Introduction

Student government elections play a pivotal role in fostering leadership and representing student interests within the university community. Traditionally, these elections at our university have been conducted using generic tools like Google Forms. While Google Forms offers a convenient and accessible platform for data collection, it is not specifically designed to handle the unique requirements of election processes. This reliance on a general-purpose tool highlights the necessity for a more specialized solution that can address the intricacies of conducting secure and efficient student council elections.

Related Work

Several existing election systems, such as Election Buddy provide functionalities that cater to various voting needs. Election Buddy, for instance, offers features like voter authentication, ballot customization, and result tabulation. However, its costs exceed the budget of our university, making it an impractical choice despite its robust capabilities. On the other hand, Google Forms is widely utilized due to its accessibility and ease of use but falls short in providing the necessary security and specialized election features.

Problem Statement

The current utilization of Google Forms for student elections introduces several critical vulnerabilities that compromise the integrity of the electoral process. Firstly, Google Forms lacks robust mechanisms to ensure voter anonymity, making it susceptible to potential breaches where voter identities could be traced back to their votes. This undermines the confidentiality essential for fair elections. Secondly, the platform does not effectively prevent multiple submissions from the same user, raising concerns about the potential for vote manipulation and multiple voting instances. Additionally, managing and verifying election results through Google Forms is cumbersome, as it lacks real-time result aggregation and comprehensive verification tools, making the process inefficient and prone to errors. These challenges not only jeopardize the security and fairness of the elections but also diminish trust among the student body in the electoral process.

Solution Proposal

To address the shortcomings of the current election method, we propose the development of a dedicated web application tailored to conduct student council elections securely and efficiently. The proposed system will integrate seamlessly with the university's existing ID infrastructure, leveraging unique student identifiers to authenticate voters reliably. Key features of the application include:

1. **Secure Authentication:** By interfacing with the university's ID system, the application ensures that only eligible students can access the voting platform, mitigating unauthorized access.
2. **Anonymity Assurance:** Utilizing encryption techniques, such as those implemented in our provided code (e.g., Fernet encryption for voter IDs), the system guarantees that individual votes cannot be traced back to voters, preserving anonymity.
3. **Prevention of Double Voting:** The application employs mechanisms to detect and prevent multiple submissions from the same user, ensuring that each student can vote only once per election cycle.
4. **Real-Time Result Management:** The system provides real-time aggregation and transparent display of election results, enhancing the efficiency and reliability of the outcome verification process.
5. **User-Friendly Interface:** Drawing inspiration from the user experience elements in our code, such as intuitive templates and clear navigation paths, the application will offer an accessible and straightforward voting experience for all students.
6. **Scalability and Maintainability:** Built using robust frameworks like FastAPI and SQLAlchemy, the application is designed to be scalable and maintainable, accommodating future enhancements and increasing user loads without compromising performance.

By addressing the identified vulnerabilities and incorporating these features, the proposed system aims to deliver a secure, transparent, and efficient electoral process that fosters trust and participation among the student body.

Validation

The success of the proposed election system will be rigorously evaluated through multiple validation criteria:

Security Assessment: Conducting thorough security audits and penetration testing to ensure that voter data is protected against unauthorized access and that anonymity is maintained throughout the voting process.

Functionality Testing: Performing extensive testing of all system features, including secure authentication, vote submission, and result aggregation, to verify that they operate as intended without vulnerabilities or glitches.

User Feedback: Engaging a diverse group of students in beta testing phases to gather feedback on the user experience, identifying areas for improvement, and ensuring that the application meets the needs and expectations of its user base.

Performance Metrics: Monitoring system performance during peak voting periods to assess scalability and responsiveness, ensuring that real-time results are accurately and promptly displayed without delays or failures.

Compliance Verification: Ensuring adherence to all relevant data protection regulations and university policies, maintaining the legal and ethical integrity of the electoral process.

Adoption Rate: Measuring the adoption and acceptance of the system in actual elections, aiming for widespread use in future student council elections as evidence of its effectiveness and reliability.

Successful validation will not only signify the system's readiness for deployment but also its potential to serve as a model for other institutions seeking to enhance the security and efficiency of their electoral processes.

Mechanisms:

1. Frameworks and Dependencies

- **FastAPI:** Utilized as the web framework, FastAPI offers high performance and modern features such as asynchronous request handling, which is essential for managing concurrent voting activities.
- **SQLAlchemy:** Employed for ORM (Object-Relational Mapping), SQLAlchemy facilitates seamless interaction with the database, allowing for efficient data manipulation and retrieval.
- **Jinja2:** Used for templating, Jinja2 enables the creation of dynamic and responsive HTML pages, ensuring a user-friendly interface.
- **Cryptography (Fernet):** Implements symmetric encryption to secure voter IDs, ensuring that vote anonymity is maintained.
- **itsdangerous:** Provides secure token generation and validation, crucial for email verification processes.
- **aiosmtplib:** Facilitates asynchronous email sending, enhancing the efficiency of email-based verification and confirmation processes.

2. Secure Authentication and Verification

- **Email Verification:**
    - **Token Generation:** The `URLSafeTimedSerializer` from `itsdangerous` is used to generate time-bound tokens for email verification. This ensures that verification links expire after a specified duration (`max_age=3600` seconds), mitigating the risk of unauthorized access through stale tokens.
    - **Verification Process:** Upon user login, a verification link containing the token is sent to the user's email. The `/verify-email` route handles token validation, ensuring that only users with valid tokens can proceed to vote. This mechanism prevents unauthorized access and ensures that only legitimate voters can participate.
- **Integration with University ID System:**

- **Email Normalization and Validation:** During the login process (`/login` route), the entered email is normalized (lowercased and stripped of whitespace) and validated to contain specific university-related components (`sias` and `krea`). This ensures that only emails affiliated with the university can access the voting system.
- **Voter Status Tracking:** The `Voter` model includes fields such as `has_voted` and `voted_at` to track voting status and enforce restrictions on multiple submissions. This integration with the database ensures robust authentication tied to the university's ID system.

3. Anonymity and Data Protection

- **Encryption of Voter IDs:**
  - **Fernet Encryption:** The `FERNET_KEY` environment variable is used to initialize a `Fernet` object, which encrypts voter IDs stored in the database (`voter.encrypted_id`). This encryption ensures that votes cannot be traced back to individual voters, preserving anonymity.
  - **Hashing Voting Tokens:** Voting tokens generated using `secrets.token_urlsafe(32)` are hashed using SHA-256 (`hashlib.sha256(voting_token.encode()).hexdigest()`) before being stored in the `Vote` model. This hashing ensures that even if the database is compromised, the tokens cannot be reverse-engineered to reveal voter identities.
- **Secure Storage of Sensitive Information:**
  - **Environment Variables:** Sensitive data such as encryption keys (`FERNET_KEY`), secret keys (`SECRET_KEY`), and email passwords (`EMAIL_PASSWORD`) are stored securely in environment variables, preventing exposure in the codebase.
  - **Session Management:** The application employs `SessionMiddleware` with secure configurations, including a secret key and appropriate session expiration (`max_age=1800` seconds), to manage user sessions securely.

4. Prevention of Double Voting

- **Voting Status Enforcement:**
  - **`has_voted` Flag:** The `Voter` model includes a `has_voted` boolean flag that is set to `True` upon successful vote submission. This flag is checked during the login and verification processes to prevent users from voting multiple times within the same election cycle.
  - **Time-Based Restrictions:** The `voted_at` timestamp records the last voting time. If a user attempts to vote again within six months (`datetime.timedelta(days=180)`), the system resets their voting status, allowing them to participate in new election cycles while preventing immediate re-voting.

- **Token-Based Vote Submission:**
    - **Voting Token Validation:** Each voting session is associated with a unique `voting_token` stored in the user's session. This token is hashed and compared against existing votes to ensure that each voter can submit only one vote per election, effectively preventing duplicate submissions.

5. Real-Time Result Management

- **Vote Tallying Mechanism:**
    - **`recalculate_vote_tally` Function:** This function resets all candidates' `vote_tally` to zero and iterates through all votes to aggregate first preferences. By updating vote tallies in real-time, the system ensures that election results are continuously up-to-date and accurately reflect current voting trends.
    - **Instant Runoff Voting (Commented Out):** Although currently commented out, the inclusion of an `instant_runoff_voting` function indicates an intent to implement advanced vote counting mechanisms. This would allow for more sophisticated result calculations, such as eliminating candidates with the fewest votes and redistributing their votes based on preferences, thereby enhancing result accuracy and fairness.

6. User Experience and Accessibility

- **Dynamic Templating:**
    - **Jinja2 Templates:** The use of Jinja2 for rendering HTML templates (`Login_EMV.html`, `Verification_Email.html`, `Voting_EMV.html`, etc.) ensures that the user interface is both dynamic and responsive. This approach allows for the seamless integration of data-driven content, providing users with real-time feedback and interactive elements.
    - **Clear Navigation Paths:** Defined routes for login (`/`), verification (`/verify-email`), rules (`/rules`), voting (`/voting`), summary (`/summary`), and thank you (`/thankyou`) guide users through the voting process in a logical and intuitive manner, minimizing confusion and enhancing usability.
- **Session Management:**
    - **State Preservation:** The application uses session variables (`user_email`, `voting_token`, `rules_read`) to maintain user state across different stages of the voting process. This ensures a smooth and coherent user experience, allowing users to progress through the voting stages without encountering unexpected interruptions or errors.

7. Error Handling and Logging

- **Comprehensive Error Handling:**

    - **Database Operations:** Each database interaction is wrapped in try-except blocks to handle potential exceptions gracefully. For instance, during vote submission and voter status updates, the system rolls back transactions in case of errors, preventing data inconsistencies and ensuring system stability.
    - **Email Operations:** The application handles exceptions during email sending (both verification and confirmation) by logging errors and providing user-friendly error messages, ensuring that users are informed of issues without exposing sensitive system details.
- **Detailed Logging:**
    - **Logging Configuration:** The application initializes logging with `logging.basicConfig(level=logging.INFO)` and creates a logger instance (`logger = logging.getLogger(__name__)`). This setup facilitates comprehensive logging of informational messages, warnings, and errors throughout the application's lifecycle.
    - **Operational Insights:** Log statements are strategically placed to capture key events, such as email verifications, vote submissions, status resets, and error occurrences. This detailed logging aids in monitoring system performance, diagnosing issues, and maintaining overall system health.

8. Security Best Practices

- **Environment Variable Management:** Sensitive credentials and keys are loaded using `dotenv` (`load_dotenv()`), ensuring that they are not hard-coded into the application. This practice enhances security by preventing accidental exposure of secrets in version control systems.
- **Token Security:**
    - **URLSafeTimedSerializer:** Utilizes a consistent secret key (`SECRET_KEY`) for token serialization, ensuring that tokens are securely generated and validated.
    - **Token Expiry:** Verification tokens are time-bound (`max_age=3600` seconds), reducing the window of opportunity for token misuse and enhancing overall security.
- **Data Encryption:**
    - **Fernet Encryption:** Ensures that sensitive data, such as voter IDs, are encrypted before storage, protecting against data breaches and unauthorized data access.
    - **Hashing Voting Tokens:** By hashing voting tokens with SHA-256 before storage, the system ensures that even if the database is compromised, the original tokens cannot be retrieved, safeguarding voter anonymity.

9. Scalability and Maintainability

- **Modular Code Structure:**

    - **Separation of Concerns:** The code separates different functionalities into distinct routes and helper functions (e.g., `get_db`, `recalculate_vote_tally`), promoting modularity and ease of maintenance.
    - **Reusable Components:** Common functionalities, such as database session management and encryption initialization, are encapsulated in reusable components, reducing code duplication and enhancing maintainability.
- **Framework Utilization:**
    - **FastAPI's Asynchronous Capabilities:** By leveraging FastAPI's asynchronous features (`async def`), the application can handle multiple concurrent requests efficiently, ensuring responsiveness even under high load.
    - **SQLAlchemy ORM:** Facilitates scalable database interactions, allowing for easy migration to more robust database systems (e.g., PostgreSQL) if needed in the future.