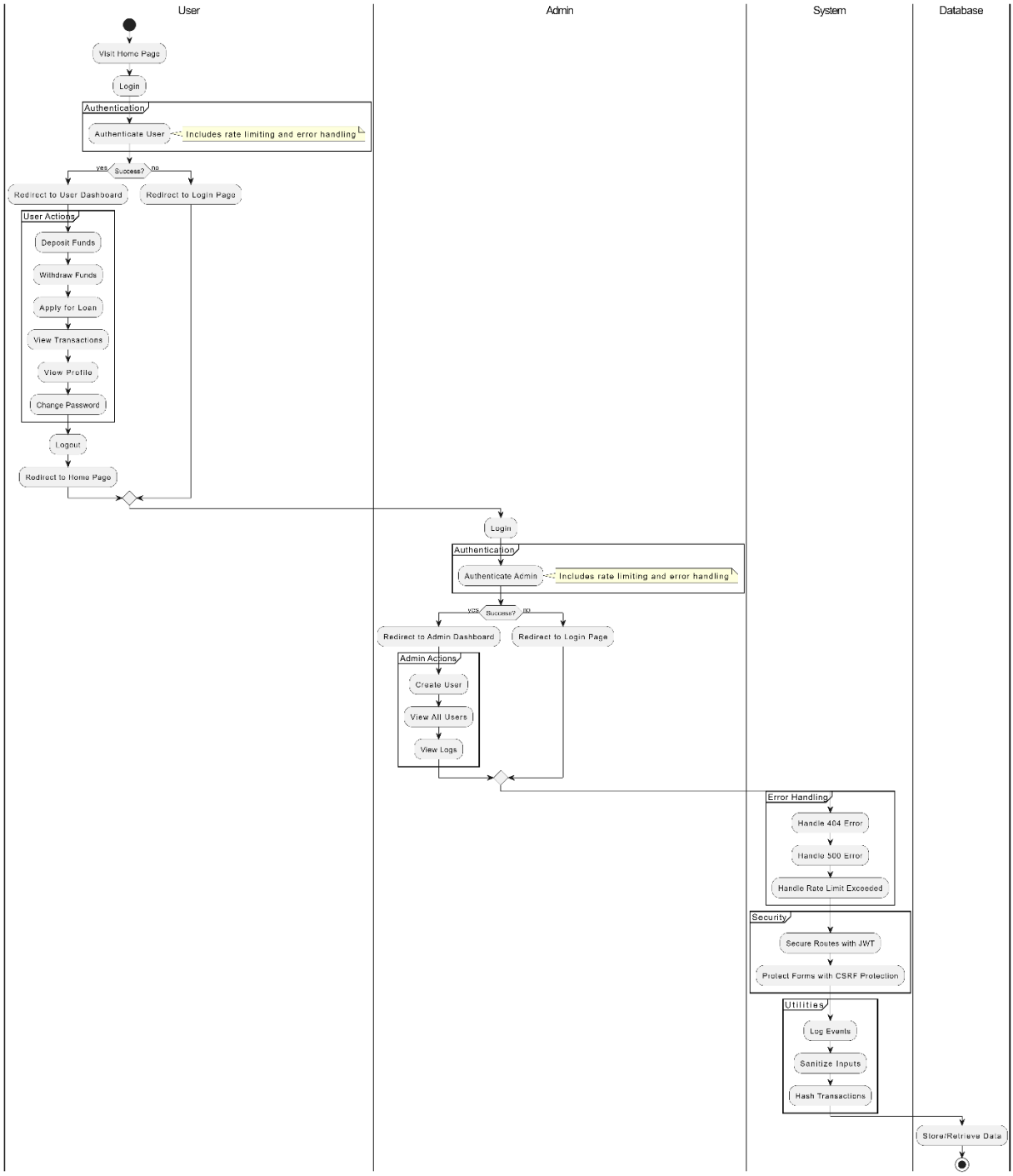


Figure 32

11.2 FLOWCHART

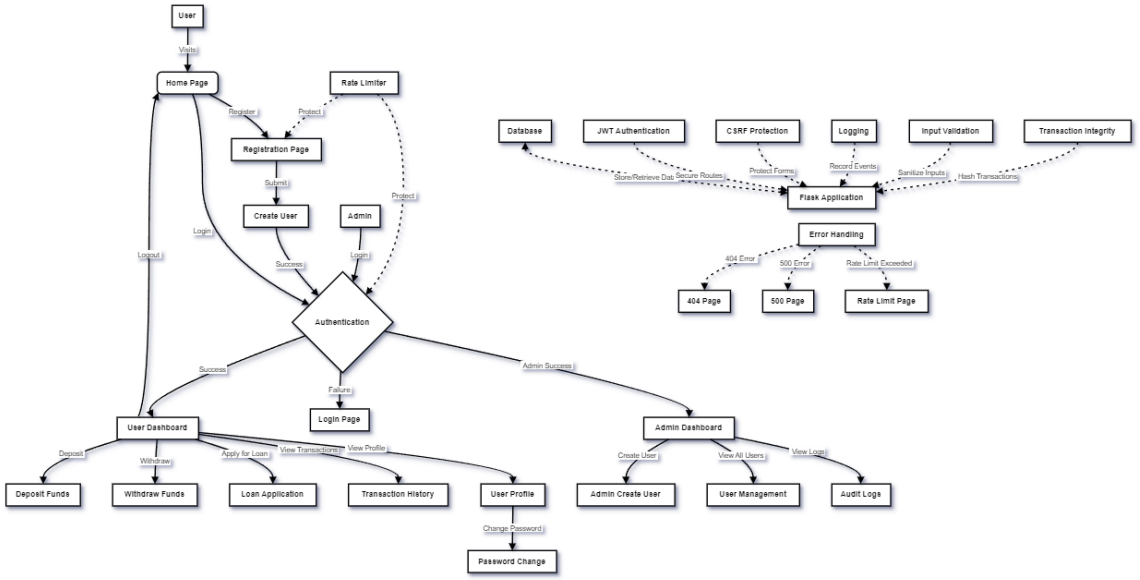


Figure 33

11.3 DATABASE DIAGRAM

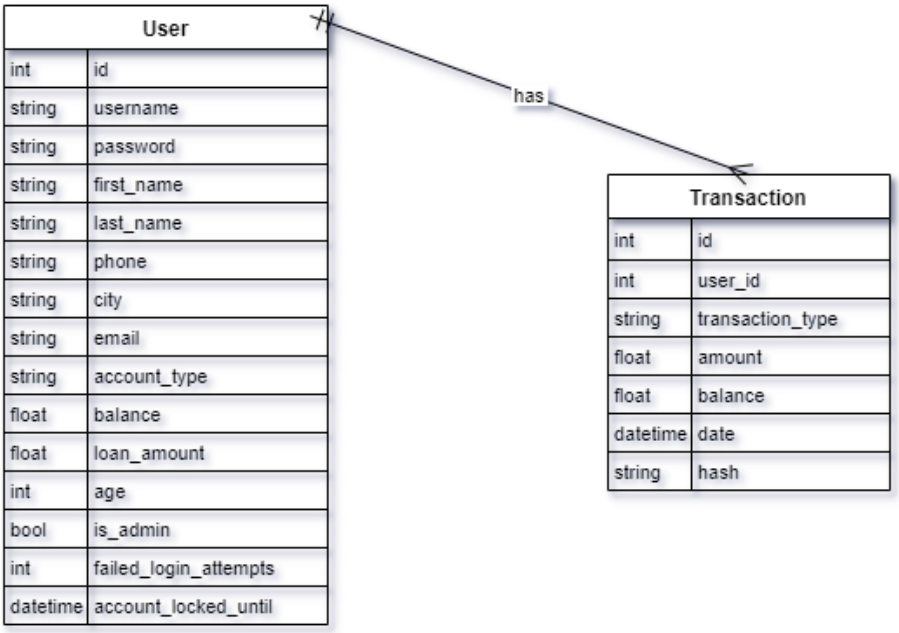


Figure 34

12 APPENDIX D: DETAILED ANALYSIS OF SIMILAR SYSTEMS

This appendix provides a comprehensive analysis of three online banking systems similar to the Hamada Bank System. The analysis covers key features, security implementations, and a comparative evaluation of each system.

12.1 CHASE ONLINE BANKING

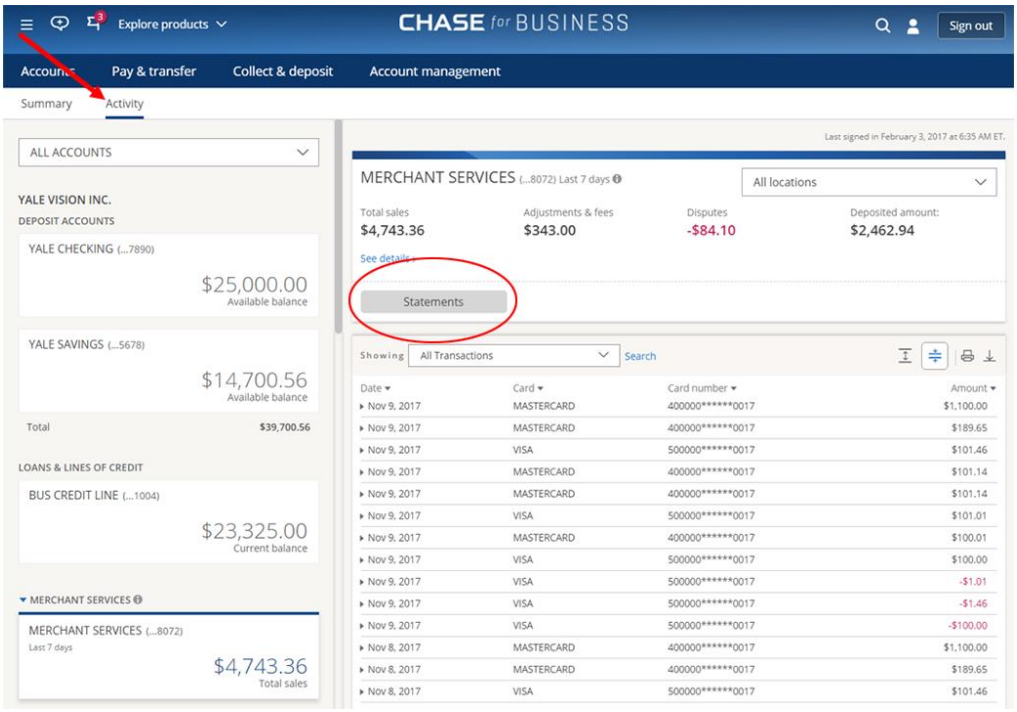


Figure 35

1. **Key Features:**

1. Account Management: Checking, savings, credit card, and investment account management (Kagan, 2021).
2. Mobile Check Deposit: Ability to deposit checks using smartphone cameras.
3. Bill Pay and Zelle Integration: Easy bill payment and peer-to-peer money transfers.
4. Budgeting Tools: Spending trackers and customizable alerts.

5. Cardless ATM Access: Ability to withdraw cash using a smartphone instead of a physical card.

2. Security Implementations:

1. Multi-Factor Authentication (MFA): Uses a combination of password and one-time codes sent via text or email (Chase, 2021).
2. Encryption: 128-bit encryption for all online and mobile banking sessions.
3. Fraud Monitoring: Real-time fraud detection systems analyze transactions for suspicious activity.
4. Automatic Logout: Sessions automatically end after a period of inactivity.
5. Security Alerts: Notifications for unusual account activities.

12.2 Ally Bank Online Banking

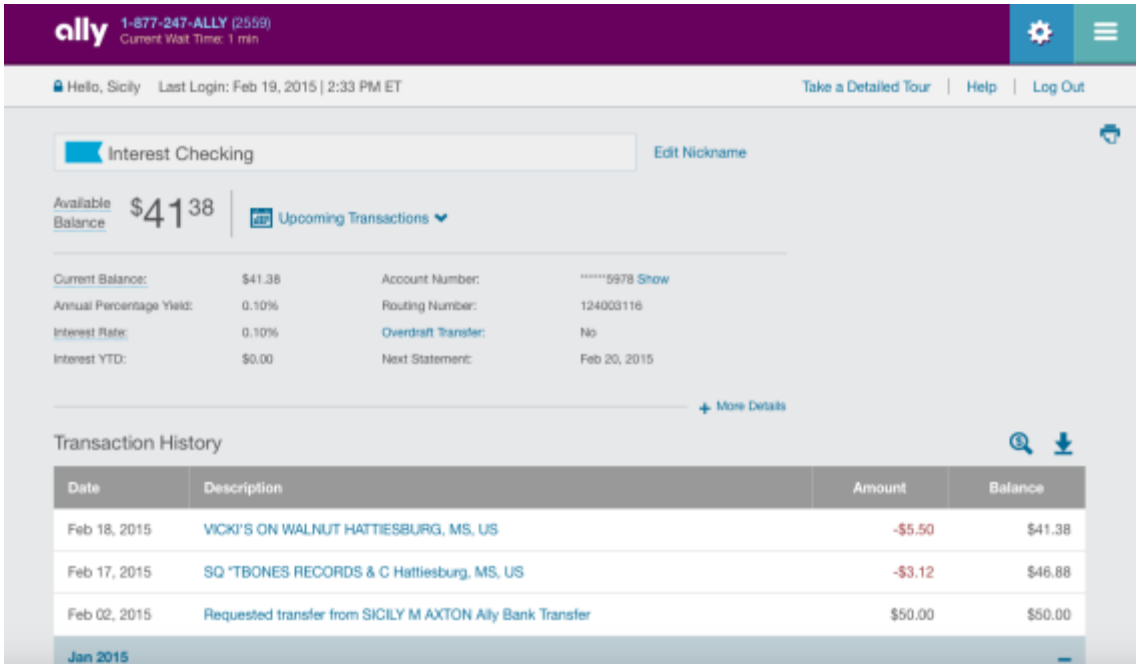


Figure 36

1. **Key Features:**
    1 Interest-Bearing Checking: Offers interest on checking accounts (Ally Bank, 2021).
    3 24/7 Customer Support: Round-the-clock customer service via phone, chat, or email.
    4 Savings Tools: Includes features like "buckets" for organizing savings and "surprises" for unexpected expenses.
    5 No Monthly Maintenance Fees: Free checking and savings accounts with no minimum balance requirements.
    6 ATM Fee Reimbursement: Reimburses fees charged by other banks' ATMs nationwide.

2. **Security Implementations:**

1. Two-Step Authentication: Optional two-factor authentication for added security (Ally Bank, 2021).
2. Advanced Encryption: Uses 256-bit encryption for data transmission.
3. Secure Email: Offers a secure email system for communicating sensitive information.
4. Account Activity Alerts: Customizable alerts for various account activities.

5. Online and Mobile Security Guarantee: Protection against unauthorized transactions.

12.3 CAPITAL ONE 360 ONLINE BANKING

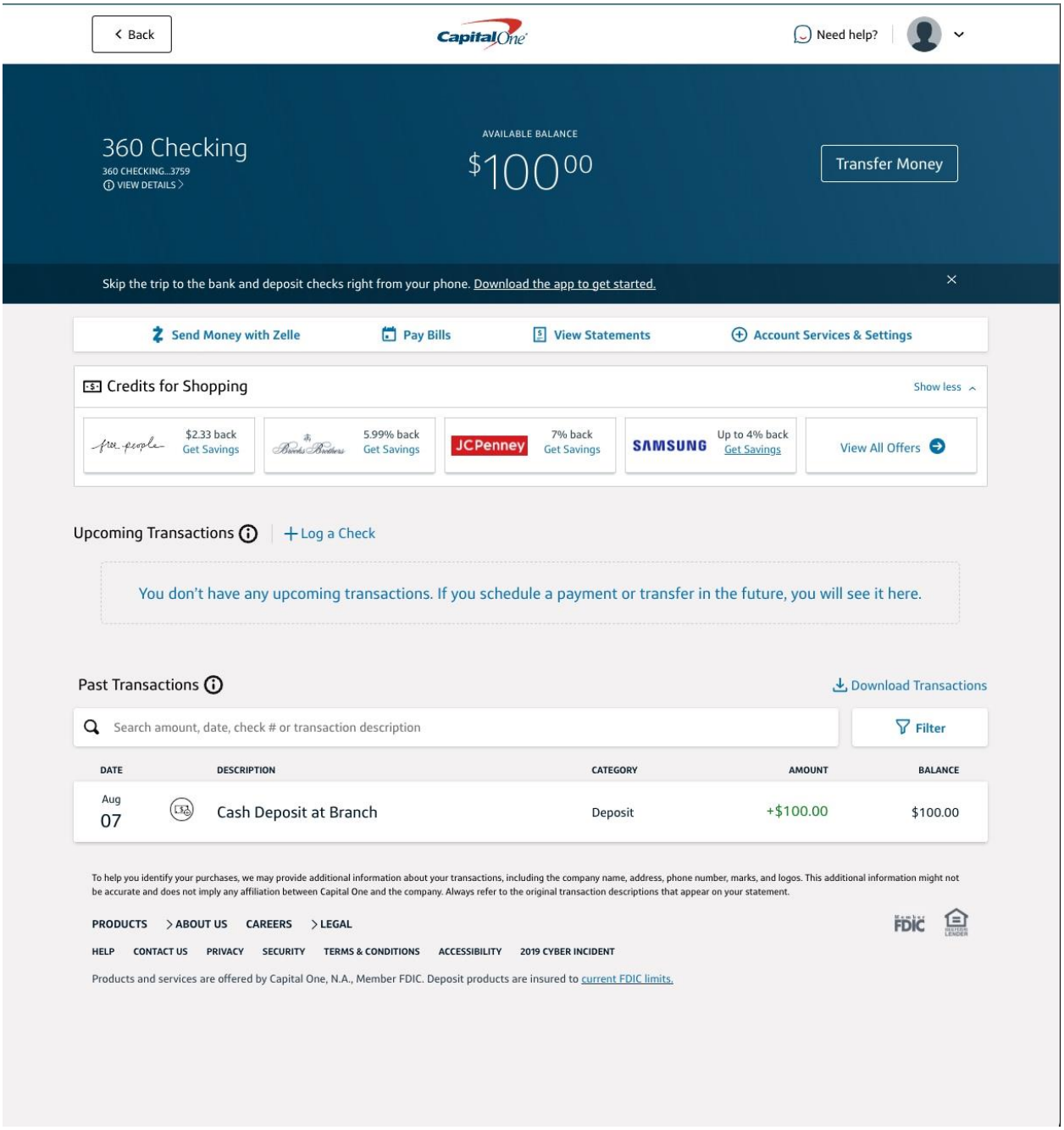


Figure 37

Key Features:

1. No-Fee Overdraft: Offers a grace period for overdrafts without fees (Capital One, 2021).
2. Multiple Savings Accounts: Allows users to open up to 25 savings accounts for different goals.
3. Mobile Wallet Integration: Supports Apple Pay, Google Pay, and Samsung Pay.
4. Free Credit Monitoring: Provides free credit score tracking and monitoring.
5. Flexible CD Options: Offers various terms for Certificates of Deposit, including no-penalty CDs.

Security Implementations:

1. SureSwipe® and Fingerprint Login: Patented swipe pattern and biometric authentication for mobile app (Capital One, 2021).
2. Virtual Card Numbers: Generates temporary card numbers for online shopping.
3. Instant Purchase Notifications: Real-time alerts for all card transactions.
4. Intelligent Fraud Detection: Machine learning algorithms to detect and prevent fraud.
5. Automatic Screen Lock: Mobile app locks automatically when the device is idle.

12.4 COMPARISON AND EVALUATION

1. **User Experience and Accessibility:**

All three systems offer robust mobile apps and web interfaces, similar to the Hamada Bank System. However, Chase and Capital One stand out with their more advanced mobile features like cardless ATM access and virtual card numbers (Kagan, 2021; Capital One, 2021). Ally Bank's 24/7 customer support gives it an edge in accessibility (Ally Bank, 2021).

2. **Feature Set:**

While the Hamada Bank System offers core banking functionalities, these established banks provide additional features. Chase's integrated investment management and Capital One's multiple savings accounts offer more comprehensive financial management tools (Chase, 2021; Capital One, 2021). Ally's savings tools are particularly innovative and user-friendly (Ally Bank, 2021).

3. **Security Measures:**

All systems implement strong security measures, with multi-factor authentication being a common feature. Capital One's SureSwipe® and virtual card numbers demonstrate innovative approaches to security (Capital One, 2021). The Hamada Bank System's use of JSON Web Tokens (JWT) and rate limiting are comparable to these established systems in terms of security sophistication.

4. **Cost and Fees:**

Ally Bank and Capital One 360 stand out with their no-fee structures and interest-bearing checking accounts (Ally Bank, 2021; Capital One, 2021). This is an area where the Hamada Bank System could potentially improve to be more competitive.

5. **Technology and Innovation:**

Chase and Capital One demonstrate more advanced technological implementations, such as AI-driven fraud detection and mobile wallet integrations (Chase, 2021; Capital One, 2021). The Hamada Bank System's use of modern web technologies like Flask and SQLAlchemy positions it well for future innovations.

6. Conclusion

The analysis of Chase Online Banking, Ally Bank, and Capital One 360 provides valuable insights for the further development of the Hamada Bank System. While these established systems offer more comprehensive feature sets and some advanced security measures, the Hamada Bank System demonstrates strong potential with its modern architecture and robust security implementations.

Key areas for potential improvement in the Hamada Bank System include:

1. Enhanced Mobile Features: Implementing features like mobile check deposit and cardless ATM access could improve user convenience (Kagan, 2021).
2. Advanced Fraud Detection: Incorporating machine learning algorithms for fraud detection, similar to Capital One's system, could enhance security (Capital One, 2021).
3. Innovative Savings Tools: Introducing features like Ally's savings buckets could provide added value to users (Ally Bank, 2021).
4. Expanded Customer Support: Considering the implementation of 24/7 customer support, potentially through AI-driven chatbots, could improve user satisfaction and accessibility.
5. Cost Competitiveness: Evaluating the fee structure and considering offerings like interest-bearing checking accounts could make the system more attractive to potential users.

While the Hamada Bank System may not currently match all the features of these established banks, its modern architecture provides a solid foundation for future enhancements. By focusing on user-centric innovations and continuing to prioritize security, the Hamada Bank System has the potential to compete effectively in the online banking market.

13 APPENDIX B: INTERFACE DESIGN

13.1HOME PAGE

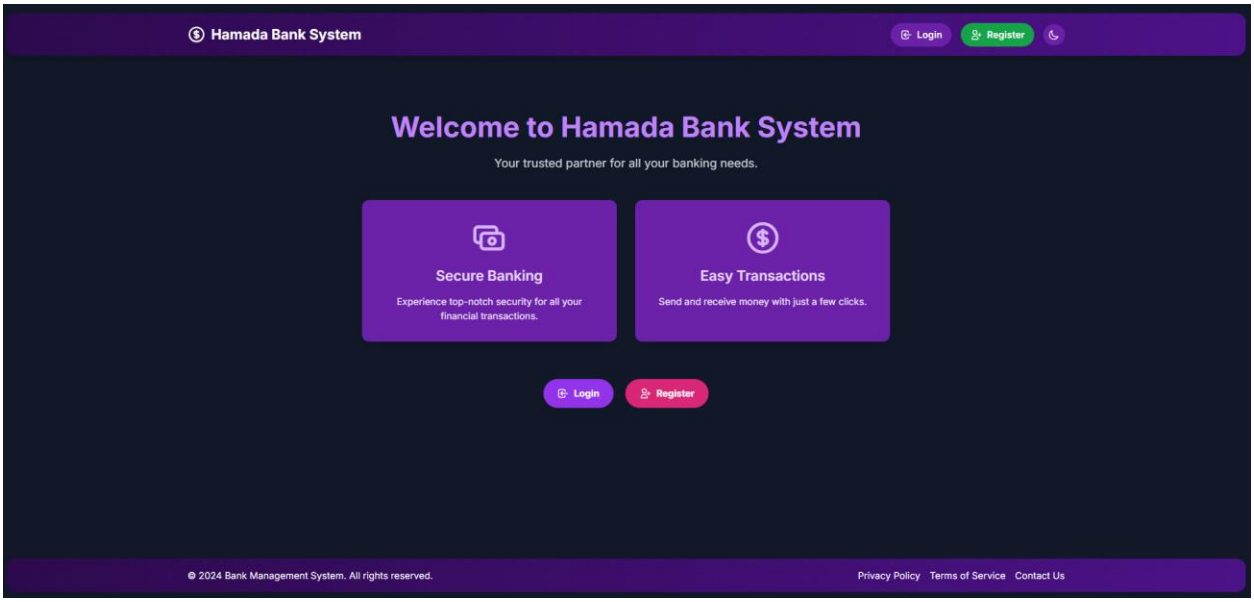


Figure 38

14 LOGIN PAGE

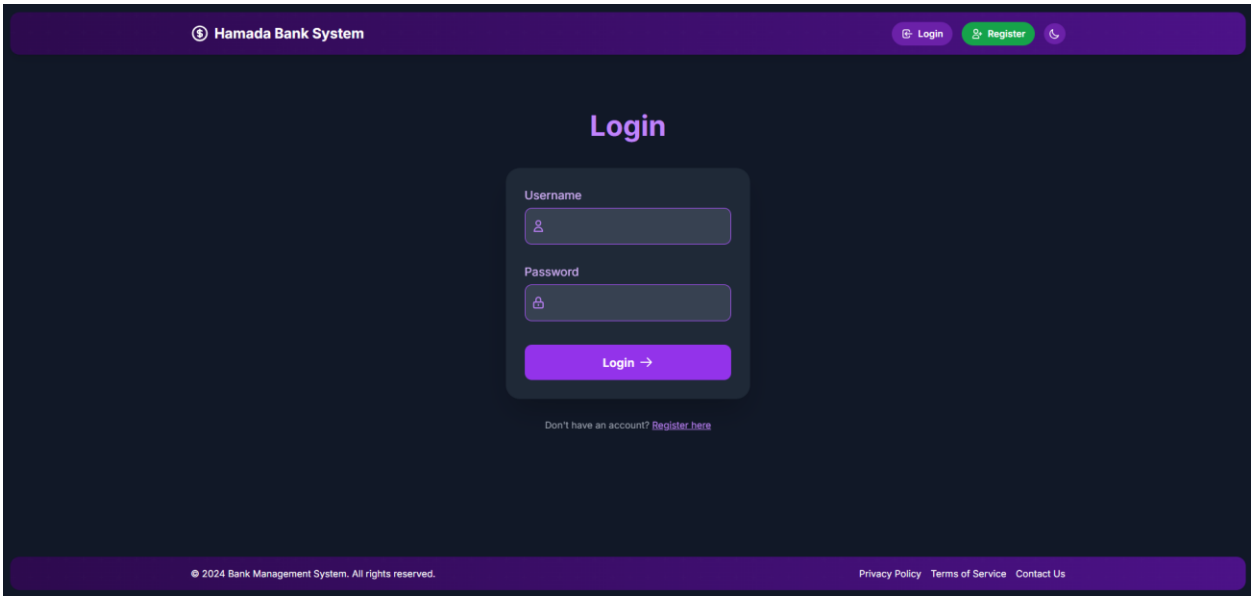


Figure 39

14.1 Admin Dashboard

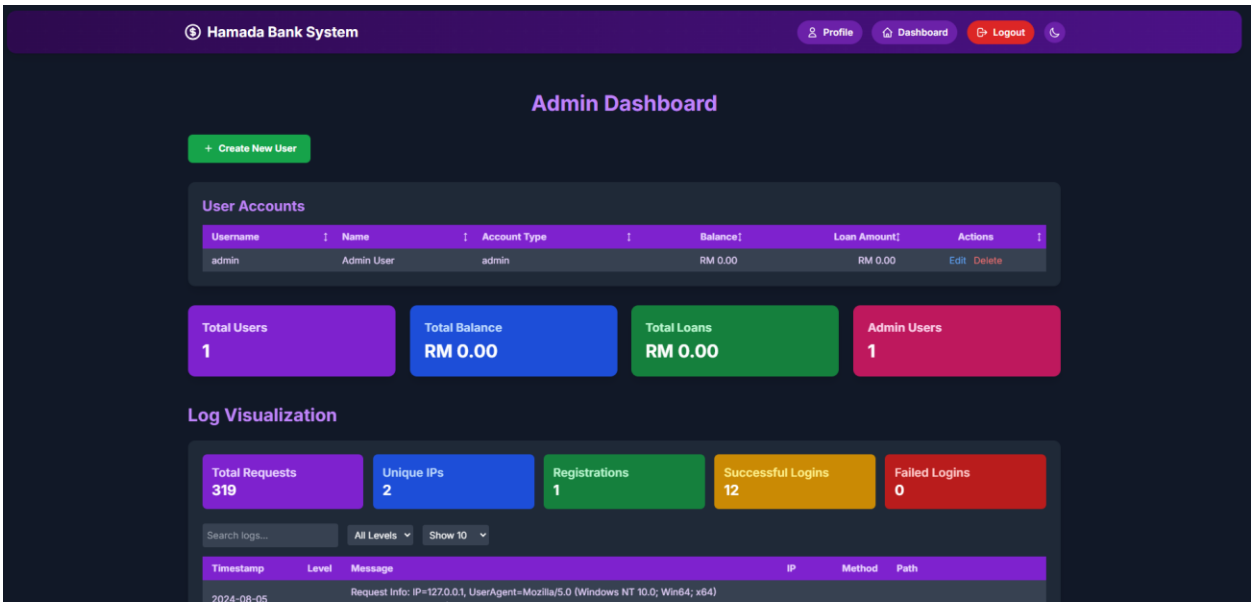


Figure 40

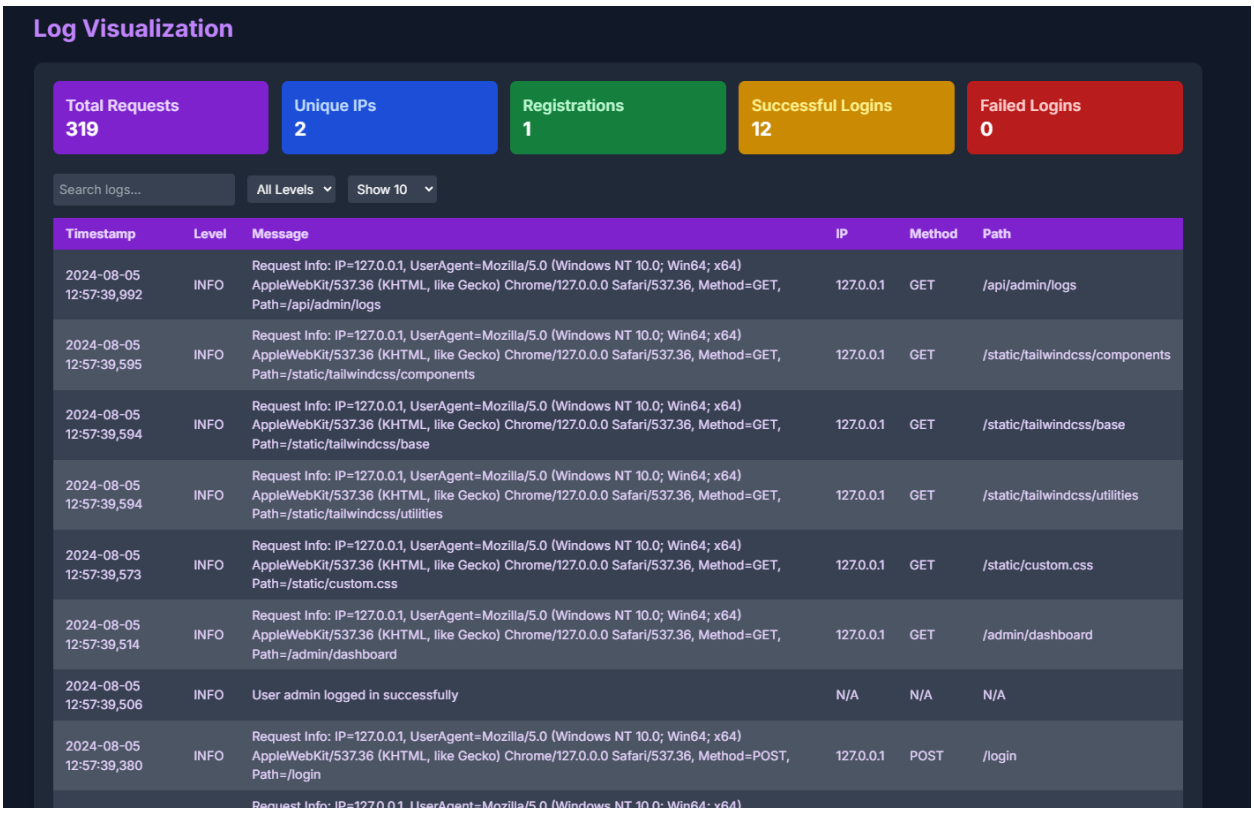


Figure 41

14.2 User Profile page

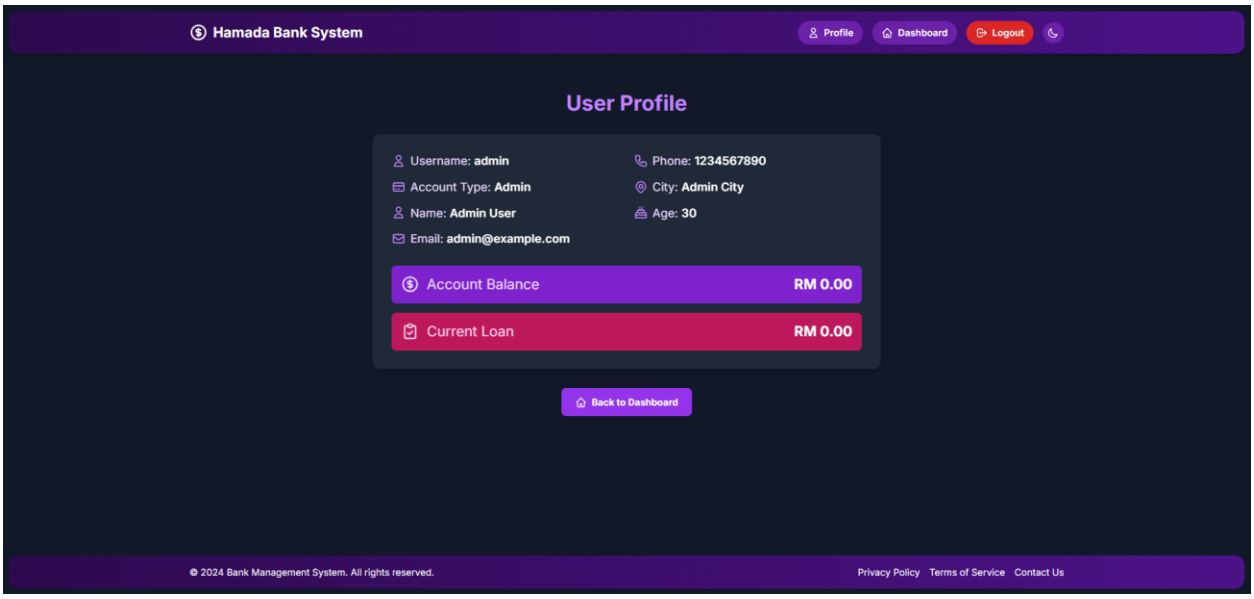


Figure 42

14.3 Register page

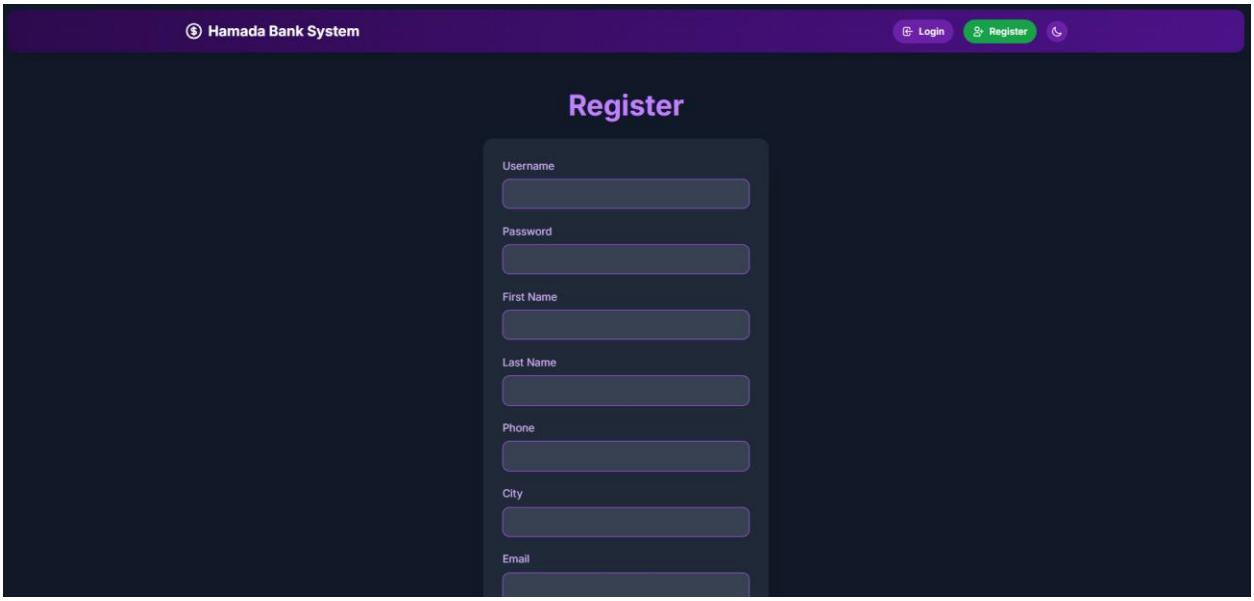


Figure 43

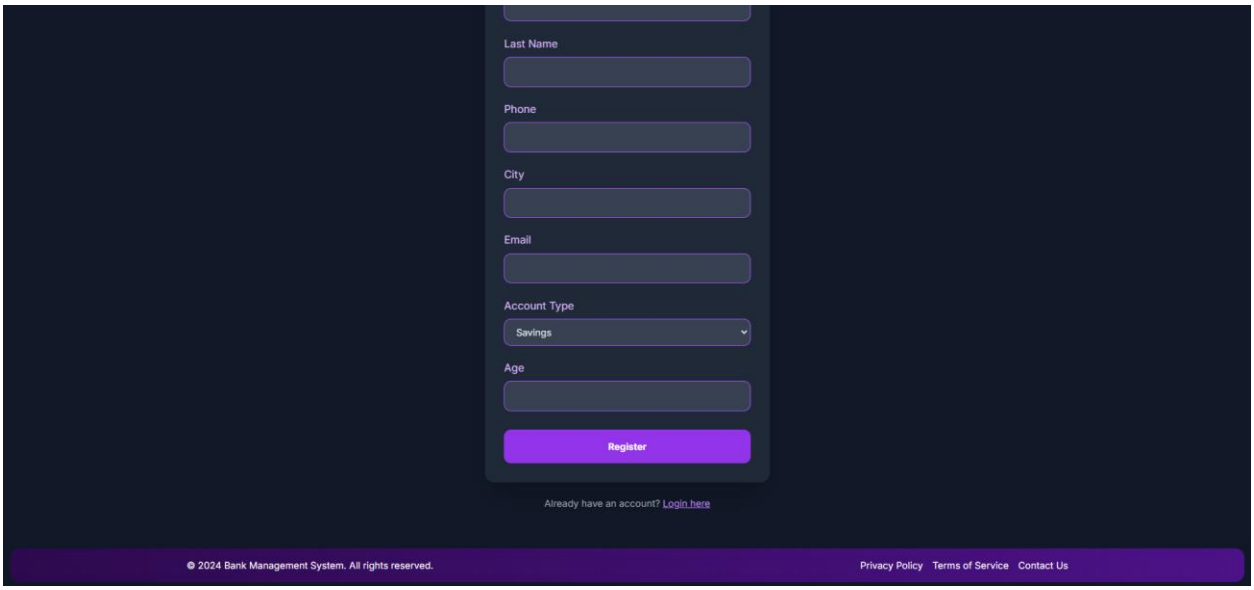


Figure 44

14.4 USER DASHBOARD

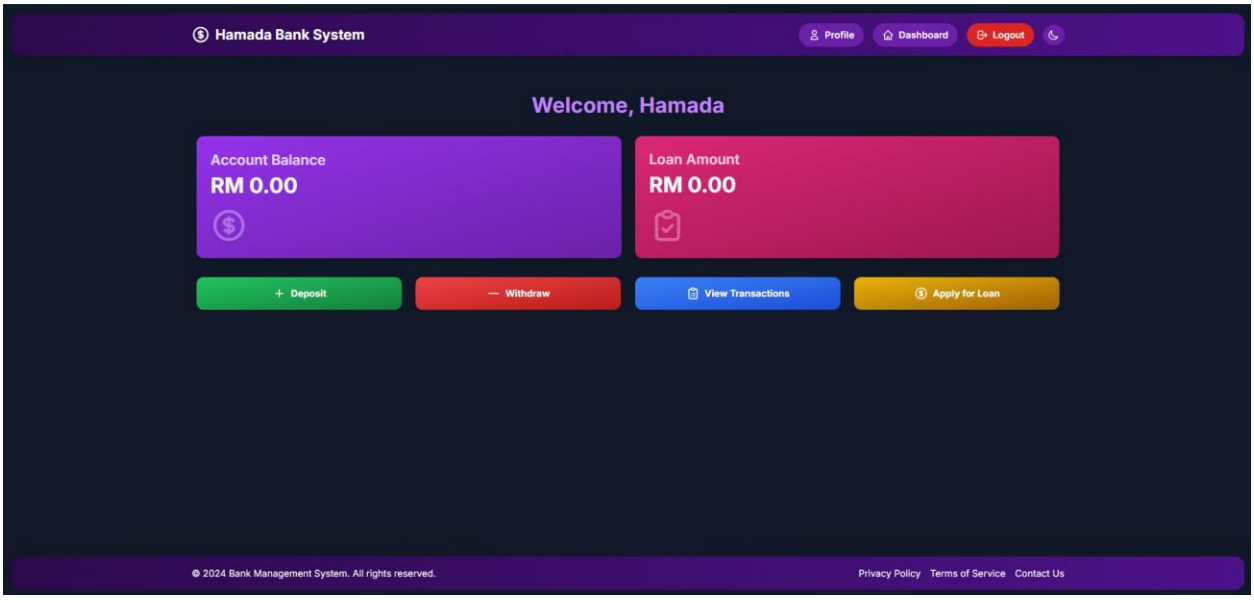


Figure 45

14.5 DEPOSIT PAGE

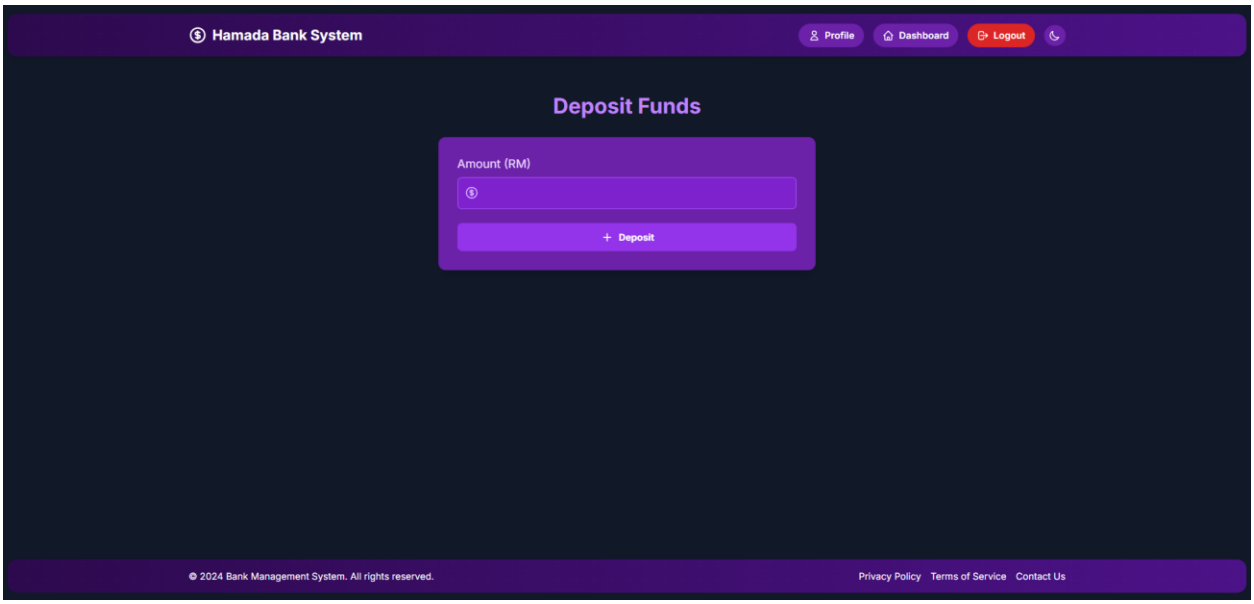


Figure 46

14.6 WITHDRAW PAGE

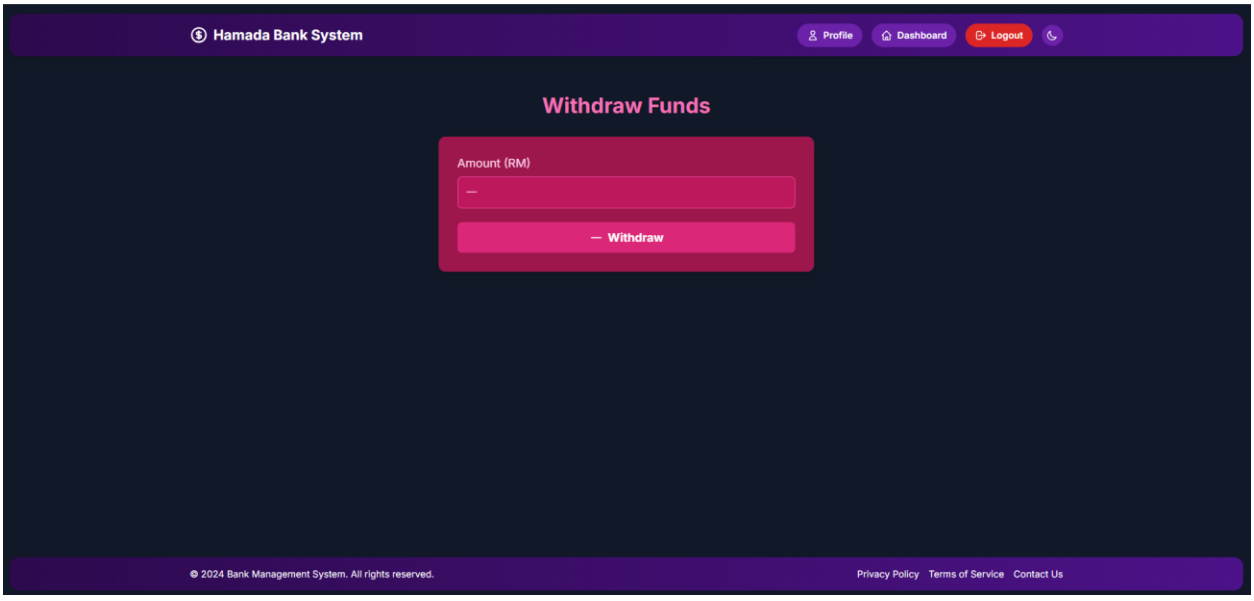


Figure 47

14.7LOAN PAGE

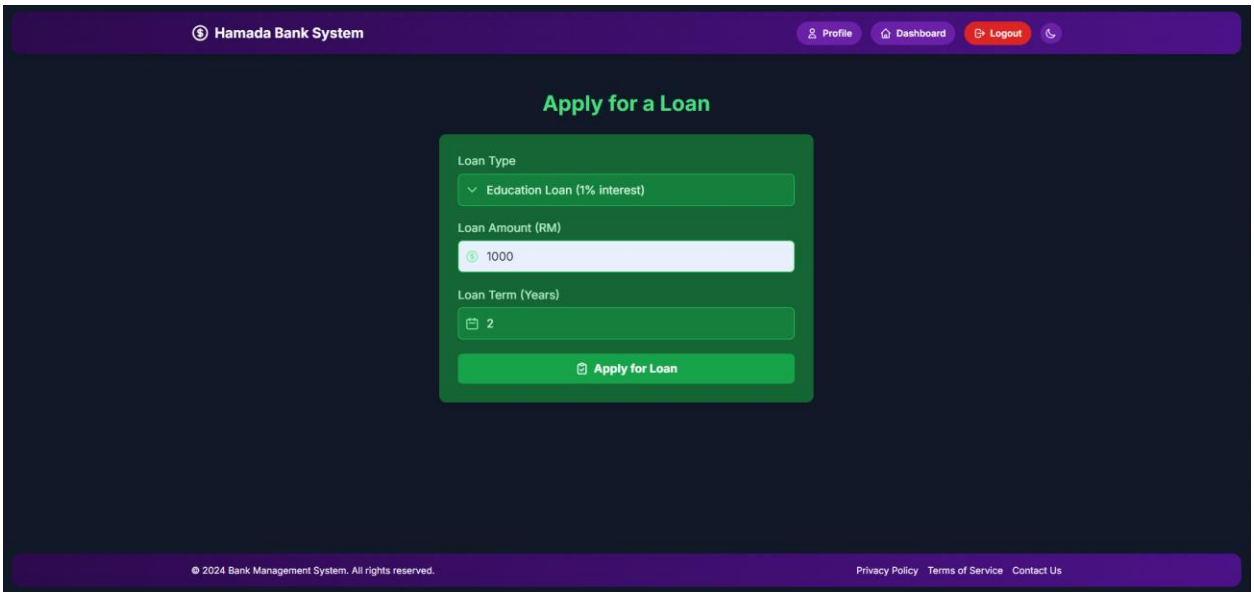


Figure 48

14.8 TRANSACTION

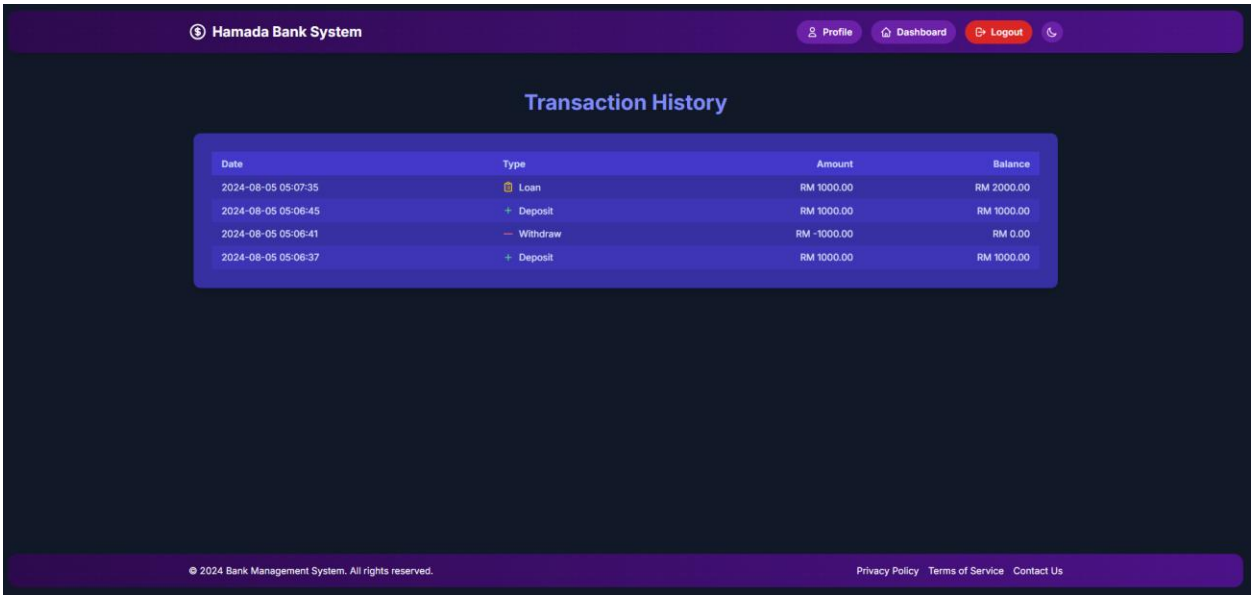


Figure 49

14.9 Admin Create a user account

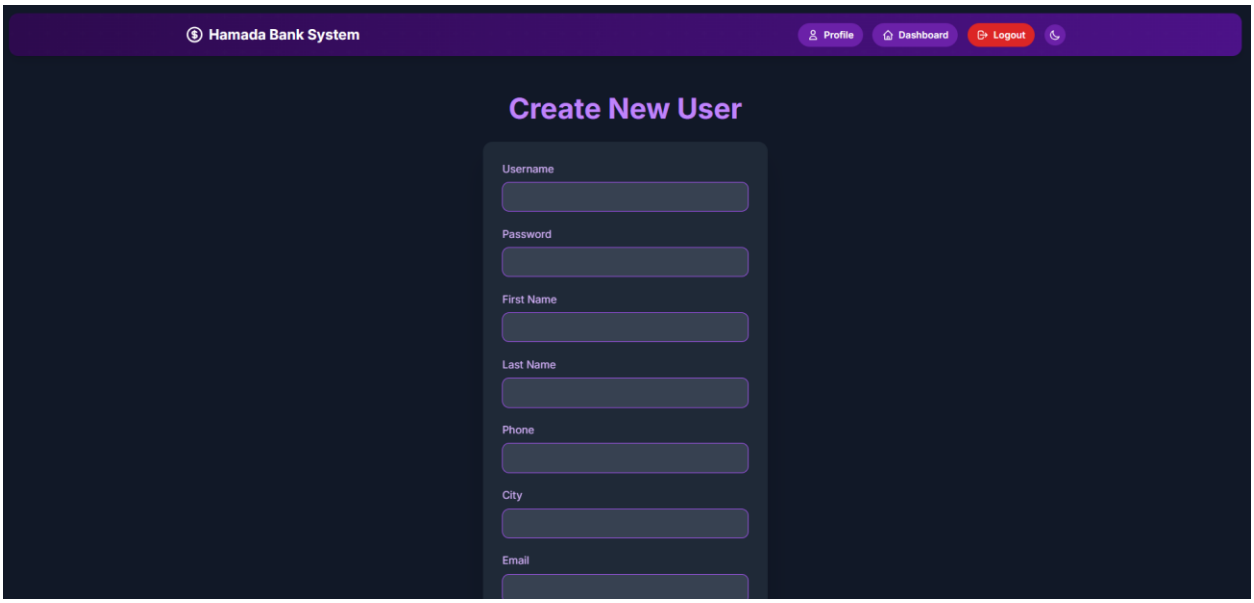


Figure 50

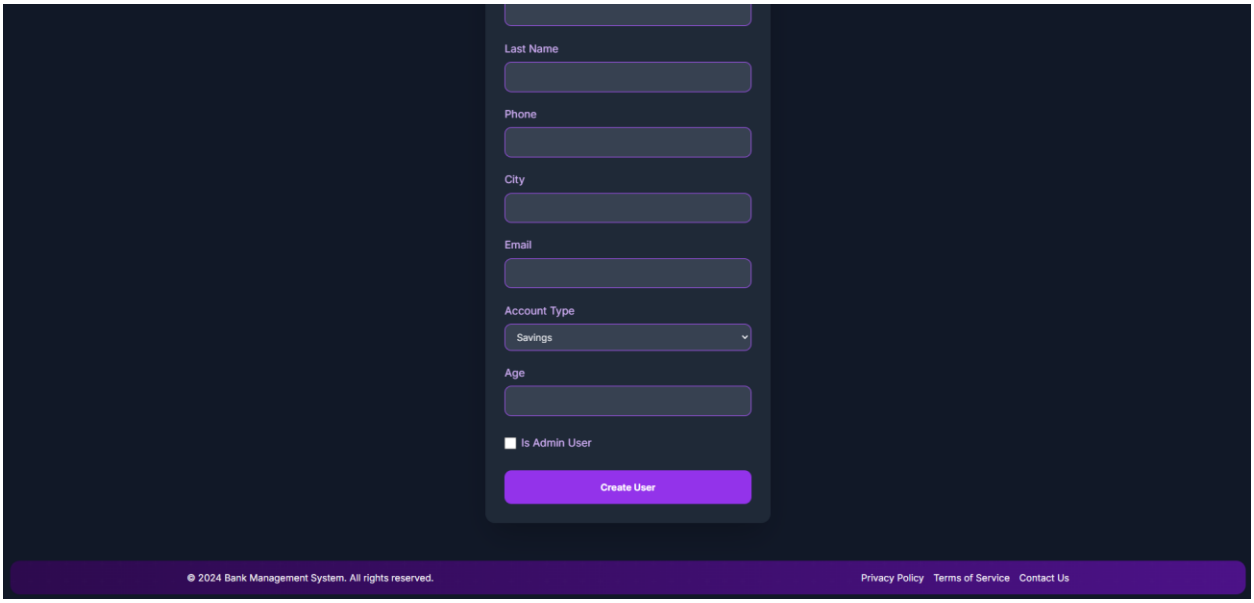


Figure 51

14.10 404 PAGE

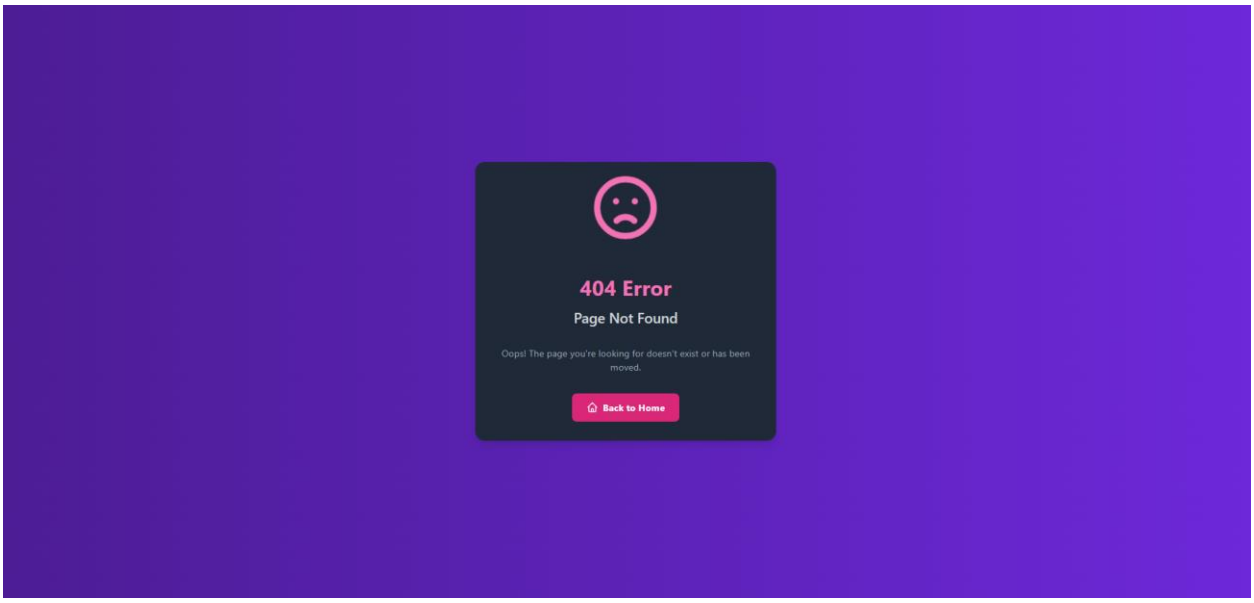


Figure 52

14.11 500 PAGE

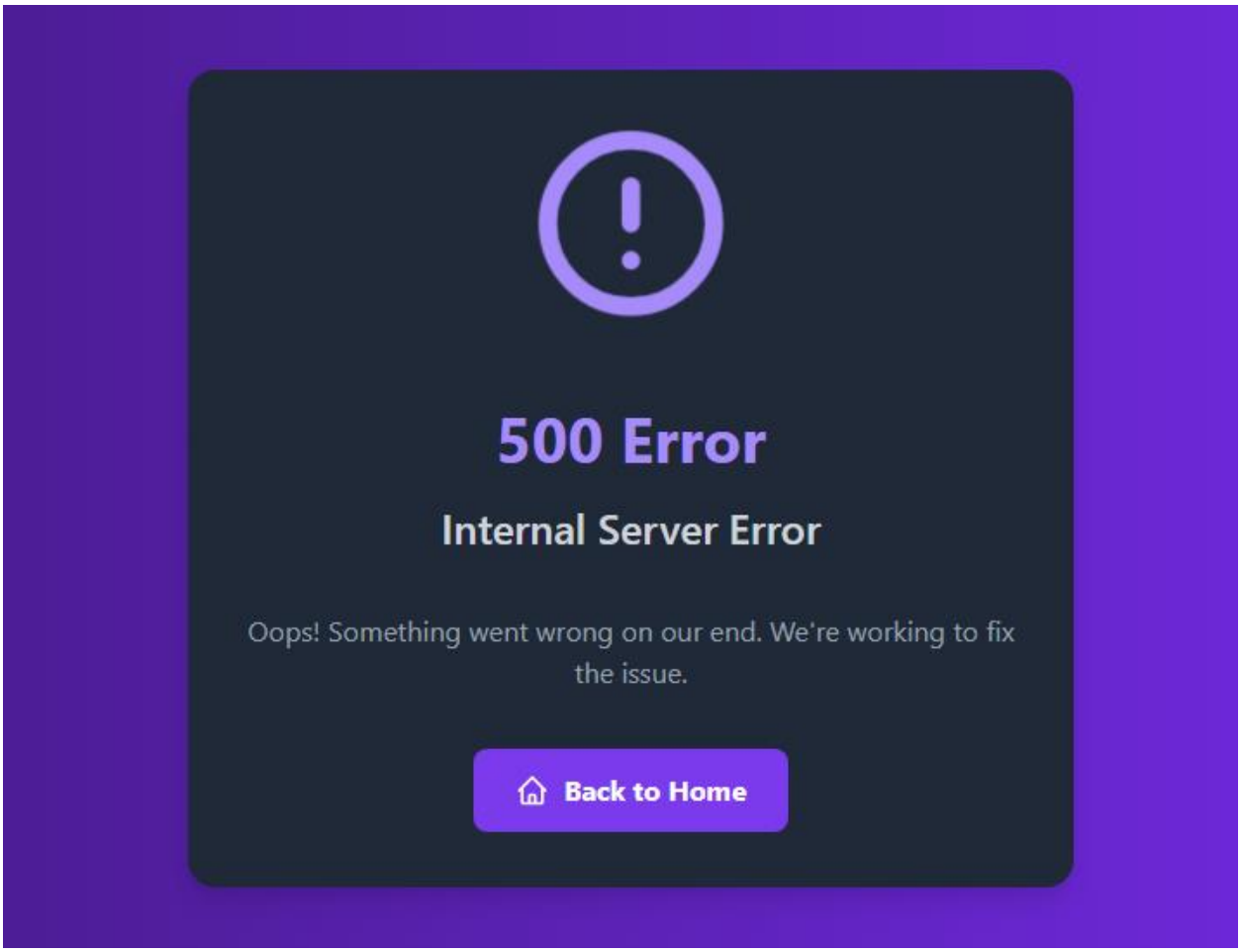


Figure 53

14.12 RATE LIMIT PAGE

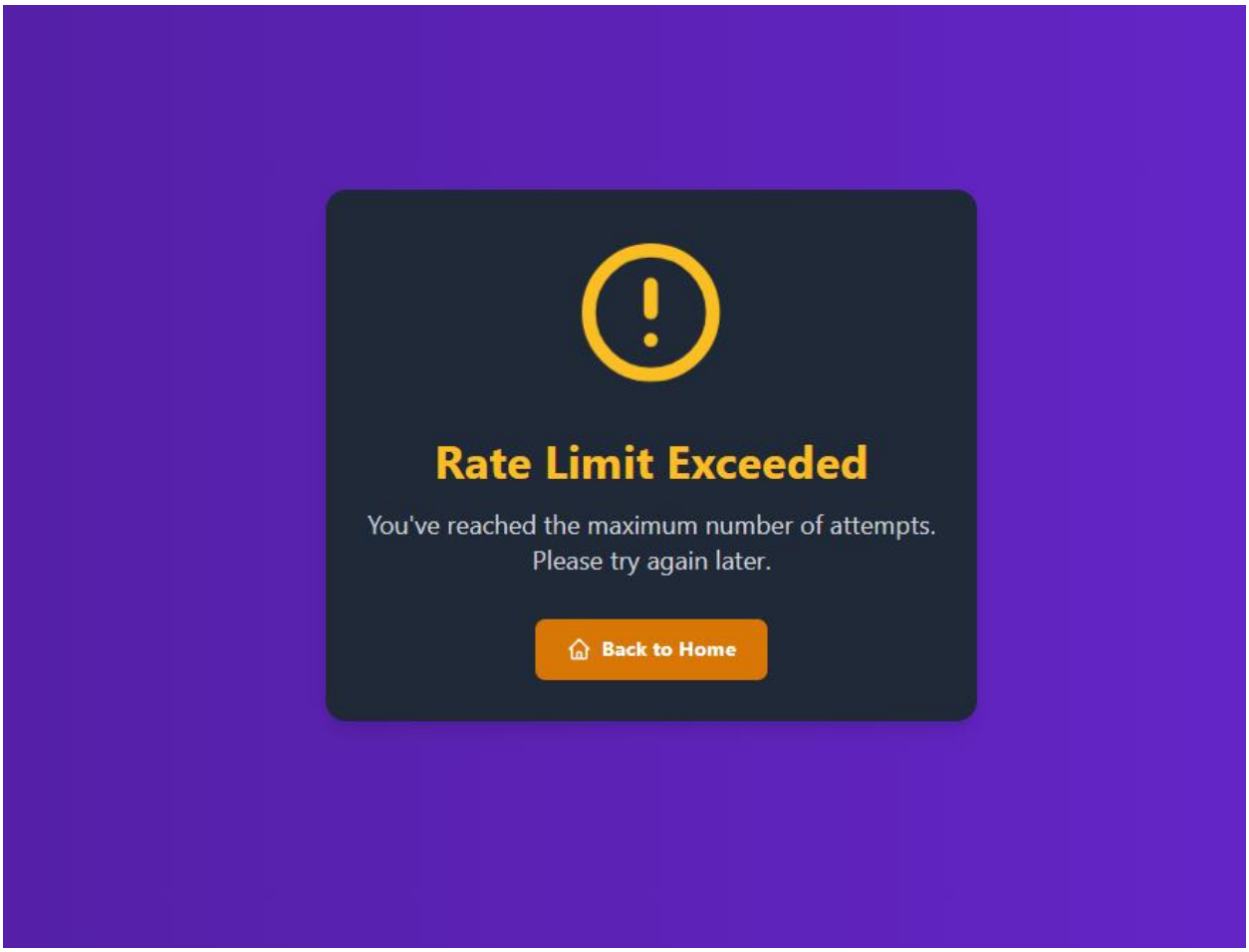
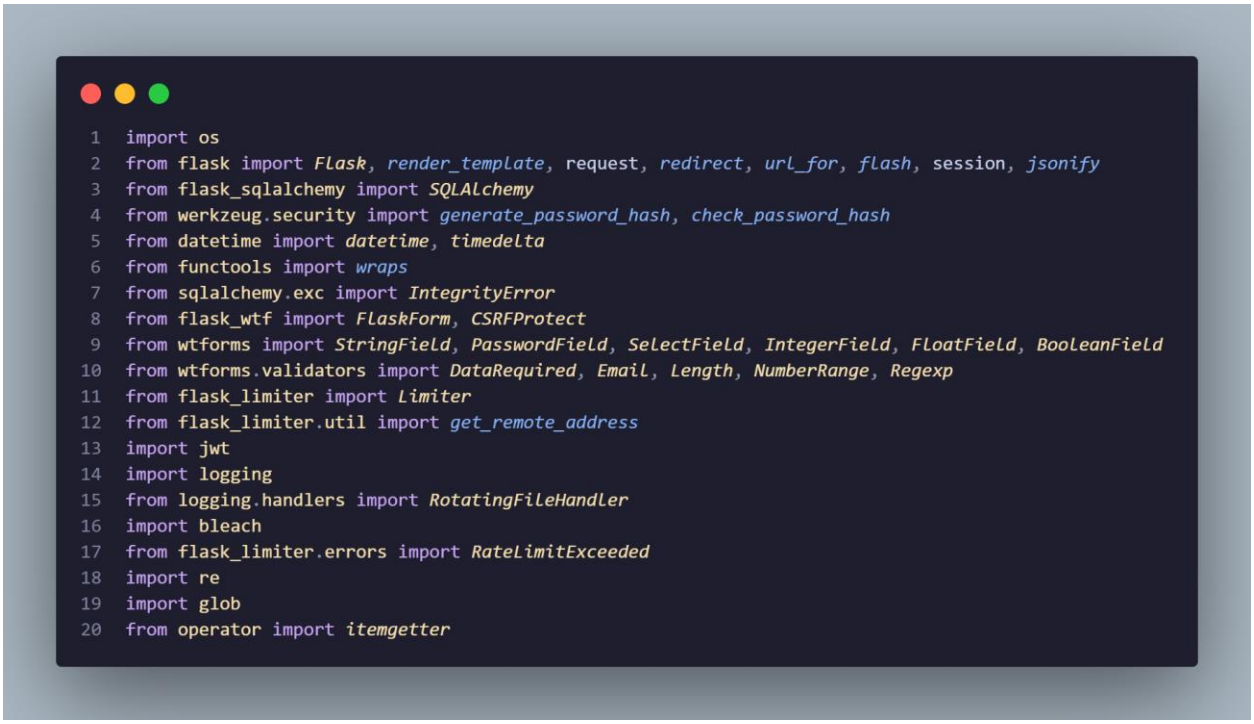


Figure 54

14.13 APPENDIX: SOURCE CODE (APP.PY)

Libraries



```
import os
from flask import Flask, render_template, request, redirect, url_for, flash, session, jsonify
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash
from datetime import datetime, timedelta
from functools import wraps
from sqlalchemy.exc import IntegrityError
from flask_wtf import FlaskForm, CSRFProtect
from wtforms import StringField, PasswordField, SelectField, IntegerField, FloatField, BooleanField
from wtforms.validators import DataRequired, Email, Length, NumberRange, Regexp
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
import jwt
import logging
from logging.handlers import RotatingFileHandler
import bleach
from flask_limiter.errors import RateLimitExceeded
import re
import glob
from operator import itemgetter
```

Figure 55

Following code consists of most of the significant modules and libraries required so that one can develop a potent web application with the programmed Flask framework. The os module is brought into the environment to work with the OS; hence, it is then capable of reading/writing files, among other things. Flask-specific imports (Flask, render_template, request, redirect, url_for, flash, session, jsonify) are pretty basic for the serving of the web server, request handling, session management, URL generation, and HTML-templating. These are the building blocks to build a dynamic web application that deals and responds intelligently to users.

It imports the module flask_sqlalchemy so as to incorporate SQLAlchemy, a potent ORM that lightens the burden on the developer by not having to write raw SQL, but Pythonic objects for the same. This not only keeps the code clean and readable, but it also lessens the possibility of SQL injection attacks. To provide more security, werkzeug.security is there to hash and verify passwords, which is really necessary in terms of safety for the credentials of the users.

The datetime module helps in storing and manipulating timestamps, and the functools module helps in constructing a new function with a collection of other functionalities from an already existing, commonly

used function; decorators are very common in any web application for tasks such as authentication and logging.

The IntegrityError comes very handy here as part of sqlalchemy.exc, as it greatly improves error handling when working with database integrity constraints. flask_wtf and wtforms provide a transparent integration of web forms with Flask and, at the same time, do not forget about many validation tools such as DataRequired and other validators to ensure the integrity and security of data.

Rate limiting is used to prevent abuse and to ensure that the application is used in a fair fashion. In this instance, incoming requests have their rate controlled using flask_limiter. This does the identification of the client by use of get_remote_address. All JSON Web Tokens are handled in this instance by the jwt library. This module ensures that information between parties is sent in secure JSON format. It is quite relevant for use in authentication and the exchange of information.

For handling logging, this application uses the logging module. The RotatingFileHandler will help avoid growing the log files without bounds, thereby putting a check on the application's performance and disk space. The Bleach library is included so that HTML is sanitized and stripped to prevent XSS attacks by rendering only safe HTML.

Finally, for re module operations related to regular expressions, the operations are very crucial in forming the patterns for matching, which in turn shall validate the given data. The items from the glob module find all pathnames that match a specified pattern, which helps in filing operations. The operator.itemgetter is helpful in sorting and efficient data retrieval. Together, they make an all-inclusive, secure environment in which to develop a scalable web application..

```python
app = Flask(__name__)
app.config['SECRET_KEY'] = os.environ.get('SECRET_KEY') or 'your_secret_key'  # Use environment variable for production
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///Hamadabank.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['JWT_SECRET_KEY'] = os.environ.get('JWT_SECRET_KEY') or 'your_jwt_secret_key'  # Use environment variable for production
app.config['JWT_ACCESS_TOKEN_EXPIRES'] = timedelta(minutes=30)  # Token expires after 30 minutes

db = SQLAlchemy(app)
csrf = CSRFProtect(app)
limiter = Limiter(key_func=get_remote_address)
limiter.init_app(app)
```

Figure 56

It's a part of the code configuring and setting up the Flask application; this forms the core of the web application. Now, the application is initialized by instantiating the Flask class with app = Flask(__name__). This instance is then used to route URLs to the handlers, manage configuration, among other things.

Several configurations have been defined here through app.config. SECRET_KEY is set from an environment variable if available, falling back to a default string. This is the key required for signing sessions and cookies. It provides integrity and confidentiality of the data. For security reasons, it's much better to set an environment variable in production.

The configuration of the database is provided with SQLALCHEMY_DATABASE_URI, and it points to an SQLite database file called Hamadabank.db. SQLite is an extremely light database suitable for development and small applications. Another configuration is the SQLALCHEMY_TRACK_MODIFICATIONS variable, which is set to False to disable modification tracking by SQLAlchemy because it can be an overhead.

Added to these configurations is a JWT_SECRET_KEY used in handling JSON Web Tokens. It fetches this key for the signing of JWTs from an environment variable or a default string, hence ensuring that the JWTs sent are original and their content has not been tampered with. The configuration JWT_ACCESS_TOKEN_EXPIRES specifies that access tokens should expire after 30 minutes. They are made more secure by tokens being valid for a short period of time.

The line db = SQLAlchemy(app) initializes the SQLAlchemy object, connecting it to the Flask application for ORM functionality, so the app is able to use Python objects to interact with the database.

The line csrf = CSRFProtect(app) protects against Cross-Site Request Forgery attacks. This extension ensures that forms in an application include a hidden tag with a CSRF token that must match with the server-side token.

Subsequently, rate limiting is set up using limiter = Limiter(key_func=get_remote_address); the Limiter object here is initialized with a key function to retrieve the client IPs. These are part of the base mechanisms for the rate limitation of requests by clients to avoid abuses and ensure each one has fair usage. Lastly, limiter.init_app(app) attaches the limit functionality to a Flask application.

Such configurations and initializations, all combined, deliver a secure and efficient base to the web application with respect to sessions, database interactions, CSRF protection, JWT handling, and rate limiting.

```python
# Set up logging
if not os.path.exists('logs'):
    os.mkdir('logs')
file_handler = RotatingFileHandler('logs/hamadabank.log', maxBytes=10240, backupCount=10)
file_handler.setFormatter(logging.Formatter(
    '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%(lineno)d]'
))
file_handler.setLevel(logging.INFO)
app.logger.addHandler(file_handler)
app.logger.setLevel(logging.INFO)
app.logger.info('Hamada Bank startup')
```

Figure 57

The following part of the code makes logging available in Flask, so important events and failures can be captured for monitoring and debugging.

This script first checks if a directory named logs exists using os.path.exists('logs'). If not, it will create the directory with os.mkdir('logs'). This will ensure all log files have a home directory, generally keeping log data organized and the project structure clean.

A RotatingFileHandler is then set up to be in charge of the log files. In this case, the RotatingFileHandler is configured to write log entries in a file called hamadabank.log inside the logs directory. It is also configured to roll over the log file when it has reached a size of 10,240 bytes (10 KB), keeping up to 10 backup files. This prevents any single log file from growing without constraint, which can consume disk space and become unwieldy to manage.

The log records are formatted with logging.Formatter. It defines what each log record looks like: timestamp (%(asctime)s), the log level (%(levelname)s), the message (%(message)s), and from where this log entry is issued—file path and line number: %(pathname)s:%(lineno)d. This full format provides maximum information for every log record.

What that means with file_handler.setLevel(logging.INFO), using file_handler, only log entries at level INFO and higher are handled, so what it means is that all the messages at INFO, WARNING, ERROR, and CRITICAL will be captured in logs, and the DEBUG entries will be filtered out.

Add the file_handler to the Flask application's logger—app.logger—with the invocation of app.logger.addHandler(file_handler). Also set the overall log level for the logger to INFO using app.logger.setLevel(logging.INFO), ensuring all log handlers associated with the application have the same filtering applied.

Finally, a first log entry is made with app.logger.info('Hamada Bank startup'). This logs a message that the application has started up; it is good for verification of proper logging from the very start.

All in all, this setup gives a robust way of capturing and managing log data. This is very important in diagnosing issues, monitoring the performance of applications, and security aspects..

```python
def recreate_database():
    # Drop all tables
    db.drop_all()

    # Create all tables
    db.create_all()

    # Create admin user
    admin_password = generate_password_hash('admin123')  # You should change this password
    admin = User(username='admin', password=admin_password, first_name='Admin',
                 last_name='User', phone='1234567890', city='Admin City',
                 email='admin@example.com', account_type='admin', age=30, is_admin=True)
    db.session.add(admin)
    db.session.commit()
    print("Database recreated and admin user added.")
```

Figure 58

The following part of the code defines a function called recreate_database. This function is responsible for resetting the database by dropping all the existing tables and then recreating them. It also adds an initial admin user in the database. This function is useful during development or testing phases when often the database schema or the initial data needs to be reset.

The function starts by dropping all the tables in the database with db.drop_all(). This removes all data and schema from it, giving a clean slate.

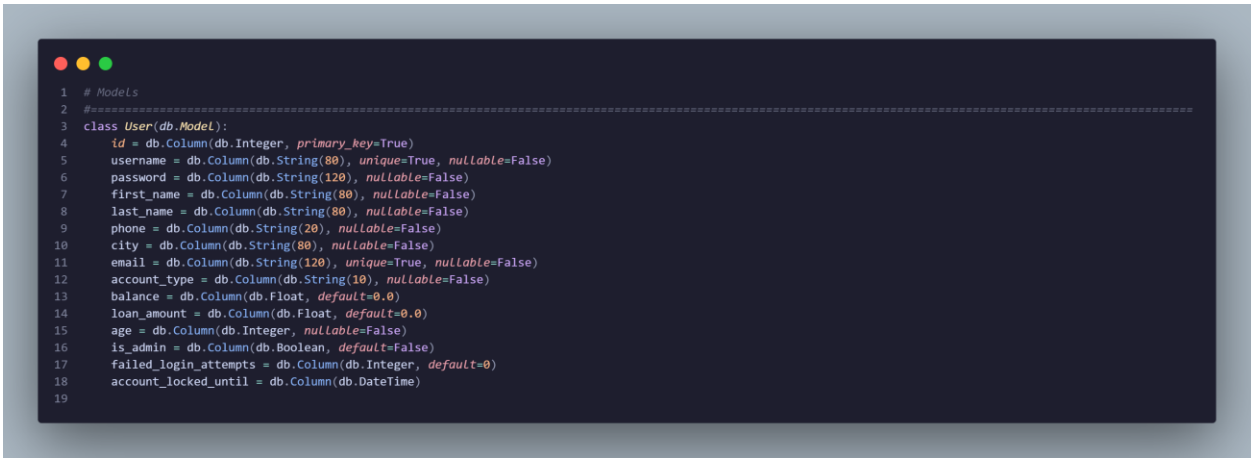
Then, create all tables based on the current database models with db.create_all(). This step will recreate the database schema as per the models defined in the application and ensure the structure of the database is current.

It then creates an admin user with predefined features, where the password is well encoded in the database using generate_password_hash('admin123'). The details of this admin user are defined—the username, first name, last name, phone number, city, email, account type, age, and whether he is an admin. The is_admin attribute is set to True to identify this user as administrative.

The newly created Admin user object is added to the database session with db.session.add(admin). This puts the user object into a pending state, ready to be inserted into the database. Finally, all the changes are committed to the database with db.session.commit(), which concludes the creation of the admin user.

A message is then printed to the console with print("Database recreated and admin user added."), indicating that the database has been reset successfully and the admin user added. This message will provide feedback to the developer or administrator running the function.

In general, the recreate_database function turns into an important utility that does not only run a development or testing server but also makes sure that the database schema is set up correctly, including the setting up of an initial admin user to log into the application..



```python
# Models
#================================================================================================================
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password = db.Column(db.String(120), nullable=False)
    first_name = db.Column(db.String(80), nullable=False)
    last_name = db.Column(db.String(80), nullable=False)
    phone = db.Column(db.String(20), nullable=False)
    city = db.Column(db.String(80), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    account_type = db.Column(db.String(10), nullable=False)
    balance = db.Column(db.Float, default=0.0)
    loan_amount = db.Column(db.Float, default=0.0)
    age = db.Column(db.Integer, nullable=False)
    is_admin = db.Column(db.Boolean, default=False)
    failed_login_attempts = db.Column(db.Integer, default=0)
    account_locked_until = db.Column(db.DateTime)

```

Figure 59

The code here defines a User model using the ORM capabilities of SQLAlchemy to model a user within the system. So, every class attribute relates to a column in the database table; structured and consistent data storage is, therefore, ensured.

It has a number of important fields in the User model. The id field is the primary key, which uniquely identifies each user and enables efficient lookups. Both the username and email fields cannot be null and are unique, therefore averting duplicate accounts and ensuring every user is uniquely identifiable within the system. The last one is a password field storing users' hashed passwords for safety measures.

It also includes personal details like first_name, last_name, phone, city, and age. All these fields are very important to identify the user and personalize the user experience inside the application. The account_type field enables the differentiation between different types of users: normal and administrator users. This allows the application to implement role-based access control.

It deals with the financial information with the fields balance and loan_amount, where the former will hold the user's account balance and the latter stores the amount they have lent. These fields are quite useful in managing user accounts and financial transactions effectively.

The is_admin, failed_login_attempts, and account_locked_until fields embed security features. A user could be an administrator; the system provides access to the user based on his/her level of authorization as an administrator. Unsuccessful login attempts increment the failed_login_attempts counter; the account is locked out for a certain account_lockout time in case the limit of failed attempts is exceeded. Features like these prevent unauthorized access or other possible hacking of user accounts.

In all, the User model represents the user comprehensively within an application through personal information, details of accounts, and other safety measures. Dealing with data in such an organized way is instrumental to the integrity, security, and functionality of the user-associated aspects of any system..

```python
class Transaction(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    transaction_type = db.Column(db.String(20), nullable=False)
    amount = db.Column(db.Float, nullable=False)
    balance = db.Column(db.Float, nullable=False)
    date = db.Column(db.DateTime, default=datetime.utcnow)
    hash = db.Column(db.String(64), nullable=False)  # For integrity checking
```

Figure 60

The next part defines a Transaction model against the ORM features provided by SQLAlchemy. The class will be an exact model for any financial transaction in the system. The Transaction class is then mapped to a database table where each attribute corresponds to a column, thereby keeping every record of the transactions in a structured and consistent way.

It has an id field that serves as the primary key in uniquely identifying every instance of a transaction and facilitates efficient lookups. The user_id field is a foreign key referring to the id column in the User table, which establishes the relationship between the transaction and the user who made it. This link is important in keeping track of which transactions belong to which users.

It contains transaction type, either deposit or withdrawal. This is important in classifying and handling various types of financial transactions. The amount field holds the monetary value involved in the transaction. The Balance field captures the account balance of the user immediately after the transaction. These fields are critical in maintaining correct financial records and displaying current information of accounts to the users.

It provides a date field, timestamping when the transaction took place; it is set to default to the current date and time using datetime.utcnow. This allows chronological tracking and auditing of transactions.

The hash field is used for integrity checking. It contains a unique hash value that will be used to verify the authenticity and integrity of transaction data. That will add more security to make sure not to tinker with the record of the transaction.

The Transaction model ensures that the system fully represents a financial transaction and secures it. Basically, this class ensures that each transaction is recorded and associated with the correct user, classified by type, and time-stamped. Moreover, integrity regarding the transactional data is guaranteed, thereby safeguarding trust and security of the financial activities of the application..

```
class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(min=3, max=15)])
    password = PasswordField('Password', validators=[DataRequired(), Length(min=8, max=15)])
    first_name = StringField('First Name', validators=[DataRequired(), Length(max=15)])
    last_name = StringField('Last Name', validators=[DataRequired(), Length(max=15)])
    phone = StringField('Phone', validators=[DataRequired(), Regexp('^[0-9]{10,15}$', message='Phone number must be between 10 and 15 digits')])
    city = StringField('City', validators=[DataRequired(), Length(max=15)])
    email = StringField('Email', validators=[DataRequired(), Email(), Length(max=120)])
    account_type = SelectField('Account Type', choices=[('savings', 'Savings'), ('current', 'Current'), ('islamic', 'Islamic')])
    age = IntegerField('Age', validators=[DataRequired(), NumberRange(min=18, max=120)])

```

Figure 61

This code defines a RegistrationForm class. It will be used with Flask-WTF, an extension for Flask that bridges Flask with WTForms, a module for handling and validating web forms. The form that the new users will be using to register an account is then represented. The form, by defining all its fields with certain validators, makes sure that the data a user inputs into any particular field has passed the required criteria and thus ensures enhanced integrity and security of data.

The fields in the form are for username, password, first_name, last_name, phone, city, email, account_type, and age. Each field represents the attribute the system is going to capture when registering a user. Each of the fields is then subjected to validators in order to put some rule enforcing and restrictions on the input data. For instance, the field of the username shall be between 3 and 15 characters to prevent very short or extremely long usernames. The password field should contain between 8 and 15 characters. This will ensure that the password used by you is strong for security.

Personal details include first_name, last_name, phone, city, and email. All these are captured with fields that not only make sure the input is provided but also meets certain length/format criteria. The phone field uses a regular expression to check that the input is a string of digits between 10 and 15 characters long, thus ensuring valid phone numbers. The email field takes an email validator that provides it with checks to ensure that the input is a valid email address and that it does not contain more than 120 characters.

The account_type field is a select field providing choices like 'savings', 'current', 'islamic', to ensure users select a type of account. This is important for standardization on the type of accounts that can be registered within the system. The age field is an integer field with a validator ensuring that the user is between 18 to 120 years old, hence ensuring the user is within the proper age to register for an account.

This RegistrationForm class plays a very significant role in capturing, and then validating, the user input coming in during the course of registration. It ensures that all necessary details are gathered in a

standardized and secure manner. By data validation, it ensures the integrity of the system's data, avoiding wrong and malicious input, and giving the user an easier time in the registration process..

```python
class AdminCreateUserForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(min=3, max=15)])
    password = PasswordField('Password', validators=[DataRequired(), Length(min=8, max=15)])
    first_name = StringField('First Name', validators=[DataRequired(), Length(max=15)])
    last_name = StringField('Last Name', validators=[DataRequired(), Length(max=15)])
    phone = StringField('Phone', validators=[DataRequired(), Regexp('^[0-9]{10,15}$', message='Phone number must be between 10 and 15 digits')])
    city = StringField('City', validators=[DataRequired(), Length(max=15)])
    email = StringField('Email', validators=[DataRequired(), Email(), Length(max=120)])
    account_type = SelectField('Account Type', choices=[('savings', 'Savings'), ('current', 'Current'), ('islamic', 'Islamic')])
    age = IntegerField('Age', validators=[DataRequired(), NumberRange(min=18, max=120)])
    is_admin = BooleanField('Is Admin')

```

Figure 62

The next line of the code defines a class, AdminCreateUserForm, using Flask-WTF. Flask-WTF bridges Flask with WTForms, a effective way of handling and validating web forms. The AdminCreateUserForm is tailored to enable administrators to create new user accounts within the application. This form ensures that data input by the admin meets the criteria required, hence improving data integrity and security.

It has fields for username, password, first_name, last_name, phone, city, email, account_type, age, and is_admin. Each field points to an attribute the system needs to capture when an admin creates a new user. Validators are applied to these fields to enforce rules and restrictions on the input data. For example, this forces the username to be between 3 and 15 characters in length, thus guaranteeing that usernames are neither too short nor too long. The password field will have to be between 8 and 15 characters long, thus compelling the use of strong passwords for security.

Personal information, such as first_name, last_name, phone, city, and email, is captured with fields that make sure input is provided and of a certain length/format. For example, it uses a regular expression to check that the phone contains a string of digits between 10 and 15 characters long, hence guaranteeing that the phone number is valid. The email field applies an email validator so that the input is a valid email address and not longer than 120 characters.

The account_type field is a select field with predefined choices such as 'savings', 'current', 'islamic'. This will enforce the admin to pick a valid account type for the user. It also sorts out the kinds of accounts that may be created within the system. The age field is an integer field with a validator, ensuring that the user's age is between 18 and 120 years, thus ensuring the proper age of the user during registration.

Another field, is_admin, will be a boolean field by which the admin will be able to tell whether the new user will be an admin or not. This field is very instrumental in determining the roles of users within the system so that only relevant and authorized users will have access to admin functionalities.

The AdminCreateUserForm class is instrumental in capturing and validating user input during the process of user creation by an administrator. This guarantees that all relevant data is collected in a unified and secure way. By enforcing data validation, it helps in the maintenance of integrity for the system's data and prevents false or malicious input. It provides a structured way for the administrator to efficiently create a new user account with accuracy..

```
def generate_token(user_id):
    return jwt.encode({'user_id': user_id, 'exp': datetime.utcnow() + timedelta(minutes=30)},
                      app.config['JWT_SECRET_KEY'], algorithm='HS256')
```

Figure 63

The generate_token function creates a JSON Web Token for a user. JSON Web Tokens are very well-known due to their use in secure information exchange and are especially helpful in authentication purposes in web applications.

The function takes a user_id as an argument, encodes it into a JWT along with an expiry time that is 30 minutes from the current time. This indicates that the token is valid only for the period of time, improving the security, and, in case it gets compromised, this will reduce the risk of its misuse.

The jwt.encode method will generate the token. This token includes a payload that contains information about a user_id and an expiration time, exp. The datetime.utcnow() + timedelta(minutes=30) adds the expiration time to 30 minutes from the current UTC time. This ensures that the token will have expired within 30 minutes of its issue.

This token will be signed with a secret key, which is defined in the configuration of your application by app.config['JWT_SECRET_KEY'], and the HS256 algorithm, HMAC-SHA256. The secret key thus ensures that the token can only be generated or validated by the server itself; otherwise, it can't let any tampering go through.

The generate_token function is the underpinning of full user authentication. It safely generates tokens for the authentication of a user's identity, therefore granting access to resources protected by an application. Including an expiration time in this token does further improve its security by limiting the window of opportunity if the token is misused..

Figure 64

The function verify_token is in charge of decoding and verifying a JSON Web Token, which is very critical to user authentication. This makes sure that the tokens used are valid and untampered.

This function takes a token as an argument and attempts to decode it using jwt.decode. It uses the secret key set up in the configuration of the application—app.config['JWT_SECRET_KEY']—and the HS256 algorithm to check the integrity and authenticity of the token. Assuming the token is valid, extract the payload contained within the token. It returns the user_id that identifies the user to whom the token belongs.

The decoding process is enclosed within a try block for handling possible exceptions. In case of an invalid token, a token that has expired, or a token that was tempered with in any way will raise an exception in jwt.decode method. The function catches the exception and returns None in these cases that indicates token verification failure.

The verify_token function is critical to the app's security in ensuring that only valid tokens are accepted so that authenticated users can gain access to protected resources; otherwise, it will return None so as not to grant unauthorized access, therefore maintaining the integrity of the authentication system. The line is central in the security infrastructure by checking the correctness of user identities..

```python
def token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = session.get('token')
        if not token:
            flash('You need to log in to access this page.', 'error')
            return redirect(url_for('login'))
        try:
            data = jwt.decode(token, app.config['JWT_SECRET_KEY'], algorithms=['HS256'])
            current_user = User.query.get(data['user_id'])
        except:
            flash('Invalid session. Please log in again.', 'error')
            return redirect(url_for('login'))
        return f(current_user, *args, **kwargs)
    return decorated
```

Figure 65

This decorator function, token_required, shall be the decorator that ensures a user is authenticated to access certain routes in the Flask application. It checks the identity of a user through JWTs. Additionally, handling will be done for a session of the user.

The decorator function itself is decorated using @wraps(f) from the functools module, which is used in preserving metadata of the original function. A function f gets wrapped in the inner function decorated.

Within decorated, it first attempts to get the token from the session using session.get('token'). If this doesn't return a token, then the user isn't logged in, so the user is flashed with a message to log in and redirected to the login page.

In case of a token, it will try to decode it using jwt.decode. Here, the same secret key is used that was configured in app.config by 'JWT_SECRET_KEY', and also the HS256 algorithm. Provided the decoding was successful, it fetches the user ID from the payload of the token and queries the database for its corresponding User object, assigning that to current_user.

In case of either an invalid token or decoding failure, it catches the exception and flashes an error message that the session is invalid, then redirects the user back to the login page.

If the token is valid and a user is authenticated, then the original function f gets called with current_user and any extra arguments and keyword arguments passed in with *args, **kwargs, thus allowing access to the authenticated user's information on that decorated route.

The token_required decorator acts as a guard for routes that are protected by authentication. So, only users who hold valid tokens and are authenticated can access these routes. It highly improves the security of the application by avoiding any unauthorized access. This class manages the session of the user and verifies tokens to maintain the integrity and confidentiality of a user's session..

```
@app.errorhandler(RateLimitExceeded)
def handle_rate_limit_exceeded(e):
    return render_template('rate_limit_exceeded.html'), 429

#=============================================================================================================================================

class DepositForm(FlaskForm):
    amount = FloatField('Amount', validators=[DataRequired(), NumberRange(min=0.01, max=1000000)])



#=============================================================================================================================================

class WithdrawForm(FlaskForm):
    amount = FloatField('Amount', validators=[DataRequired(), NumberRange(min=0.01, max=1000000)])
```

Figure 66

This portion of the code snippet provides mission-critical functionality to the application handling errors and financial transactions to ensure security and user experience are well managed.

The first section defines an error handler for rate limiting. The @app.errorhandler(RateLimitExceeded) decorator registers the handle_rate_limit_exceeded function to handle cases of the rate limit being exceeded. This will be called when a user creates too many requests in a very short period (in this case, it will render a rate_limit_exceeded.html template and return a 429 status code). It protects an application against possible abusive situations and provides for reasonable use by regulating the frequency of incoming requests. Also, it will improve the user's experience by sending a clear message and status code that let the user know why their request has been blocked.

The second section defines a class, DepositForm, using Flask-WTF. This form handles deposit transactions and includes one field: amount. The FloatField is validated to not allow the field to be left empty with DataRequired, and NumberRange enforces that the deposit amount should lie between 0.01 and 1,000,000. This validation is very important for maintaining the integrity of your data and avoiding potential errors or malicious input. It ensures that only legal and meaningful deposit amounts are processed by forcing these constraints, improving security and reliability.

The third section defines a class, WithdrawForm, that handles withdrawal transactions. It has got one field, amount, which uses the same validators as DepositForm. Validation allows making sure that no withdrawal request with either a too small or too large value is received and processed by the application; hence, the protection of the application against invalid or excessive withdrawal attempts is very critical to the financial stability and security of the application's operations.

All these components contribute to the security and usability of the application as a whole. The rate limit error handler prevents abuse and establishes a fair share of access. Deposit and withdrawal forms keep tight

validation rules to maintain integrity in financial transactions. Structuring and validation by forms will prevent a number of errors and various malicious activities from misusing the application, displacing its smooth and secure operation..

```python
class LoanForm(FlaskForm):
    amount = FloatField('Amount', validators=[DataRequired(), NumberRange(min=100, max=1000000)])
    years = IntegerField('Years', validators=[DataRequired(), NumberRange(min=1, max=30)])
    loan_type = SelectField('Loan Type', choices=[('education', 'Education'), ('car', 'Car'), ('home', 'Home'), ('personal', 'Personal')])



#=============================================================================================================================================

def admin_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        token = session.get('token')
        if not token:
            flash('You do not have permission to access this page.', 'error')
            return redirect(url_for('login'))
        try:
            data = jwt.decode(token, app.config['JWT_SECRET_KEY'], algorithms=['HS256'])
            user = User.query.get(data['user_id'])
            if user:
                print(f"Admin access granted for user: {user.username}")
            if not user or not user.is_admin:
                flash('You do not have permission to access this page.', 'error')
                return redirect(url_for('dashboard'))
        except Exception as e:
            print(f"Admin access denied: {e}")
            flash('You do not have permission to access this page.', 'error')
            return redirect(url_for('login'))
        return f(*args, **kwargs)
    return decorated_function
```

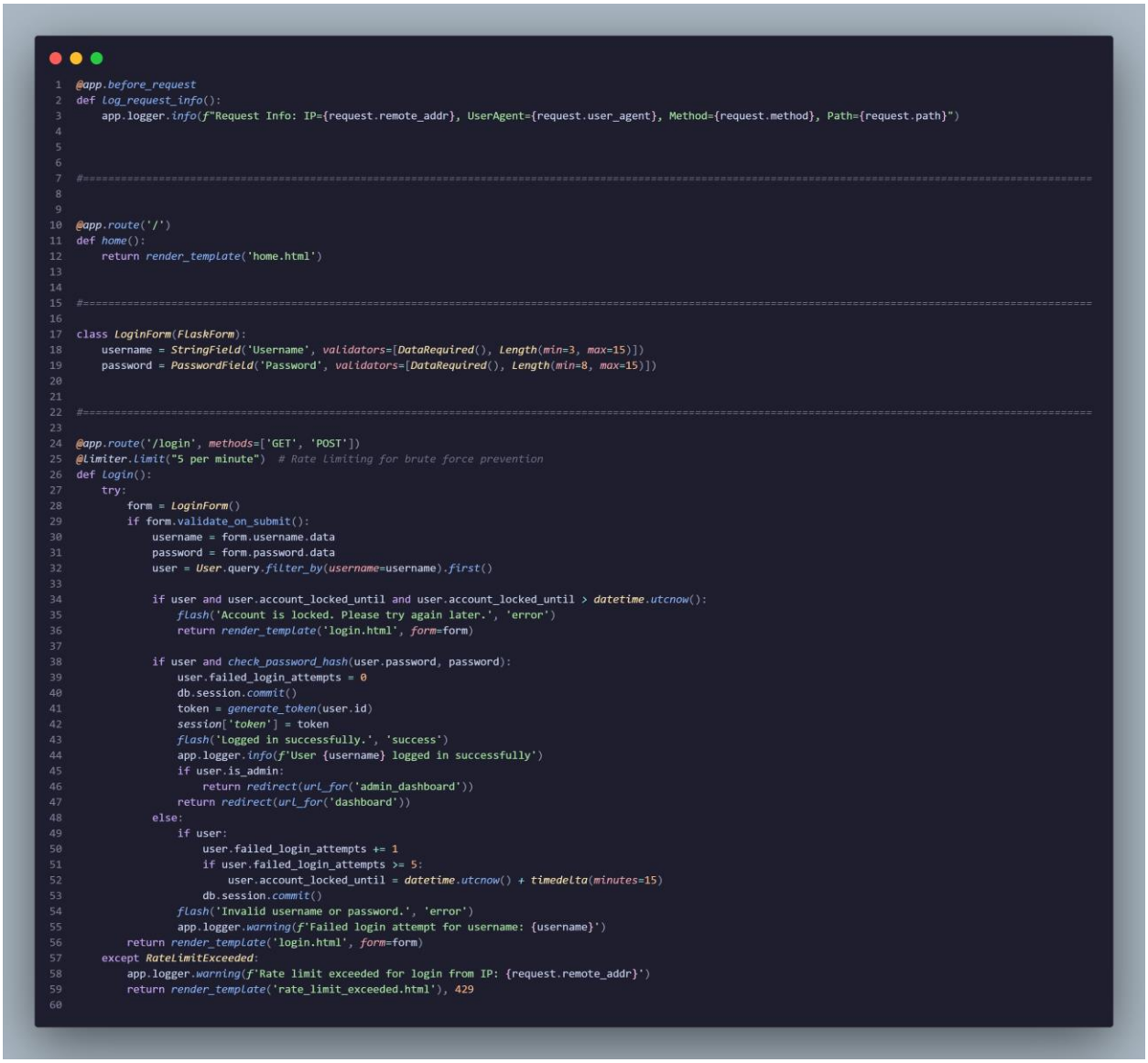
Figure 67

This is a code snippet that includes a form class definition to handle loan requests and a decorator to ensure that some routes in the application are accessed only by admin users.

Above, the class definition LoanForm uses Flask-WTF to handle submissions regarding a loan application. There are three fields in this form: amount, years, and loan_type. The amount field is a FloatField with validators in place to ensure that an amount is provided and falls between 100 and 1,000,000 to avoid unreasonable loan requests. There's an IntegerField for years, with validators that guarantee the length of the loan will lie between 1 and 30 years, so the conditions for a loan are real and endurable. The loan_type field is a SelectField offering predefined choices like 'education', 'car', 'home', 'personal', etc., to make the type of loan clear. This is to validate the integrity of the loan application and allow for every exigency of loan requests to fall within the acceptable parameters.

The admin_required decorator function forces some routes to be available only to users who have administrative privileges. It wraps the target function, checking for a valid token in the session. If none is found, an error message is flashed to the user, and they are redirected back to the login page. If a token exists, decode it using the secret key set up in the application configuration file. It retrieves user information from the token and makes a database query for the user. If the user exists and the user is an admin, then access is granted. Otherwise, a flash message will appear with an error, and the user will be redirected to the dashboard or the login page. This decorator is fundamental to the security of admin routes, ensuring only those authorized have the ability for sensitive operations.

These elements are thus important in maintaining the security and functionality of the application. Class LoanForm makes sure that a loan application is valid and within reasonable limits to protect the financial integrity of the system. The admin_required decorator secures the administrative routes; hence, it allows only licensed people to have access and perform sensitive operations. This structured approach helps in maintaining a secure, reliable, and user-friendly application..



```python
@app.before_request
def log_request_info():
    app.logger.info(f"Request Info: IP={request.remote_addr}, UserAgent={request.user_agent}, Method={request.method}, Path={request.path}")



#=============================================================================================================================================


@app.route('/')
def home():
    return render_template('home.html')


#=============================================================================================================================================

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(min=3, max=15)])
    password = PasswordField('Password', validators=[DataRequired(), Length(min=8, max=15)])


#=============================================================================================================================================

@app.route('/login', methods=['GET', 'POST'])
@limiter.limit("5 per minute")  # Rate limiting for brute force prevention
def login():
    try:
        form = LoginForm()
        if form.validate_on_submit():
            username = form.username.data
            password = form.password.data
            user = User.query.filter_by(username=username).first()

            if user and user.account_locked_until and user.account_locked_until > datetime.utcnow():
                flash('Account is locked. Please try again later.', 'error')
                return render_template('login.html', form=form)

            if user and check_password_hash(user.password, password):
                user.failed_login_attempts = 0
                db.session.commit()
                token = generate_token(user.id)
                session['token'] = token
                flash('Logged in successfully.', 'success')
                app.logger.info(f'User {username} logged in successfully')
                if user.is_admin:
                    return redirect(url_for('admin_dashboard'))
                return redirect(url_for('dashboard'))
            else:
                if user:
                    user.failed_login_attempts += 1
                    if user.failed_login_attempts >= 5:
                        user.account_locked_until = datetime.utcnow() + timedelta(minutes=15)
                    db.session.commit()
                flash('Invalid username or password.', 'error')
                app.logger.warning(f'Failed login attempt for username: {username}')
        return render_template('login.html', form=form)
    except RateLimitExceeded:
        app.logger.warning(f'Rate limit exceeded for login from IP: {request.remote_addr}')
        return render_template('rate_limit_exceeded.html'), 429

```

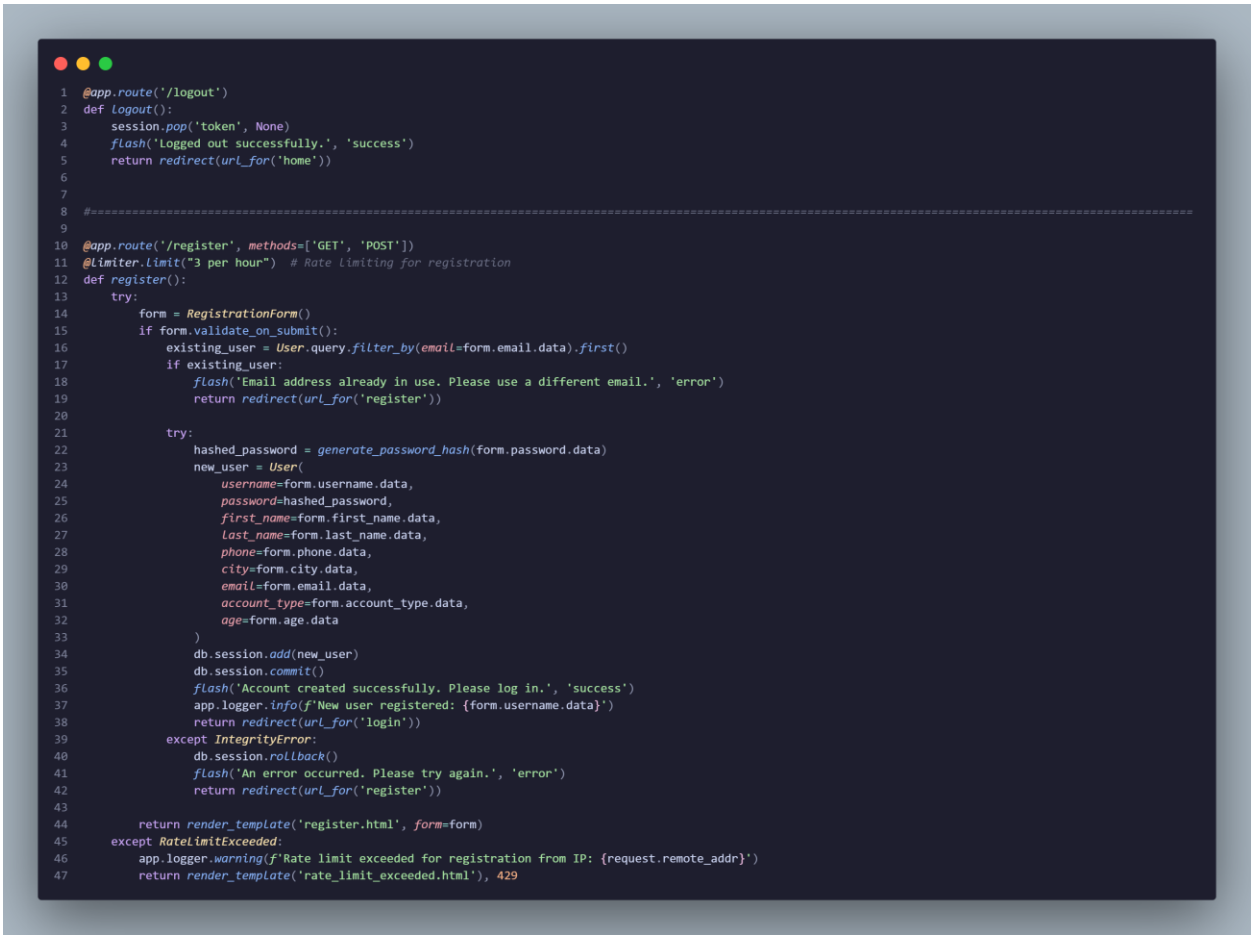
Figure 68

Above is a code snippet including a form class that handles loan requests and a decorator that enforces access to routes within the application only to admin users.

Defined here is the class LoanForm, which utilizes Flask-WTF for submission handling of a loan application. It has three fields: amount, years, and loan_type. The amount field is a FloatField with validators to ensure an amount exists and it will be in the range of 100 through 1,000,000; that is, to avoid silly loan requests. It has an IntegerField with validators, thus it falls between 1 and 30 years, so the loan terms are much realistic and manageable. The loan_type field is a SelectField offering predefined choices, making sure that the type of loan is well specified, such as 'education', 'car', 'home', and 'personal'. These are significant cross-checks in maintaining the integrity of the process with respect to loan applications and in ensuring that all requests for loans remain within acceptable parameters.

The admin_required decorator function is used to ensure that only users who are administrators have access to certain routes. It wraps the target function, checking for a valid token to be present in the session. If not, it flashes an error message to the user and redirects to the login page. If a token is available, decode it using the secret key set in the application configuration. Extract user data from the token and query the database for the user. If the user exists and he/she is an admin, allow access; otherwise, flash an error message and redirect the user to the dashboard or login page. This decorator is critical to lock down administrative routes so that only properly authorized users can perform sensitive operations.

These are components that ensure the application is tight and secure in general. The LoanForm class guarantees the validity and reasonability of loan applications, thereby protecting the financial integrity of the system. The admin_required decorator keeps the administrative routes secure and, therefore, only allows certain types of users to view and make sensitive operations. With this structured approach, the application will be secure, reliable, and friendly to the end user..



```python
@app.route('/logout')
def logout():
    session.pop('token', None)
    flash('Logged out successfully.', 'success')
    return redirect(url_for('home'))


#=============================================================================================================================================

@app.route('/register', methods=['GET', 'POST'])
@limiter.limit("3 per hour")  # Rate limiting for registration
def register():
    try:
        form = RegistrationForm()
        if form.validate_on_submit():
            existing_user = User.query.filter_by(email=form.email.data).first()
            if existing_user:
                flash('Email address already in use. Please use a different email.', 'error')
                return redirect(url_for('register'))

            try:
                hashed_password = generate_password_hash(form.password.data)
                new_user = User(
                    username=form.username.data,
                    password=hashed_password,
                    first_name=form.first_name.data,
                    last_name=form.last_name.data,
                    phone=form.phone.data,
                    city=form.city.data,
                    email=form.email.data,
                    account_type=form.account_type.data,
                    age=form.age.data
                )
                db.session.add(new_user)
                db.session.commit()
                flash('Account created successfully. Please log in.', 'success')
                app.logger.info(f'New user registered: {form.username.data}')
                return redirect(url_for('login'))
            except IntegrityError:
                db.session.rollback()
                flash('An error occurred. Please try again.', 'error')
                return redirect(url_for('register'))

        return render_template('register.html', form=form)
    except RateLimitExceeded:
        app.logger.warning(f'Rate limit exceeded for registration from IP: {request.remote_addr}')
        return render_template('rate_limit_exceeded.html'), 429
```

Figure 69

In this part of the code snippet, there are two routes involved: /logout and /register. These routes are for special functionalities within the application, reaching out to help provide the best user experience while ensuring the security of the app.

The /logout route logs the user out. In this route, every time a user accesses it, it removes the session token with session.pop('token', None), thereby logging out the user. A flash message will appear, informing the user of successful logging out. Finally, it redirects the user to the home page. This route is an important one in letting users log out safely and, therefore, in keeping their accounts safe from unwanted access.

This is the user registration view. It accepts both GET and POST methods so that users can view the registration form and send in their registration details. The @limiter.limit("3 per hour") decorator adds a rate limit: users are allowed to register only up to three times per hour from the same IP address. This is useful in avoiding automated spam registrations.

An instance of the class RegistrationForm is created within the route. If the form is submitted and is valid, it checks if a user with the given email address already exists. In case of a user's existence, it flashes an error message, redirecting the user back to the page of registration.

In case the email is not in use, the password of the user is hashed for security, and a new user object with the form data provided is created. The new user is added to the database session and committed on completion. If registration goes well, a success message is flashed, and the user is redirected to the login page. Also, a log entry will be made in order to record this new registration.

In case of database integrity errors, for instance, duplicate entries, the session is rolled back, a flash message is shown with an error, and the user is redirected to try again. In case of exceeding the rate limit, a warning will be logged, and the user will get a rate limit exceeded page.

These routes are key to ensuring safe management of user sessions and their registration. The /logout route lets users terminate their sessions safely, and the /register route cares for new registrations of users with appropriate validations, rate limiting, and error handling. This keeps the process of user management within the application very safe and integral..

```python
@app.route('/dashboard')
@token_required
def dashboard(current_user):
    return render_template('dashboard.html', user=current_user)


#########################################################################################################################################

@app.route('/deposit', methods=['GET', 'POST'])
@token_required
def deposit(current_user):
    form = DepositForm()
    if form.validate_on_submit():
        try:
            amount = form.amount.data
            amount = float(bleach.clean(str(amount), strip=True))
            current_user.balance += amount
            transaction = Transaction(user_id=current_user.id, transaction_type='Deposit',
                                      amount=amount, balance=current_user.balance)
            transaction.hash = generate_transaction_hash(transaction)
            db.session.add(transaction)
            db.session.commit()
            flash(f'Deposited {amount:.2f} successfully.', 'success')
            app.logger.info(f'User {current_user.username} deposited {amount:.2f}')
            return redirect(url_for('dashboard'))
        except Exception as e:
            db.session.rollback()
            app.logger.error(f'Error during deposit: {str(e)}')
            flash("An error occurred during the deposit. Please try again.', 'error')
    return render_template('deposit.html', form=form)

```

Figure 70

The program defines two routes: /dashboard and /deposit. Individual routes handle specific functionalities regarding user interactions and financial transactions within the app. Both routes are protected in that a user must be logged in to execute these actions. This is assured by the @token_required decorator.

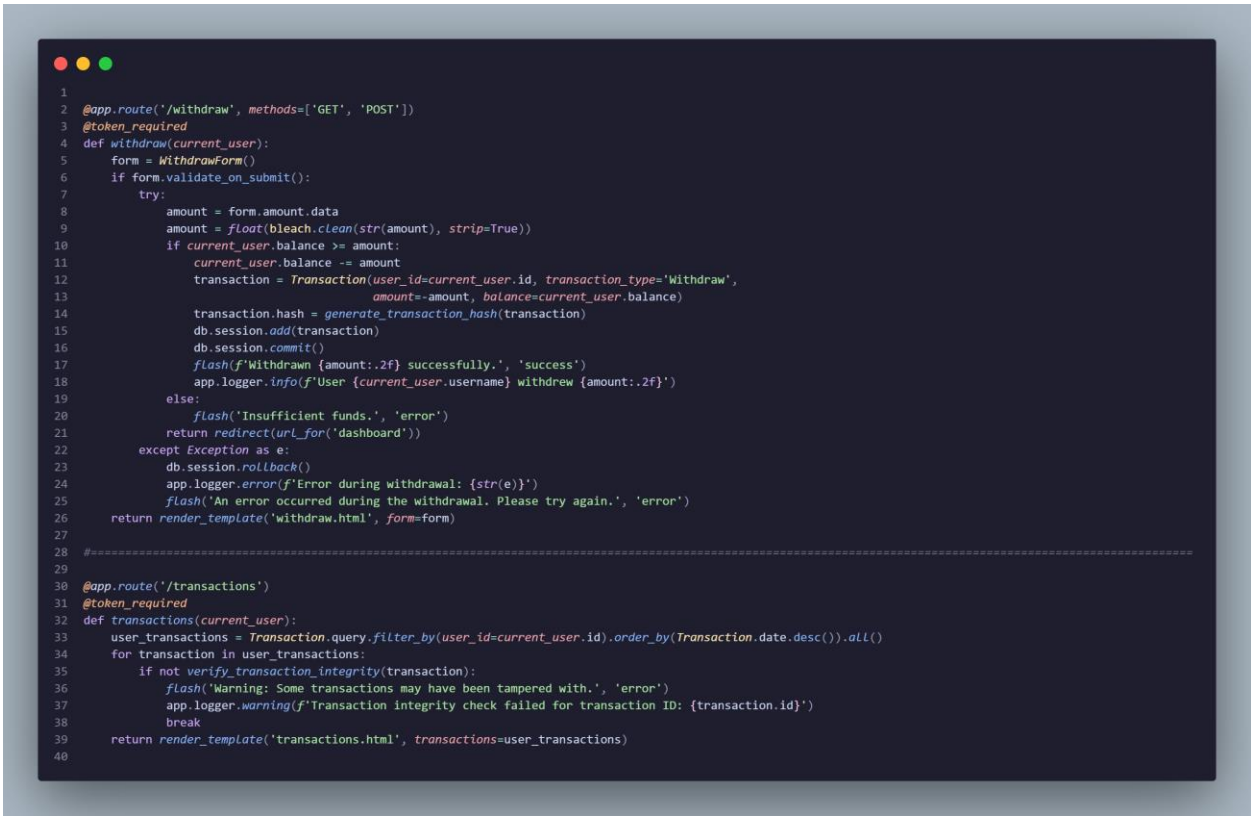
The /dashboard route is for the user's dashboard. The @token_required decorator makes sure that only authenticated users can access this route. It passes the current authenticated user to the dashboard function for processing. After processing, it renders the dashboard.html template, passing the current_user to display personalized information in the template. This route serves to provide users with access to their personal dashboard, where they can view details regarding their accounts and any other relevant information.

The /deposit route handles deposit transactions. It supports both GET and POST methods that allow users to view the deposit form and submit their amounts for depositing. The decorator @token_required at the very top ensures that only authenticated users have access to this route. The instance of DepositForm is created within the function and is validated upon submission.

In the case of form validation, the amount to be deposited is cleaned and sanitized with the bleach.clean method to avoid any kind of malicious input processing. The cleaned amount is added to current_user balance. A new object, Transaction, is instantiated to log this deposit. It logs the user ID, type of transaction, amount, and balance after update. A unique hash is generated against this transaction for integrity. A transaction is added to the database session and committed.

On successful deposit, flash a success message and redirect the user to the dashboard. The application logs the successful deposit for auditing purposes. In case of an error raised during the process, it rolls back the session and flashes an error message, logging the error. Finally, the function renders the deposit.html template, passing the form for the user to try again.

While most of those routes are focal to the functionality and usability of the application, /dashboard brings in a personalized dashboard for the user, and the /deposit route securely yet efficiently performs deposit transactions. Requiring authentication, sanitization of inputs, and error handling within these routes helps keep the application user-friendly and secure..



```python

@app.route('/withdraw', methods=['GET', 'POST'])
@token_required
def withdraw(current_user):
    form = WithdrawForm()
    if form.validate_on_submit():
        try:
            amount = form.amount.data
            amount = float(bleach.clean(str(amount), strip=True))
            if current_user.balance >= amount:
                current_user.balance -= amount
                transaction = Transaction(user_id=current_user.id, transaction_type='Withdraw',
                                          amount=-amount, balance=current_user.balance)
                transaction.hash = generate_transaction_hash(transaction)
                db.session.add(transaction)
                db.session.commit()
                flash(f'Withdrawn {amount:.2f} successfully.', 'success')
                app.logger.info(f'User {current_user.username} withdrew {amount:.2f}')
            else:
                flash('Insufficient funds.', 'error')
            return redirect(url_for('dashboard'))
        except Exception as e:
            db.session.rollback()
            app.logger.error(f'Error during withdrawal: {str(e)}')
            flash('An error occurred during the withdrawal. Please try again.', 'error')
    return render_template('withdraw.html', form=form)

#=============================================================================================================================================

@app.route('/transactions')
@token_required
def transactions(current_user):
    user_transactions = Transaction.query.filter_by(user_id=current_user.id).order_by(Transaction.date.desc()).all()
    for transaction in user_transactions:
        if not verify_transaction_integrity(transaction):
            flash('Warning: Some transactions may have been tampered with.', 'error')
            app.logger.warning(f'Transaction integrity check failed for transaction ID: {transaction.id}')
            break
    return render_template('transactions.html', transactions=user_transactions)

```

Figure 71

The above code snippet defines two routes: /withdraw and /transactions. Each route is assigned certain functionalities regarding financial transactions and the history of such transactions performed within the application. Both routes require user authentication, which was assured by the @token_required decorator.

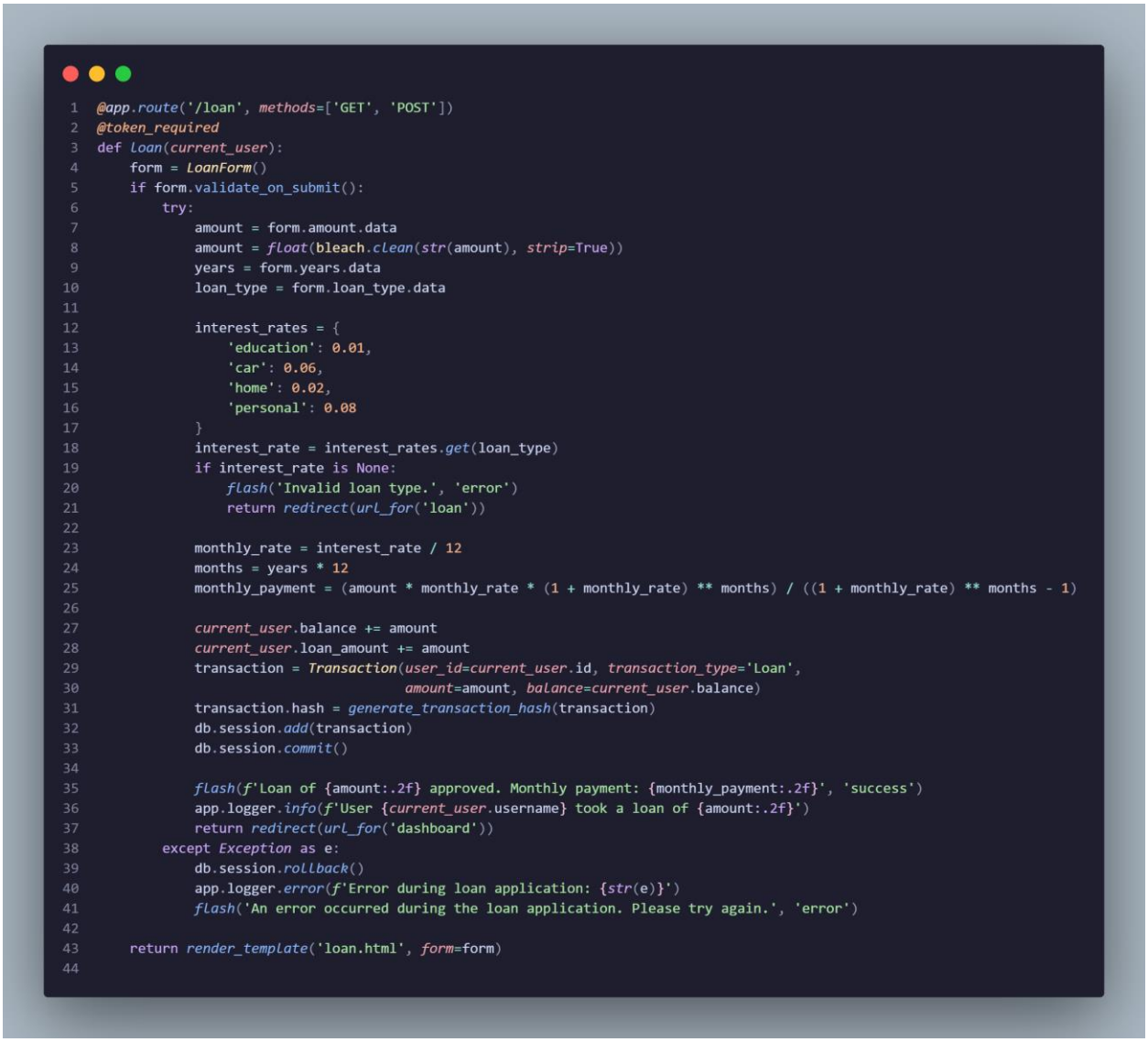
The /withdraw route is responsible for processing the withdrawal transactions. It supports both GET and POST methods so that a user can view a form for withdrawal and actually send in their requests. This decorator, @token_required, ensures that access to this route is by authenticated users only. The instance of class WithdrawForm is created in the function and then validated on form submission.

If the form is valid, clean and sanitize the withdrawal amount using the bleach.clean method to prevent any malicious input. Check whether the current_user has sufficient balance for the withdrawal amount. If there are enough funds, the amount is subtracted from balance and a new instance of class Transaction will be created to track this withdrawal with all details, such as user ID, transaction type, amount, and new balance. A unique hash is generated to guarantee the integrity of the transaction. Afterwards, it adds the transaction to the database session and commits it. On successful withdrawal, a success message is flashed, and the user is redirected to their dashboard. In case of insufficient balance, an error message is flashed. Any errors

in the process are logged, and an appropriate error message is flashed to the user. The function finally renders the withdraw.html template with the form for the user to try again.

The /transactions route returns the history of transactions for a user. The @token_required decorator ensures that this endpoint is accessible only by authenticated users. In the function, all transactions for the current_user are queried from the database in descending order of transaction date. Each and every integrity of the transaction is ensured through a call to the verify_transaction_integrity function. In case of failure of integrity checks on any transaction, a warning flashes to the user, and a warning message logs. The function finally renders the transactions.html template, passing it a list of transactions for display.

All these routes are critical for the processing of financial transactions and for the user to view his transaction history. The /withdraw route facilitates secure withdrawal by checking balances and sanitizing inputs; the /transactions route is in charge of the integrity of the transaction records, warning users in case of tampering. This design will help in keeping the financial activities of the application secure and reliable..



```python
@app.route('/loan', methods=['GET', 'POST'])
@token_required
def loan(current_user):
    form = LoanForm()
    if form.validate_on_submit():
        try:
            amount = form.amount.data
            amount = float(bleach.clean(str(amount), strip=True))
            years = form.years.data
            loan_type = form.loan_type.data

            interest_rates = {
                'education': 0.01,
                'car': 0.06,
                'home': 0.02,
                'personal': 0.08
            }
            interest_rate = interest_rates.get(loan_type)
            if interest_rate is None:
                flash('Invalid loan type.', 'error')
                return redirect(url_for('loan'))

            monthly_rate = interest_rate / 12
            months = years * 12
            monthly_payment = (amount * monthly_rate * (1 + monthly_rate) ** months) / ((1 + monthly_rate) ** months - 1)

            current_user.balance += amount
            current_user.loan_amount += amount
            transaction = Transaction(user_id=current_user.id, transaction_type='Loan',
                                      amount=amount, balance=current_user.balance)
            transaction.hash = generate_transaction_hash(transaction)
            db.session.add(transaction)
            db.session.commit()

            flash(f'Loan of {amount:.2f} approved. Monthly payment: {monthly_payment:.2f}', 'success')
            app.logger.info(f'User {current_user.username} took a loan of {amount:.2f}')
            return redirect(url_for('dashboard'))
        except Exception as e:
            db.session.rollback()
            app.logger.error(f'Error during loan application: {str(e)}')
            flash('An error occurred during the loan application. Please try again.', 'error')

    return render_template('loan.html', form=form)

```

Figure 72

This part of the code snippet tells about an application route to handle loan applications to the application. The `/loan` route addresses the process of loan application specifically dealing with how users make applications for loans in a secure and effective manner. The route is made such that it allows request methods both in GET and POST. This implicates that the user will view the loan application form and request their loan right on the request methods. The @token_required decorator is there to make sure that only authenticated users will be allowed to go along with accessioning the route.

A class LoanForm instance is built and validated within the loan function on form submission. In case the form has a valid passing, clean and sanitize the loan amount for malicious input using the bleach.clean method, also taking the loan's tenure in year input. It also retrieves the loan type from the form data.

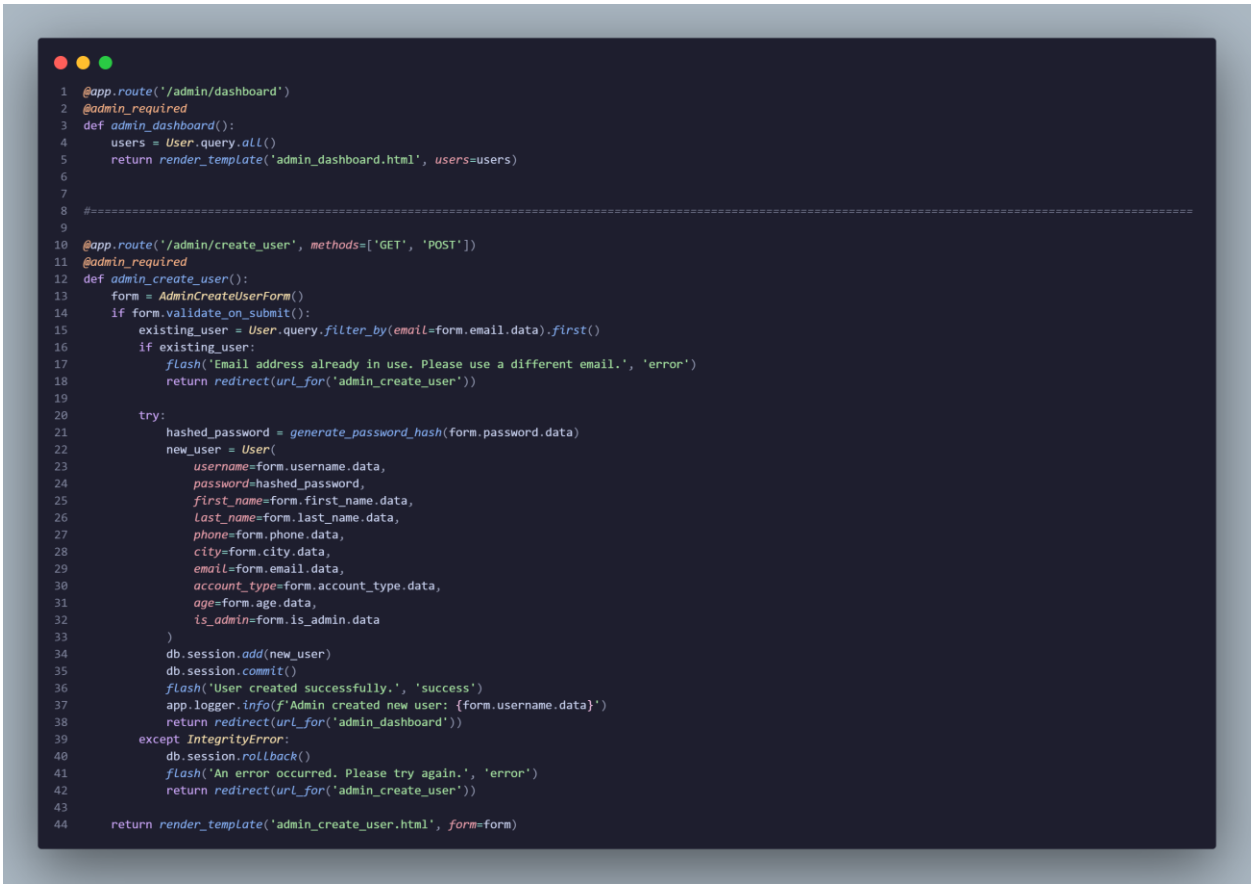
Displays the definition of interest rates for education, car, home and personal. Retrieves by interest rate that corresponds to the selected loan type. If an invalid loan type is selected, a flash message of error is displayed and will be redirected to the loan application page.

The monthly interest rate is the annual interest rate divided by 12. The total number of months for the loan is equal to the number of years times 12. The monthly payment is then calculated using the general formula for monthly payments on a fully amortizing loan.

The loan amount is cumulatively added to the user's balance and loan amount. To record the loan, a new Transaction object is created, which would hold other details like the user ID, type of transaction, amount and new balance. It also creates a hash for the transaction for finality. Then I waited to add the transaction to the session of the database and committed.

If the loan application has been successfully executed, flash a success message that the loan amount has been credited and also mention the monthly payment. Redirect the user to his/her dashboard. Log the successful loan application for auditing purposes. In case of any errors during the process, roll back the session, flash an error message, and log the error. Render the loan.html template, passing the form for the user to try again.

Generally, the /loan route is of much importance when it comes to securely taking care of loan applications effectively: it assures that the user can apply for a loan with proper validation, interest calculation, and error handling from the start to the finish of the loan application. This maintains the integrity of the loan application process and provides a good experience for the user..



```python
@app.route('/admin/dashboard')
@admin_required
def admin_dashboard():
    users = User.query.all()
    return render_template('admin_dashboard.html', users=users)


#=============================================================================================================================================

@app.route('/admin/create_user', methods=['GET', 'POST'])
@admin_required
def admin_create_user():
    form = AdminCreateUserForm()
    if form.validate_on_submit():
        existing_user = User.query.filter_by(email=form.email.data).first()
        if existing_user:
            flash('Email address already in use. Please use a different email.', 'error')
            return redirect(url_for('admin_create_user'))

        try:
            hashed_password = generate_password_hash(form.password.data)
            new_user = User(
                username=form.username.data,
                password=hashed_password,
                first_name=form.first_name.data,
                last_name=form.last_name.data,
                phone=form.phone.data,
                city=form.city.data,
                email=form.email.data,
                account_type=form.account_type.data,
                age=form.age.data,
                is_admin=form.is_admin.data
            )
            db.session.add(new_user)
            db.session.commit()
            flash('User created successfully.', 'success')
            app.logger.info(f'Admin created new user: {form.username.data}')
            return redirect(url_for('admin_dashboard'))
        except IntegrityError:
            db.session.rollback()
            flash('An error occurred. Please try again.', 'error')
            return redirect(url_for('admin_create_user'))

    return render_template('admin_create_user.html', form=form)
```

Figure 73

The code snippet creates two routes in this part: /admin/dashboard and /admin/create_user. Both routes are specifically designed to hold administrative functions and ensure, through the decorator @admin_required, that only users with administrative privileges can access said functionalities.

The /admin/dashboard route renders the admin dashboard. This retrieves all users from the database using User.query.all() and passes this list of users to the admin_dashboard.html template to process. Since this is an @admin_required decorator, only authenticated users who are admin can access this route. This view is important in giving administrators an overview of all accounts of users so that they can manage the different users.

The /admin/create_user route is used for creating new user accounts as an administrator. It supports both GET and POST methods for the admin to view the new user creation form and to send in data for a new user. In the function, an instance of AdminCreateUserForm is created and validated upon form submission.

If the form is valid, it checks whether a user with the email provided already exists. If the existing user is found, it flashes an error message; otherwise, admin redirects back to the creation form of a user. In the event of no usage of the email, for security purposes, the password is hashed for this new user. A new user's object is created, including fields such as username, password, first name, surname, phone, city, email, account type, age, and admin status.

A new user object is added to the database session and committed. In case of successful user creation, a success message is flashed and the administrator is redirected to the admin dashboard. The application will also log the creation of the new user for auditing purposes. If a database integrity error happens, such as a duplicate entry, the session is rolled back; an error message is flashed; and the administrator is redirected to try again.

These routes are, therefore, very important in user account management in the application. The /admin/dashboard route allows the admin to obtain an elaborate dashboard view of all the users for easy management of users. The /admin/create_user route enables admins to create new user accounts safely, with appropriate validation and handling of user data. These thus ensure the security and integrity of the user management process within the application..

```python
@app.route('/profile')
@token_required
def profile(current_user):
    return render_template('profile.html', user=current_user)


#########################################################################################################################################

@app.route('/change_password', methods=['GET', 'POST'])
@token_required
def change_password(current_user):
    if request.method == 'POST':
        old_password = request.form['old_password']
        new_password = request.form['new_password']
        confirm_password = request.form['confirm_password']

        if not check_password_hash(current_user.password, old_password):
            flash('Current password is incorrect.', 'error')
        elif new_password != confirm_password:
            flash('New passwords do not match.', 'error')
        elif len(new_password) < 8:
            flash('Password must be at least 8 characters long.', 'error')
        else:
            current_user.password = generate_password_hash(new_password)
            db.session.commit()
            flash('Password changed successfully.', 'success')
            app.logger.info(f'User {current_user.username} changed their password')
            return redirect(url_for('profile'))

    return render_template('change_password.html')
```

Figure 74

The upper part of this snippet of code defines two routes: /profile and /change_password. Both routes require user authentication, which has been ensured through the @token_required decorator.

The /profile route renders the user's profile page. It is decorated with the @token_required decorator. This route is accessible only to users who are authenticated. It passes the current_user down to the profile.html template, enabling the template to display information about the user. This route is important to give users access to their personal profile, where they can, probably, view and update their account details.

The /change_password route changes the password of the authenticated user. It supports both GET and POST methods so that it's able to return a form to the user to change their password, and also actually change the user's password. The function retrieves the old password, the new password, and the confirmation of the new password from the form data if this is a POST request.

First, it checks whether the old password provided by the user matches the existing password in the database with check_password_hash. If the old password is wrong, flash an error message. Next, it checks if the new password and confirm password match; otherwise, another error message is flashed. Additionally, it ensures the new password has a minimum length of 8 characters; otherwise, an error message is flashed.

Finally, in case of the passing of all the validations, a new hashed password is generated for the user using the generate_password_hash function and then updated in the database. Success is flashed, and the user is redirected to their profile view. Moreover, the application logs an event of changing a password for auditing purposes.

These routes are essential to the security and usability of the application. The /profile route grants access to personal information; the /change_password securely updates user passwords. Enforcing authentication and validation, these routes ensure that user data is handled in a safe and correct way..

```python
@app.route('/audit_log')
@admin_required
def audit_log():
    logs = []
    with open('logs/hamadabank.log', 'r') as log_file:
        logs = log_file.readlines()
    return render_template('audit_log.html', logs=logs)

#=============================================================================================================================================

@app.route('/api/balance')
@token_required
def get_balance(current_user):
    return jsonify({'balance': current_user.balance})


#=============================================================================================================================================


def generate_transaction_hash(transaction):
    """Generate a hash for the transaction to ensure integrity."""
    data = f"{transaction.user_id}{transaction.transaction_type}{transaction.amount}{transaction.balance}{transaction.date}"
    return generate_password_hash(data)


#=============================================================================================================================================


def verify_transaction_integrity(transaction):
    """Verify the integrity of a transaction by checking its hash."""
    data = f"{transaction.user_id}{transaction.transaction_type}{transaction.amount}{transaction.balance}{transaction.date}"
    return check_password_hash(transaction.hash, data)


#=============================================================================================================================================


@app.errorhandler(404)
def not_found_error(error):
    return render_template('404.html'), 404

#=============================================================================================================================================

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500

```

Figure 75

Now, the script defines a number of routes and functions to handle some specific functionality within the application, making sure it is also well secured and logged, with appropriate error handling.

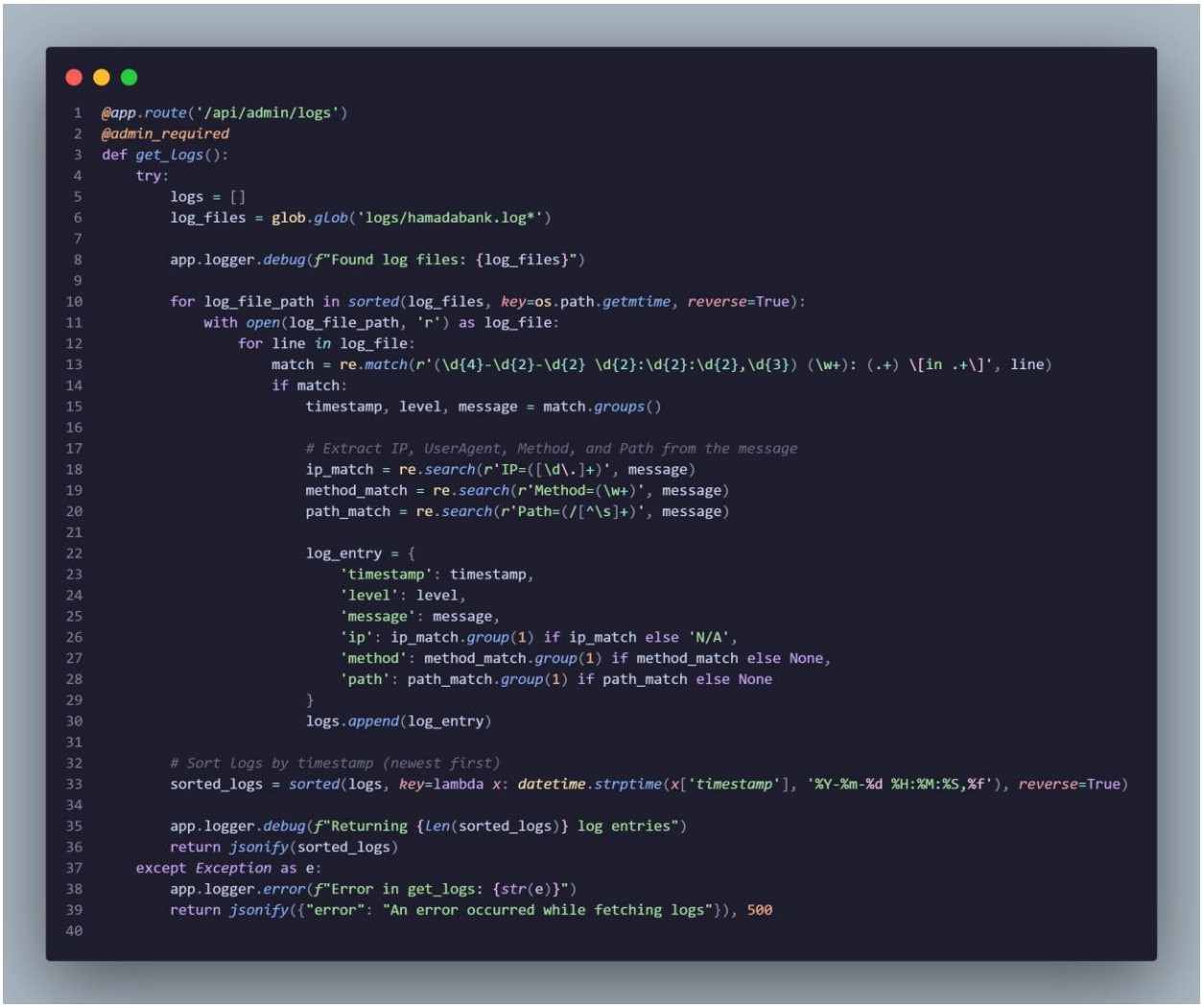
This view is created for the /audit_log route and is intended to display audit logs for the administrator. It is added with an @admin_required decorator, which ensures the route is accessible only to authenticated administrators. It reads a log file named hamadabank.log and passes its content to the audit_log.html template to be displayed. The route is very critical to ensuring transparency and accountability within the system by giving administrators the capability to review activity logs of the application.

The /api/balance route exposes an API to obtain the current user balance. It is also protected by the @token_required decorator, meaning only an authenticated user can call it. This function is particularly useful for front-end applications or other services aiming to show the user's balance by returning a response in JSON format.

It generates a hash for any given transaction, ensuring the integrity through multifaceted attributes of the transaction. These are combined into a string and hashed using generate_password_hash. That's an important function when making a unique and secure hash that will later help to verify the integrity of a transaction.

The function verify_transaction_integrity checks the integrity of a given transaction. This function simply joins back the transaction attributes into a string, then does the matching of the hash using check_password_hash. This is a very critical function in detecting any tampering or corruption in transaction records to ensure the integrity of financial data.

The Error handlers for the 404 and 500 HTTP status codes handle not found and internal server errors respectively. The @app.errorhandler(404) decorator allows the not_found_error function to render a custom 404.html template when a page has not been found. Similarly, the @app.errorhandler(500) decorator registers the internal_error function for rendering a custom 500.html template and rolling back the database session in case of an internal server error. These handlers enhance the user experience with meaningful error pages and make the application more stable by gracefully handling errors.



```python
@app.route('/api/admin/logs')
@admin_required
def get_logs():
    try:
        logs = []
        log_files = glob.glob('logs/hamadabank.log*')

        app.logger.debug(f"Found log files: {log_files}")

        for log_file_path in sorted(log_files, key=os.path.getmtime, reverse=True):
            with open(log_file_path, 'r') as log_file:
                for line in log_file:
                    match = re.match(r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2},\d{3}) (\w+): (.+) \[in .+\]', line)
                    if match:
                        timestamp, level, message = match.groups()

                        # Extract IP, UserAgent, Method, and Path from the message
                        ip_match = re.search(r'IP=([\d\.]+)', message)
                        method_match = re.search(r'Method=(\w+)', message)
                        path_match = re.search(r'Path=(/[^\s]+)', message)

                        log_entry = {
                            'timestamp': timestamp,
                            'level': level,
                            'message': message,
                            'ip': ip_match.group(1) if ip_match else 'N/A',
                            'method': method_match.group(1) if method_match else None,
                            'path': path_match.group(1) if path_match else None
                        }
                        logs.append(log_entry)

        # Sort logs by timestamp (newest first)
        sorted_logs = sorted(logs, key=lambda x: datetime.strptime(x['timestamp'], '%Y-%m-%d %H:%M:%S,%f'), reverse=True)

        app.logger.debug(f"Returning {len(sorted_logs)} log entries")
        return jsonify(sorted_logs)
    except Exception as e:
        app.logger.error(f"Error in get_logs: {str(e)}")
        return jsonify({"error": "An error occurred while fetching logs"}), 500

```

Figure 76

The next part of the code snippet simply defines an API route that administrators can call to retrieve the log files of their application. At the route /api/admin/logs, it handles getting and processing log data and makes sure that, thanks to the @admin_required decorator, only authenticated administrators may view this information.

The get_logs function begins by initializing an empty list to hold log entries. It then uses glob.glob to find all log files that fit the pattern logs/hamadabank.log*, hence including the main log file and any rotated log files. Found log files are sorted in order of modification time in descending order, so the newest logs will be processed first. This will become useful to give priority to the newest log entries.

This function opens every log file and reads it line by line. It uses a regular expression to do an initial match and extraction of parts from every log entry relevant to this program, including timestamp, log level, and

log message. Further regular expression searches extract additional information, as available, such as an IP address, an HTTP method, and a request path from the message.

Every log record is maintained as a dictionary. This dictionary includes timestamp, log level, message, IP address, HTTP method, and request path. Such entries are added to the logs list. Finally, all log entries will be sorted by timestamp in descending order after all log files have been read through; thus, the newest entries appear first.

Finally, it returns a JSON response with the log entries that are sorted, detailing every small piece of information from the app logs for administrators. This API route will turn into an important route for monitoring and auditing, because admins will have the ability to check detailed information from the logs and find out possible problems or security incidents.

In case of any errors during log retrieval or processing, catch the exception, log an error message, and return a JSON response indicating a 500 status code with an error message, which means there is an internal server error.

The route opens a secure, high-performance path through which administrators are able to access and review the log data, thus improving the logging and monitoring capabilities of the application. All this works to help in maintaining transparency, accountability, and security within the system.

```
if __name__ == '__main__':
    with app.app_context():
        if not os.path.exists('Hamadabank.db'):
            print("Database does not exist. Creating new database...")
            db.create_all()
            recreate_database()
        else:
            try:
                User.query.first()
            except Exception as e:
                if 'no such column: user.is_admin' in str(e):
                    print("Updating database schema...")
                    recreate_database()
                else:
                    raise e


    app.run(debug=False, port=5000)  # Run without SSL

```

Figure 77

This section of code is expected to instantiate a Flask application, run it, and make sure that the database is well set before this application begins.

The block if __name__ == '__main__': ensures that this code is run only in the case this script is run directly and not if it is imported as a module in another script. This is a common idiom in Python that allows the aim of making the code importable as a module while remaining executable as a script.

In this block, app.app_context() is setup. It is used to allow access to the context of the Flask application itself when needed; this can be the case for database queries and changes.

First, the program checks for a database file named Hamadabank.db using os.path.exists('Hamadabank.db'). If there is no database, it will output a message that a new database will be created. Then, it will call db.create_all() to generate the database with tables according to the models defined in your application. After creating the database, it calls the recreate_database() function to set up initial data—for instance, creating an admin user.

Assuming the database file is already in place, it tries to query the User model using User.query.first() to ensure that its database schema is current. Logging ensures that if an error occurs while searching, it logs whether 'no such county: user.is_admin' is in the error message. This error message will be received if the schema of the database was updated and in the new scheme, there was no such column that existed for is_admin. If so, print out an informative message that the database schema will be updated, and then actually do that by calling recreate_database() to update the schema and create initial data. If it is for any other reason, it reraises the exception letting whatever called it handle this problem.

Finally, app.run(debug=False, port=5000) initiates the Flask development server at port 5000 without SSL but with debug mode off. Running the application in this way makes it possible to accept any incoming HTTP requests and serve back responses; thus, it becomes usable to the user.

This initializing and startup script is, therefore, very critical in ensuring that the application database is well set and ready for action before the server starts handling requests. It deals with database creation, updating schema, and initializing the Flask application, which is the solid base for running an application.