Decentralized application for student voting on Midnight Blockchain

Dimitrios Vassiliou

February 3, 2026

Introduction

Significant challenges in traditional voting systems, whether paper-based or electronic.

- Possibility of human error or manipulation
- Existence of a central server, possibility of disclosure of individual votes
- No guarantee of the accuracy of the results

Introduction

Goal? To create a decentralized voting platform that guarantees:

- Freedom to create votings
- Avoidance of double voting
- The inability to link a vote to a voter's identity
- The guarantee of the process and results
- Only eligible users can participate

We propose such a solution using Midnight blockchain technology.

Introduction

Students will be able to vote about university topics such as

- Student council elections
- Food and Facilities
- Educational trips

- It is a distributed database where information is spread across multiple nodes
- Each node in the network keeps a copy of all data
- It is impossible for anyone to alter the data, as they would need to control the majority of the network nodes

- **Hash Functions**: Mathematical string functions - it is computationally impossible to calculate the input from the output
- **Zero Knowledge Proofs**: A method by which the prover can convince the verifier that a particular statement is true, without revealing any additional information (witness) beyond the truth of the statement
- **Merkle Trees**: Data structure that allows someone to prove that a specific value belongs to a set without revealing that value. Recursively, each leaf node contains the hash of an element and each parent node contains the hash of the concatenation of its child nodes

- Blockchain technology that focuses on protecting users' personal data through the use of zero knowledge proofs
- Each user has a wallet (**Lace Wallet**) to interact with decentralized applications and perform transactions
- **Proof Server**: A container that runs locally on each user's device and generates zero knowledge proofs for each transaction performed by each user. These proofs are sent to the network and, if verified, the transaction is successful

- Contains the rules governing the use of the application
- Two states exist: The **public state**, which consists of the current value of all public variables, and the **private state**, which consists of each user's secret information and is always stored on their local system
- The **Compact** programming language clearly separates the two states: **ledger** variables are public and **witness** variables are an interface for implementing the private state, which is left to each dapp user

- Easy implementation of complex structures such as ZK Proofs and Merkle Trees
- No need to be an expert in cryptography
- Clear separation of which data is disclosed and which data remains private

- **Students as voters**: All those enrolled at the university have the right to vote in all available votings.
- **Students as organizers**: All those enrolled at the university have the right to create votings themselves
- **The university**: A trusted central authority and the only thing it has the authority to do as a stakeholder is to create the smart contract, ensuring that only all registered students of the university are eligible voters.

Methodology and Implementation

System Components

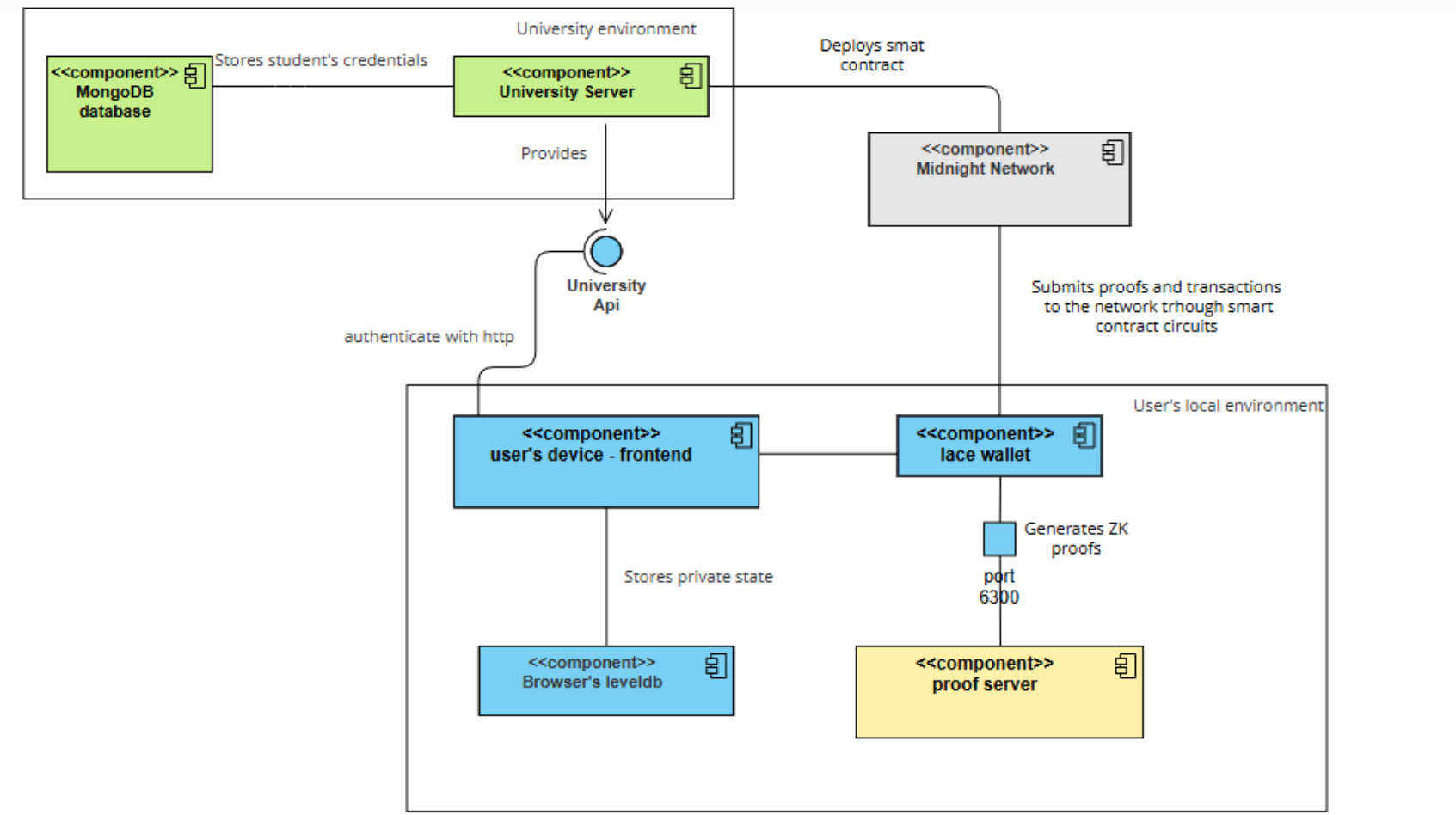

Figure: The component diagram of the system

Authentication process - Creation of a smart contract by the university

- Students are registered in the university database
- They calculate a secret key locally and send its hash (public key - sha256) along with their username and password to the university
- If the student exists, their public key is registered in the database
- The university creates the smart contract - Merkle Tree with the students' public keys

Methodology and Implementation

Application flow

- University deploys smart contract
- Students create votings
  - Two voting states: CLOSED (can be edited) and OPEN (ready to receive votes)
- Two voting periods
  - Cast vote period: Students cast their hashed vote
  - Publish vote period: Students publish their vote
- In each step students must prove they are eligible to participate

- It functions as a database for the system's data

```
export ledger votings: Set<Bytes<32>>;
export ledger voting_options: Map<Bytes<32>, Set<Bytes<32>>>;
export ledger voting_questions: Map<Bytes<32>, Opaque<"string">>;
export ledger voting_results: Map<Bytes<32>, Map<Bytes<32>, Counter
   >>;
export ledger eligible_voters: HistoricMerkleTree<5, Bytes<32>>;
export ledger voting_states: Map<Bytes<32>, VOTE_STATE>;
export ledger voting_nulifiers: Map<Bytes<32>, Set<Bytes<32>>>;
export ledger publish_voting_nulifiers: Map<Bytes<32>, Set<Bytes
   <32>>>;
export ledger voting_organizers: Map<Bytes<32>, Bytes<32>>;
export ledger hashed_votes: Map<Bytes<32>, Set<Bytes<32>>>;
export ledger publish_vote_expiration_time: Map<Bytes<32>, Uint
   <64>>;
export ledger cast_vote_expiration_time: Map<Bytes<32>, Uint<64>>;
```

Figure: The ledger variables

Each user's private state consists of 3 witness variables

- **local_secret_key**: Each user's secret key
- **find_voter_public_key**: The knowledge possessed by the valid voter that they know a valid path within the Merkle tree
- **secret_vote**: Map with the voting id as the key and the user's vote for it as the value

How does a voter prove that they are valid? Use of the **prove_eligibility** circuit

- It is checked whether the path held by the user (find_voter_public_key) can validly reconstruct the root.
- Why Merkle Trees and not Set? Because Merkle Trees do not reveal the user's public key - More security

```
circuit prove_eligibility(): Boolean {
    const participant_public_key = public_key(local_secret_key());
    const path = find_voter_public_key(participant_public_key);

    return eligible_voters.checkRoot(disclose(merkleTreePathRoot<5,
        Bytes<32>>(path)));
}
```

Figure: The prove_eligibility circuit

The actions performed by each voter are carried out through circuits and constitute transactions in Midnight. For each transaction, the validity of the user is checked - prove_eligibility

- Creating a vote - **create_voting**: Random vote ID - organizer is the public key of the current user - setting a deadline for casting and publishing votes.
- Add question, options - **edit_question**, **add_option**: Check if the current user is the organizer - ledger variable **voting_organizers**

Methodology and Implementation

The Smart Contract - Voting Rules/Actions

```
export circuit create_voting(publish_vote_expiration_time_v: Uint
   <64>, cast_vote_expiration_time_v: Uint<64>): [] {
    const organizer_public_key = disclose(organizer(local_secret_key
       ()));

    assert (publish_vote_expiration_time_v >
       cast_vote_expiration_time_v,
        "Publish vote deadline must be after cast vote deadline");

    assert (prove_eligibility(),
        "Not authorized!");

    const voting_id = generate_voting_id(count, local_secret_key());

    voting_organizers.insert(disclose(voting_id),
       organizer_public_key);
    votings.insert(disclose(voting_id));
    voting_states.insert(disclose(voting_id), VOTE_STATE.closed);
    voting_options.insert(disclose(voting_id), default<Set<Bytes
       <32>>>);
    voting_results.insert(disclose(voting_id), default<Map<Bytes
       <32>, Counter>>);
    publish_vote_expiration_time.insert(disclose(voting_id),
       disclose(publish_vote_expiration_time_v));
    cast_vote_expiration_time.insert(disclose(voting_id), disclose(
       cast_vote_expiration_time_v));
    count.increment(1);
}
```

Figure: The create_voting circuit

Methodology and Implementation

The Smart Contract - Voting Rules/Actions

```
export circuit edit_question(voting_id: Bytes<32>, voting_question:
    Opaque<"string">): [] {
     const organizer_public_key = disclose(organizer(local_secret_key
        ()));

     assert (prove_eligibility(),
         "Not authorized");

     assert (voting_organizers.lookup(disclose(voting_id)) ==
        organizer_public_key,
         "Not authorized");

     const current_voting_state = voting_states.lookup(disclose(
        voting_id));

     assert (current_voting_state == VOTE_STATE.closed,
         "Cannot edit the question since the voting is open");

     voting_questions.insert(disclose(voting_id), disclose(
        voting_question));
}
```

Figure: The edit_question circuit

- Voting - **cast_vote**:
  1. Deadline compliance
  2. How is double voting avoided? Use of a set of **voting_nulifiers** where hashes parameterized with voting id and secret key are registered
  3. The hash of the local vote parameterized with the secret key and the vote id is recorded in the ledger - **hashed_votes**

- Vote publication - **publish_vote**:
  1. Deadline compliance
  2. Nulifiers to avoid double posting, as in cast_vote
  3. Calculation of the vote hash based on the current private state - Check if this hash belongs to hashed_votes. If it does not belong, the user changed their vote - the transaction is rejected
  4. Check if the current vote belongs to the available options
  5. If all of the above are met, the vote is recorded in the results

  Although the vote is revealed, it is not possible to link it to the voter's public key due to the existence of Merkle Tree

Methodology and Implementation

The Smart Contract - Voting Rules/Actions

```
export circuit cast_vote(voting_id: Bytes<32>): [] {
    const current_voting_state = voting_states.lookup(disclose(
        voting_id));
    const current_expiration_time = cast_vote_expiration_time.looku
        (disclose(voting_id));
    assert (blockTimeLte(disclose(current_expiration_time)),
        "The deadline for vote casting has expired");
    assert (current_voting_state == VOTE_STATE.open,
        "Voting is not open");
    const voter_public_key = disclose(public_key(local_secret_key()
        );
    assert (prove_eligibility(),
        "Not authorized!");
    const voting_nullifier = disclose(pullifier(local_secret_key(),
        voting_id));
    assert (!voting_nulifiers.lookup(disclose(voting_id)).member(
        voting_nullifier),
        "Already voted for this voting");
    voting_nulifiers.lookup(disclose(voting_id)).insert(disclose(
        voting_nullifier));
    const hashed_vote = disclose(hash_secret_vote(secret_vote(
        voting_id), voting_id, local_secret_key()));
    hashed_votes.lookup(disclose(voting_id)).insert(disclose(
        hashed_vote));
}
```

Figure: The cast_vote circuit

Methodology and Implementation

The Smart Contract - Voting Rules/Actions

```
export circuit publish_vote(voting_id: Bytes<32>): [] {
    const current_voting_state = voting_states.lookup(disclose(
        voting_id));
    const current_publish_expiration_time =
        publish_vote_expiration_time.lookup(disclose(voting_id));
    assert (blockTimeLte(disclose(current_publish_expiration_time)),
        "The deadline for vote publishing has expired");
    const current_cast_expiration_time = cast_vote_expiration_time.
        lookup(disclose(voting_id));
    assert (blockTimeGt(disclose(current_cast_expiration_time)),
        "The deadline for vote casting has not expired yet");
    assert (current_voting_state == VOTE_STATE.open,
        "Voting is not open");
    assert (prove_eligibility(),
        "Not authorized!");
    const publish_voting_nullifier = disclose(publish_nullifier(
        local_secret_key(), voting_id));
    assert (!publish_voting_nulifiers.lookup(disclose(voting_id)).
        member(publish_voting_nullifier),
        "Already published the vote for this voting");
    const hashed_vote = disclose(hash_secret_vote(secret_vote(
        voting_id), voting_id, local_secret_key()));
    assert (hashed_votes.lookup(disclose(voting_id)).member(disclose
        (hashed_vote)),
        "Vote not correct!");
    assert (voting_options.lookup(disclose(voting_id)).member(
        disclose(secret_vote(voting_id))),
        "Not a valid option!");
    voting_results.lookup(disclose(voting_id)).lookup(disclose(
        secret_vote(voting_id))).increment(1);
    publish_voting_nulifiers.lookup(disclose(voting_id)).insert(
        publish_voting_nullifier);
}
```

- The compiler generates **Typescript** code and provides functions for submitting transactions.
- Implementation of an API that acts as a communication interface: Calling circuits and accessing the public state.

```
export interface DeployedVoteGuardianAPI {
  readonly deployedContractAddress: ContractAddress;
  readonly state$: Observable<VoteGuardianDerivedState>;

  cast_vote: (voting_id: Uint8Array) => Promise<void>;
  close_voting: (voting_id: Uint8Array) => Promise<void>;
  open_voting: (voting_id: Uint8Array) => Promise<void>;
  edit_question: (voting_id: Uint8Array, vote_question: string) => Promise<
     void>;
  create_voting: (publish_vote_expiration_time: bigint,
     cast_vote_expiration_time: bigint) => Promise<void>;
  add_option: (voting_id: Uint8Array, vote_option: Uint8Array) => Promise<
     void>;
  publish_vote: (voting_id: Uint8Array) => Promise<void>;
}
```

Figure: The API

Use Cases

Landing Page

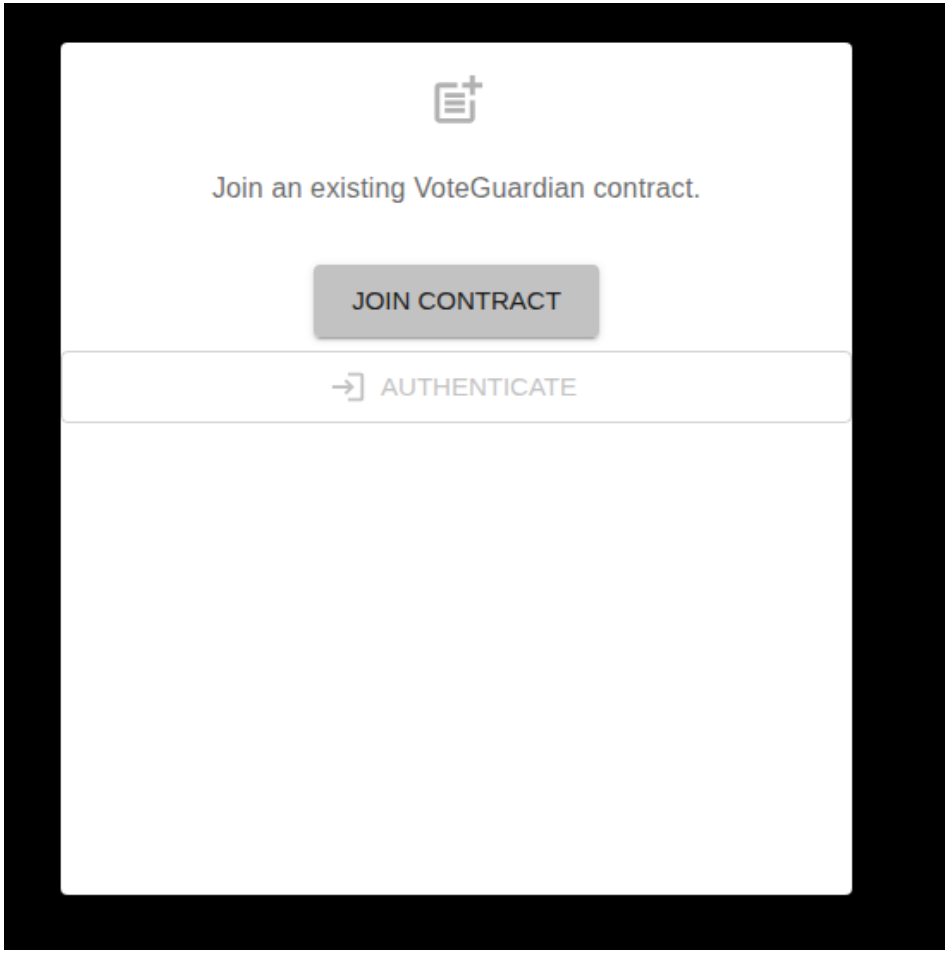

Figure: The home page

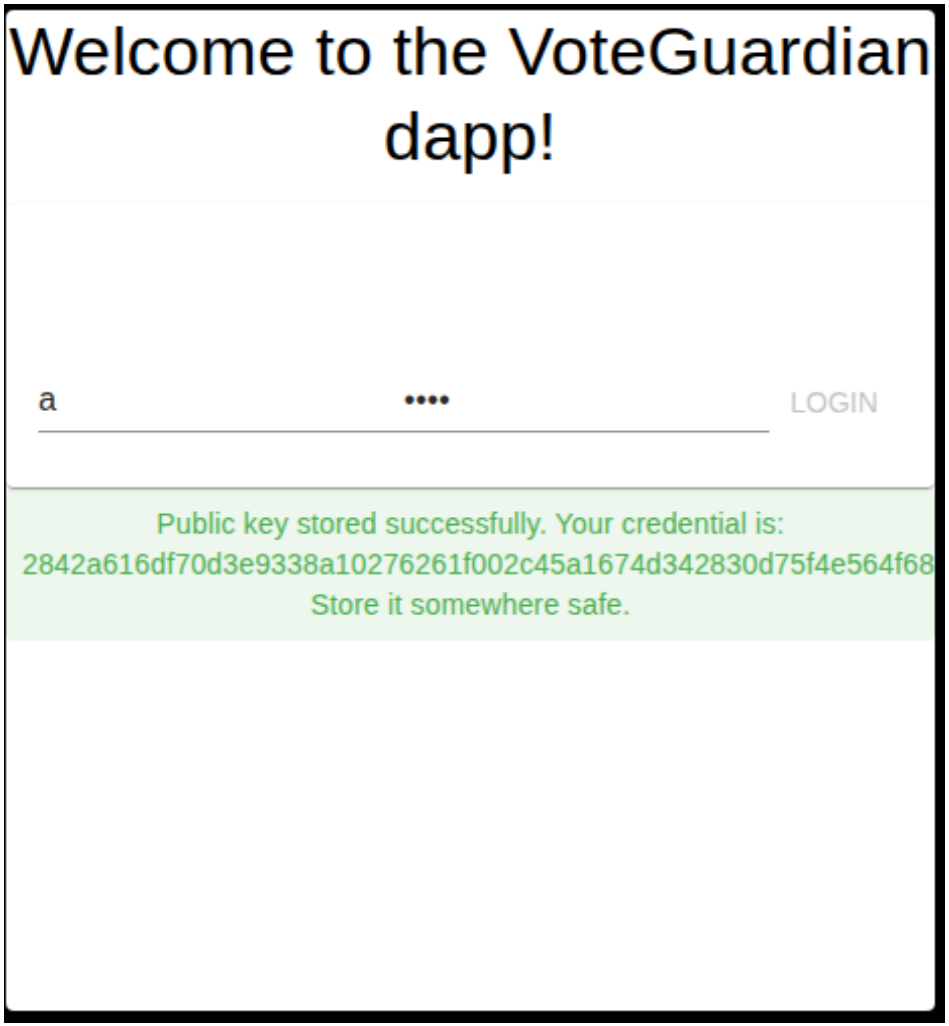

Figure: The login page

Use Cases

Participation in smart contract - Address registration

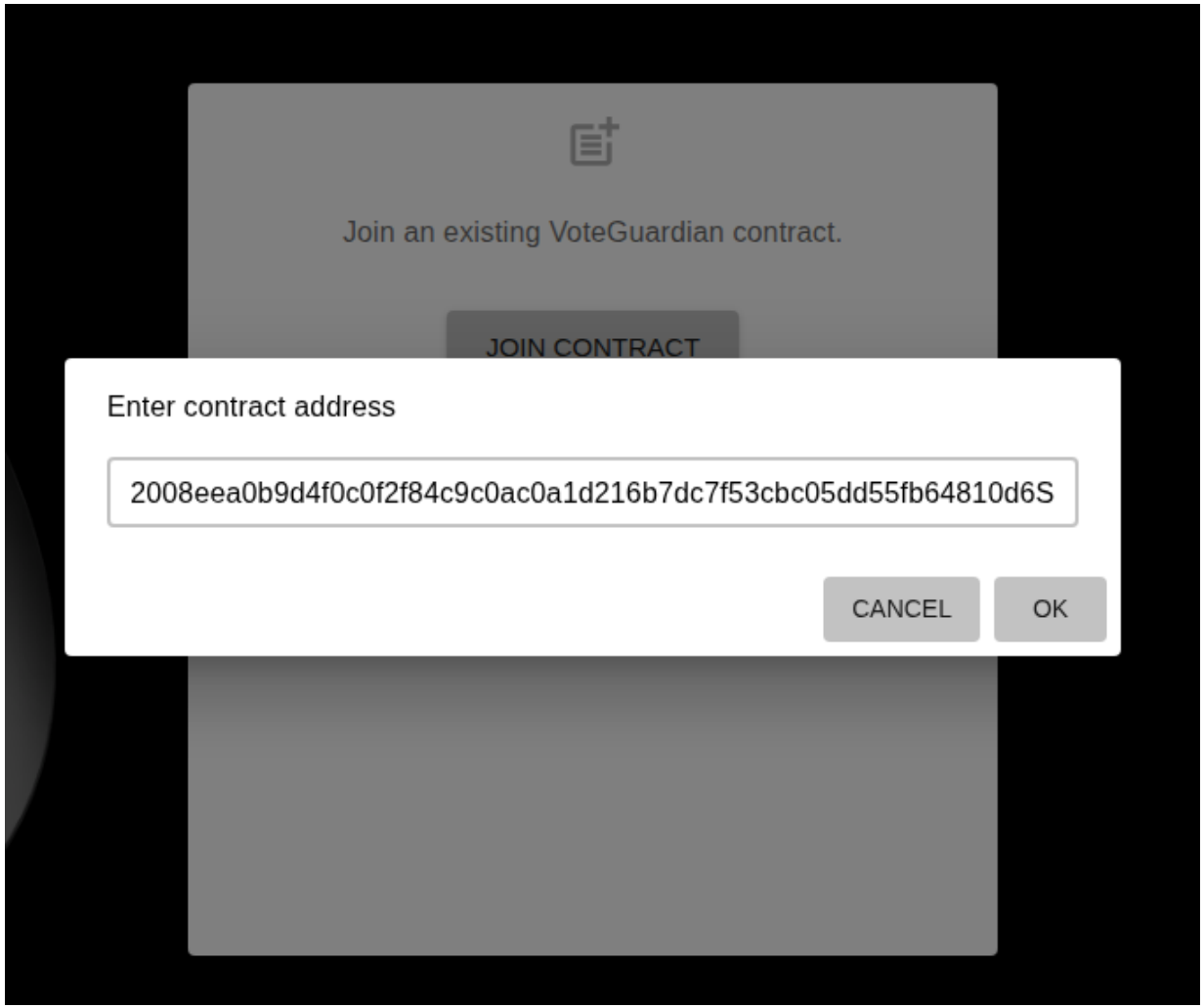

Figure: Address registration of the smart contract

Use Cases

Participation in smart contract - Secret key registration

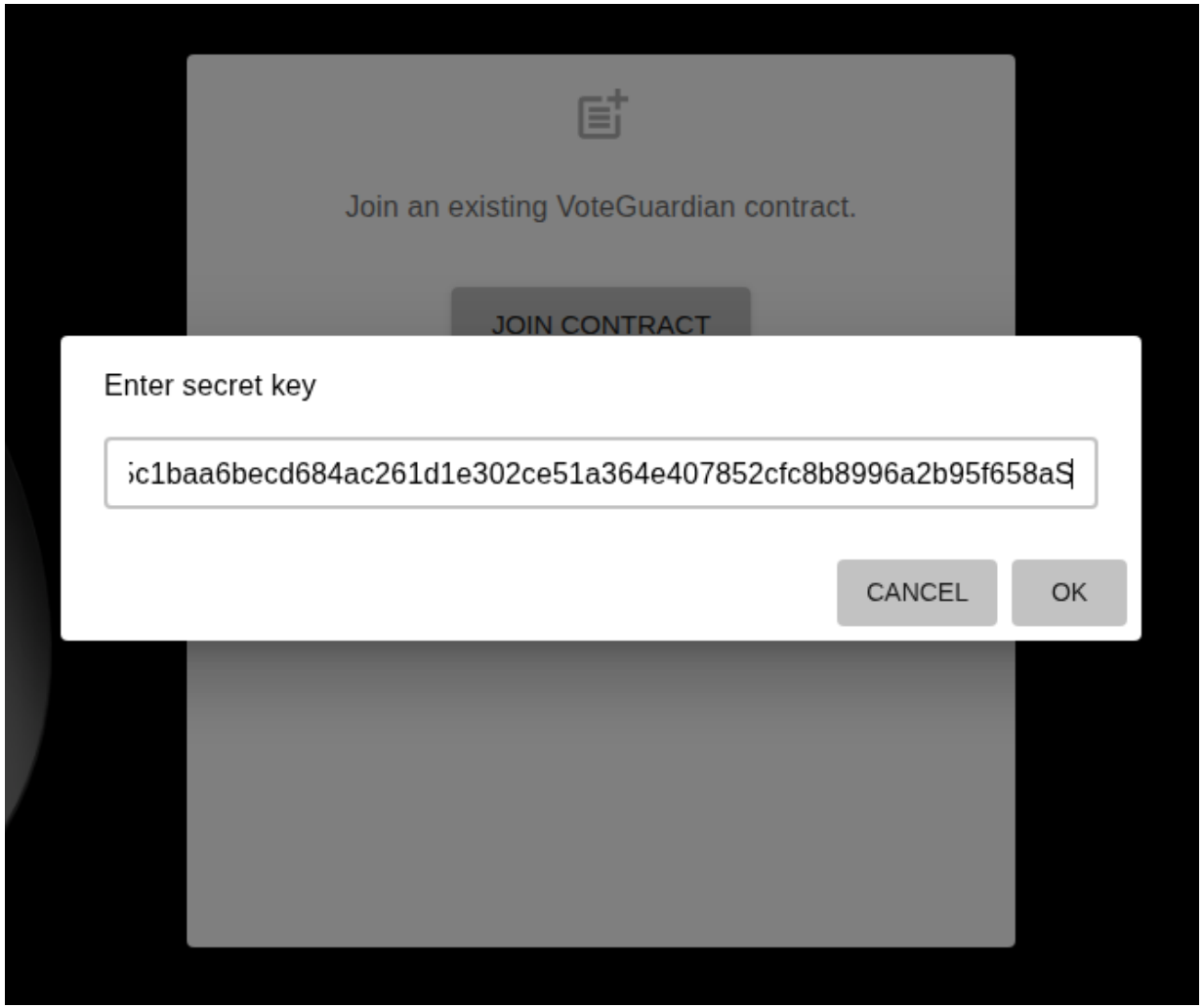

Figure: Entering the secret key

Use Cases

Initial Menu

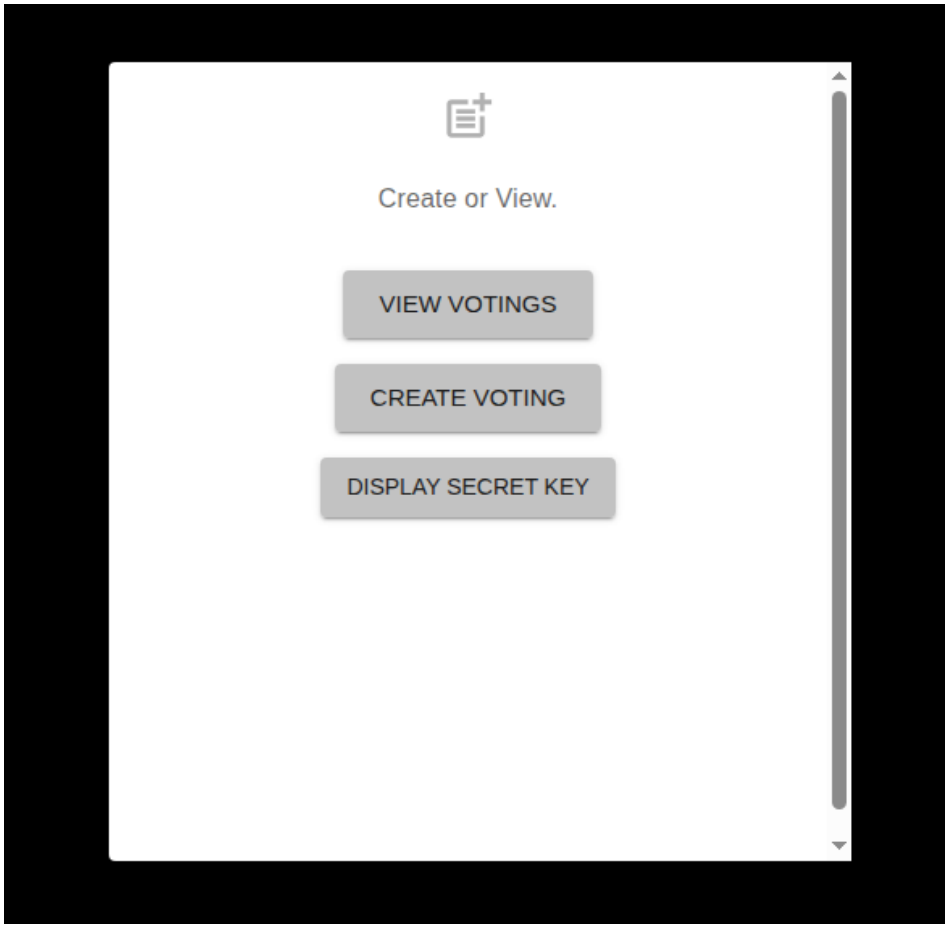

Figure: The initial menu

Use Cases

Available votings

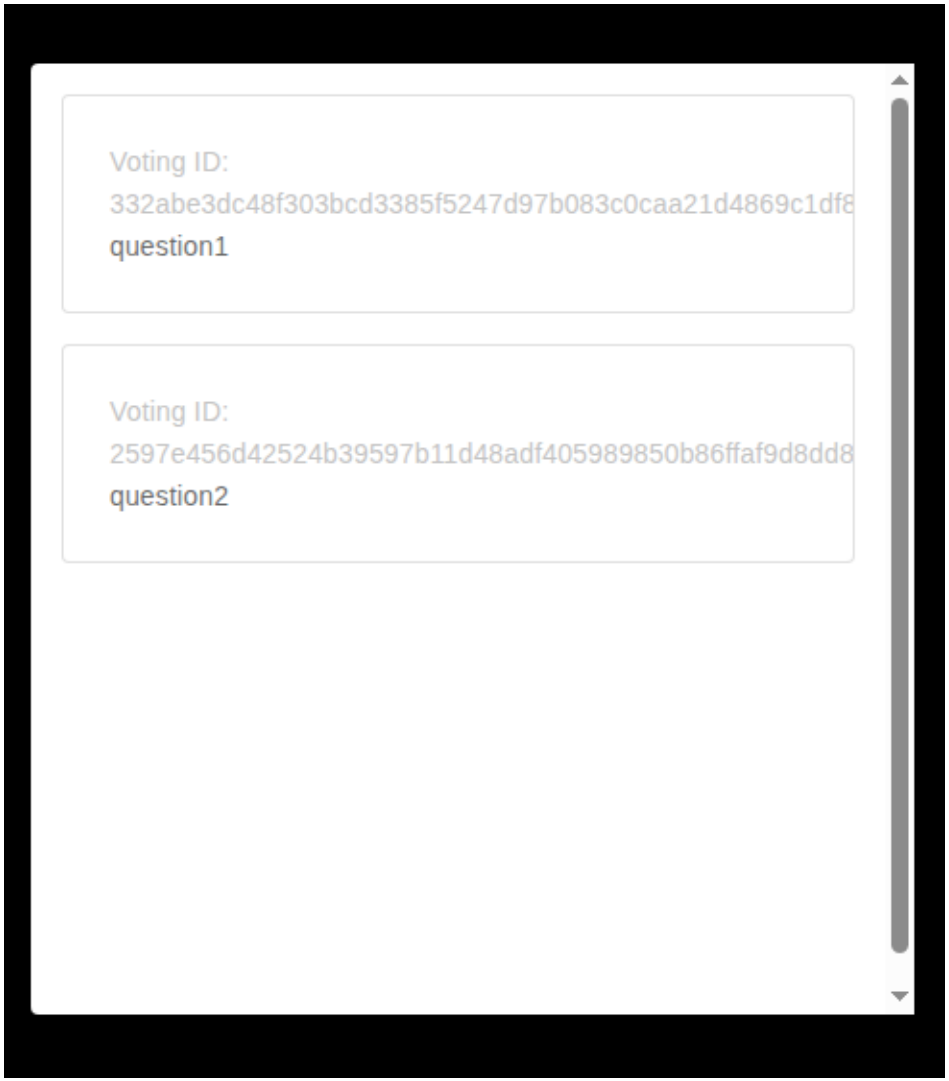

Figure: Available votings

Use Cases

Creating a voting

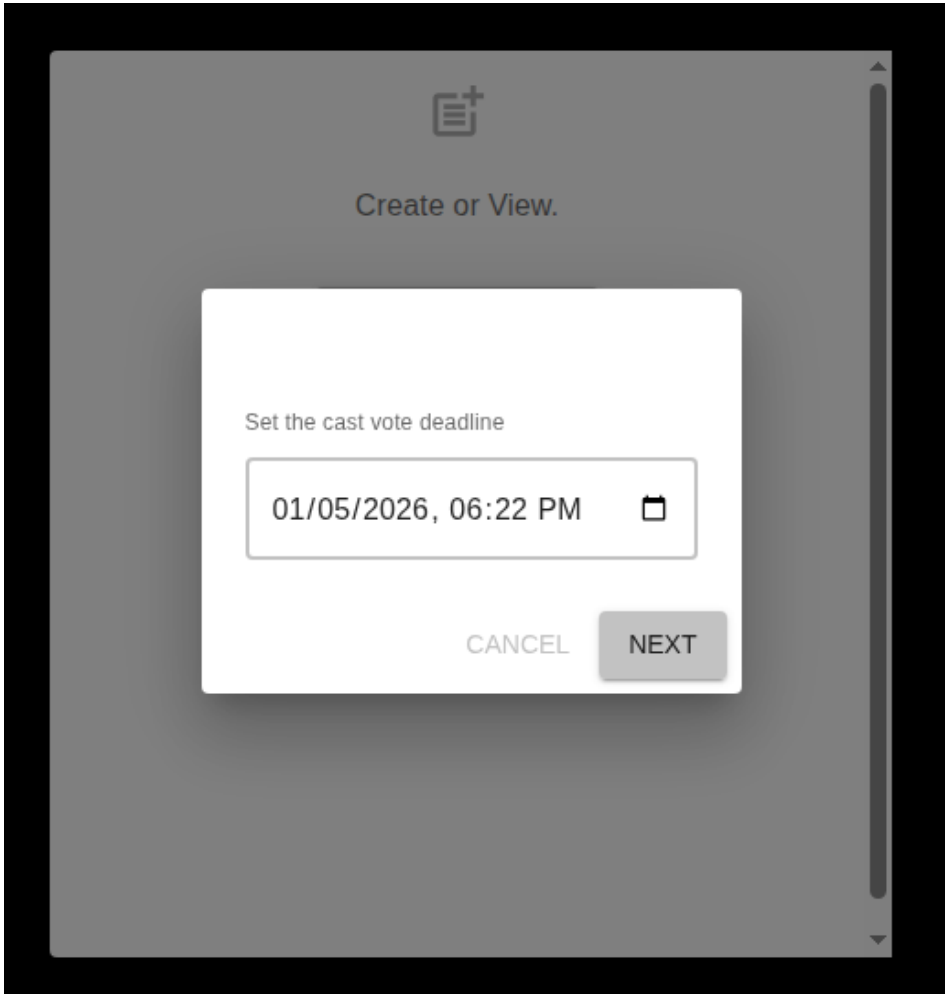

Figure: Setting a deadline for vote casting

Use Cases

Creating a voting

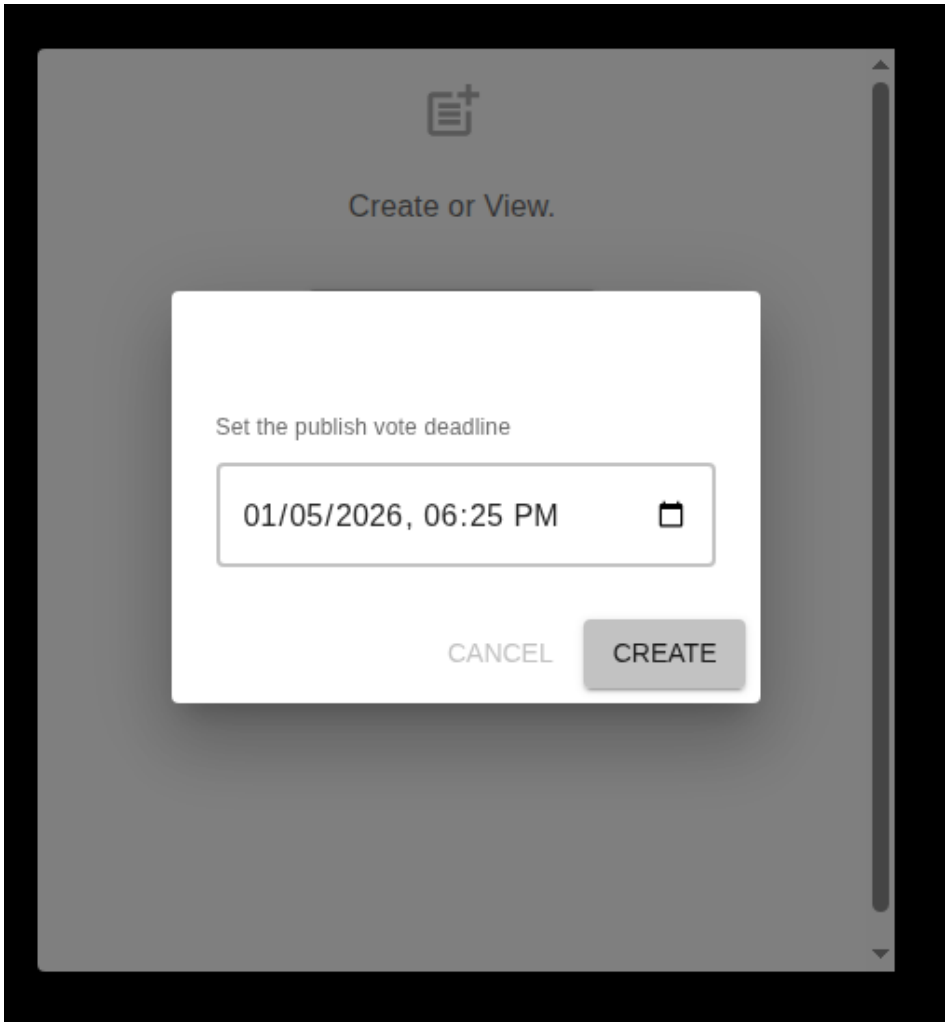

Figure: Setting a deadline for vote publishing

Use Cases

The initially empty voting

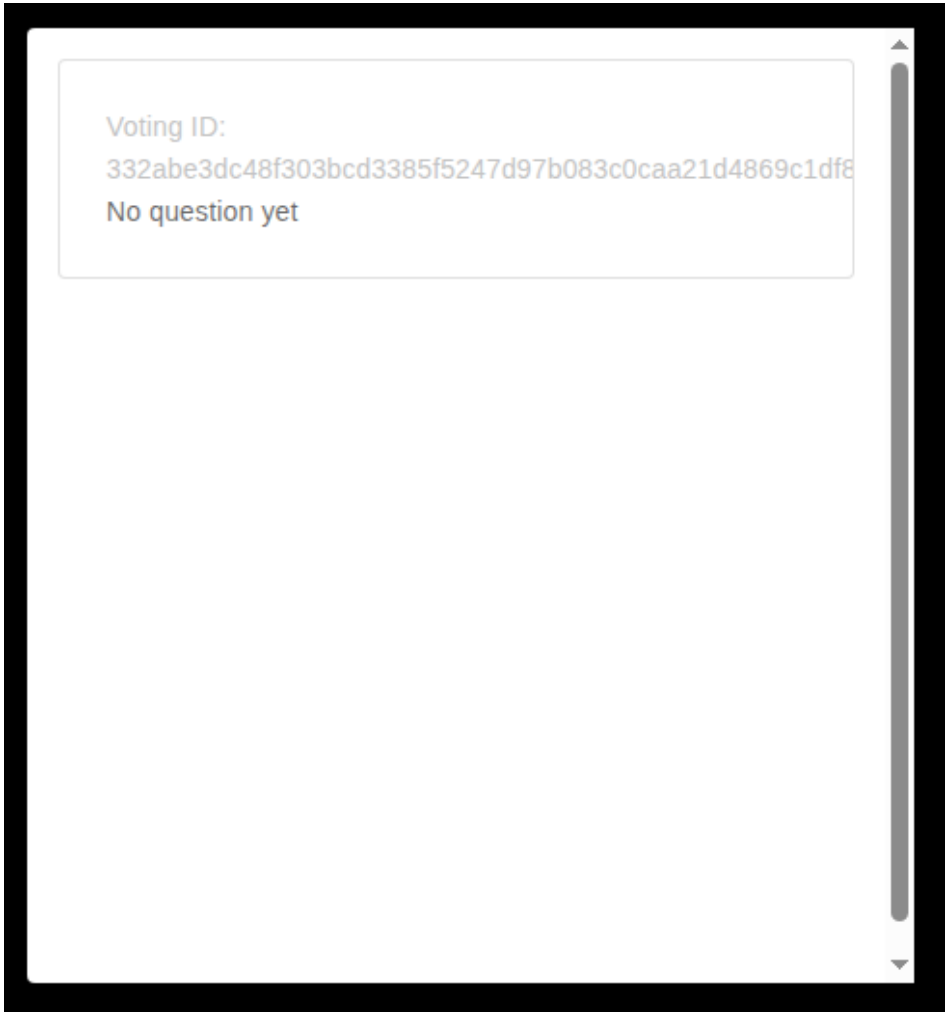

Figure: The initial blank voting

Use Cases

The voting menu

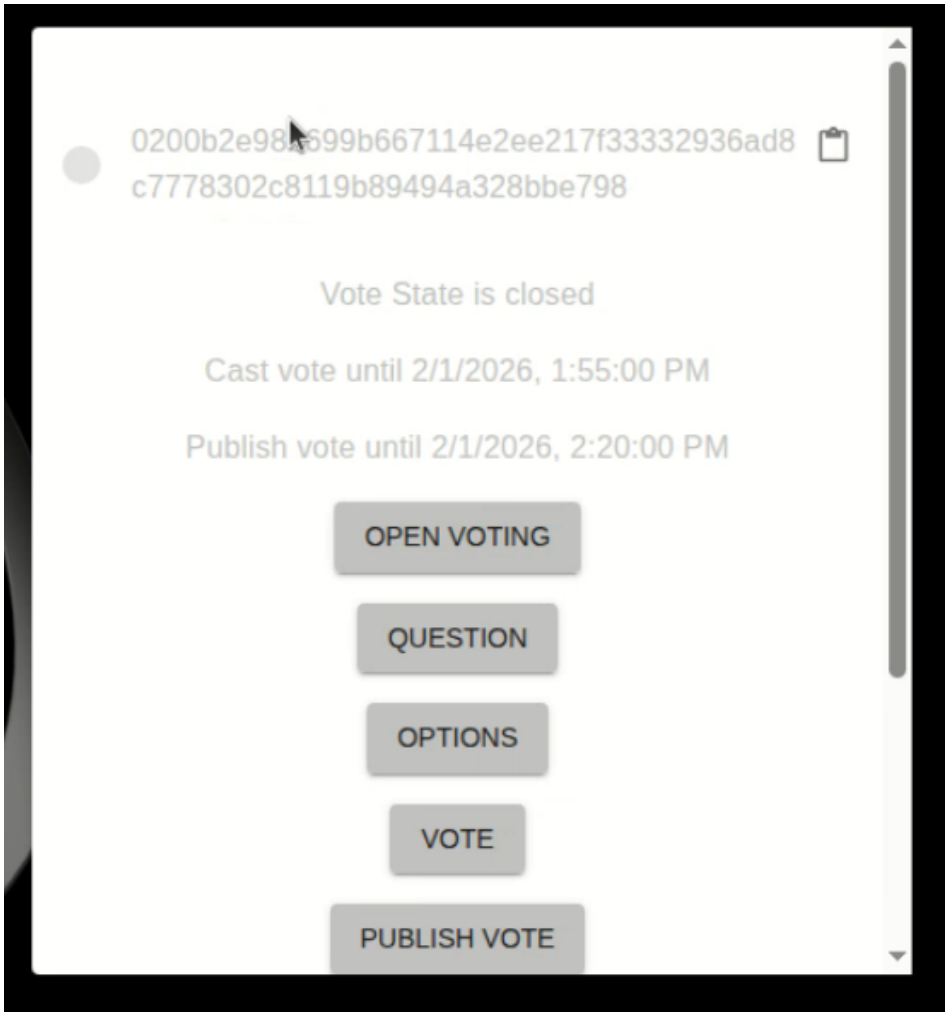

Figure: The voting menu

Use Cases

The voting menu

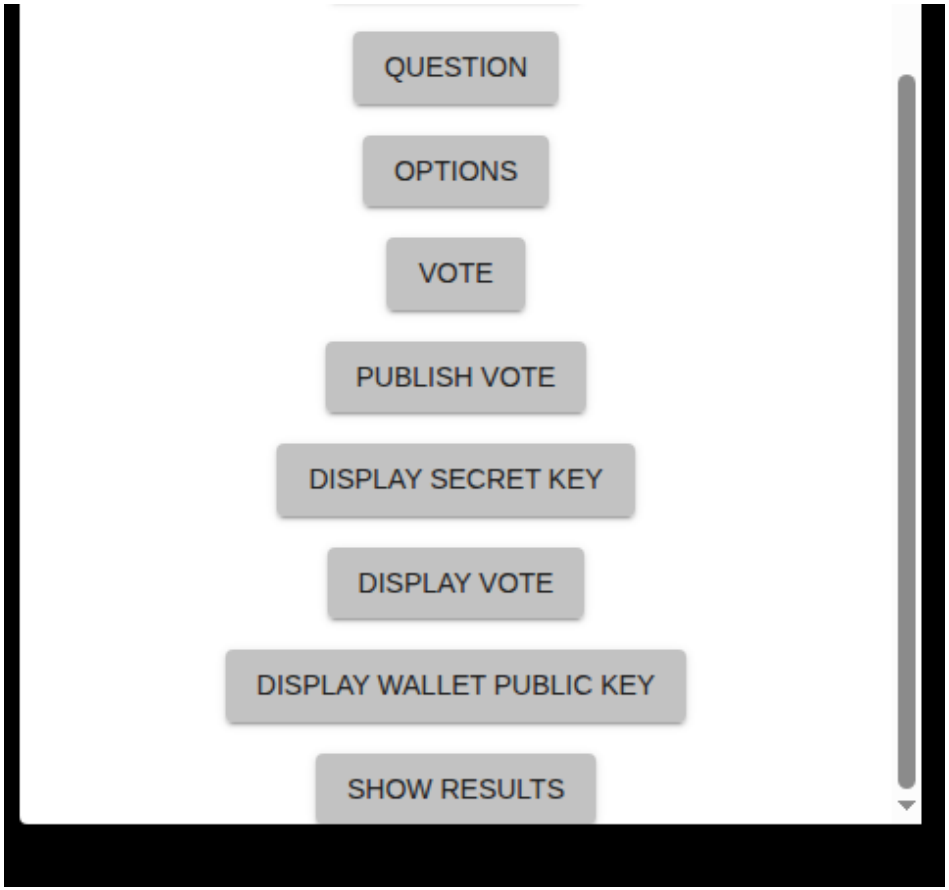

Figure: The voting menu

Use Cases

Add question

Figure: Add question

Use Cases

Add question - Sign from Lace Wallet

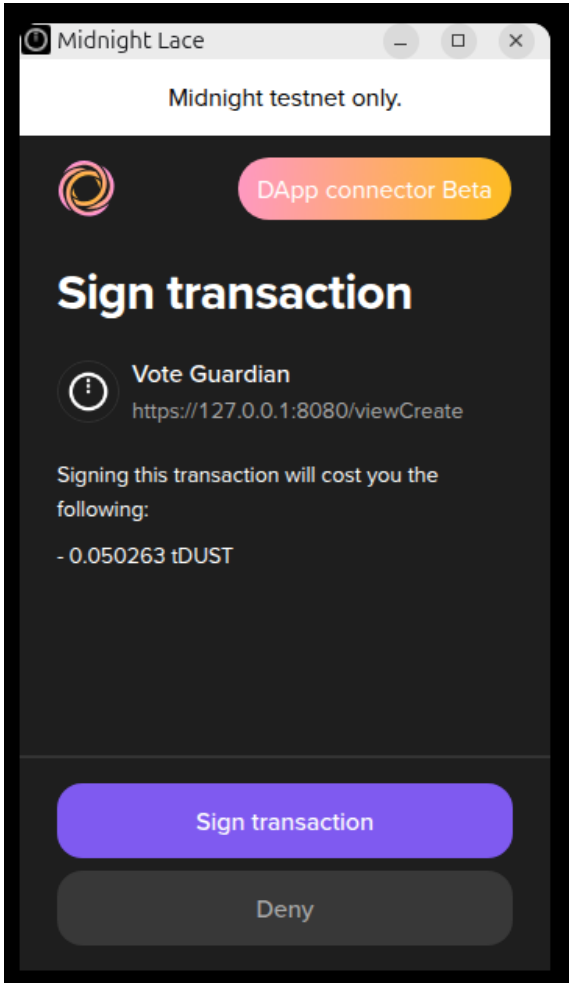

Figure: Signing a transaction from the lace wallet

Use Cases

Adding Options - Malicious Use

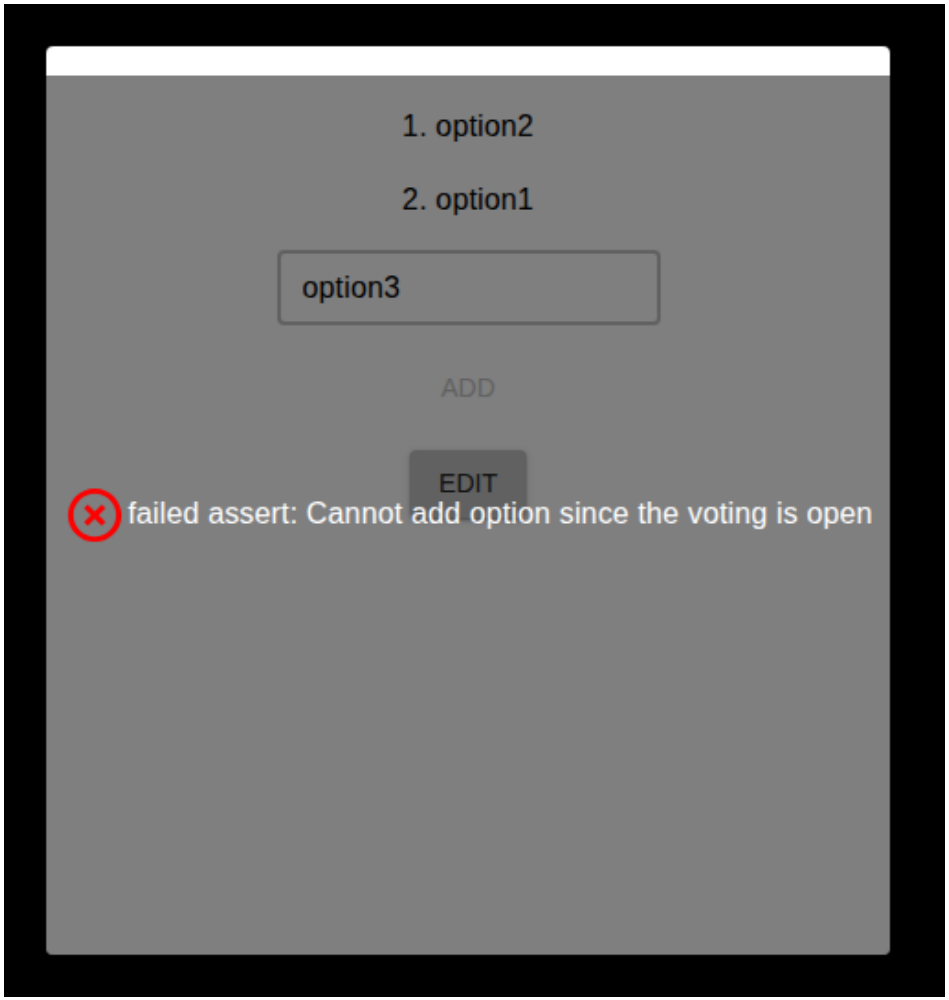

Figure: Adding an option to an OPEN voting

Figure: Casting a vote

Use Cases

Vote Casting - Results not available

After voting, the results are not visible. They become visible when the votes are published.

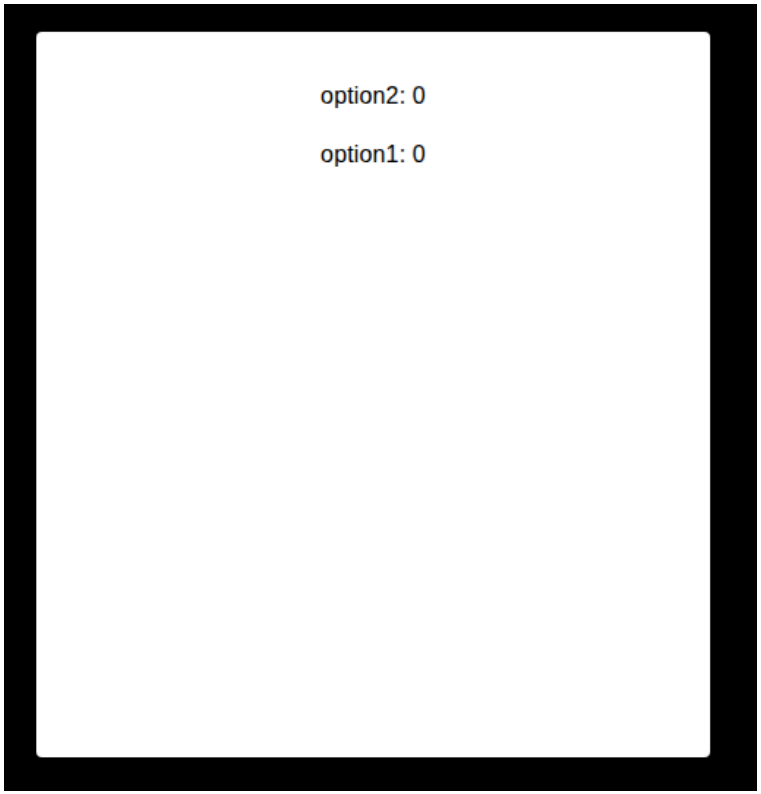

Figure: Results are empty during cast vote period

Use Cases

Voting - Double Voting

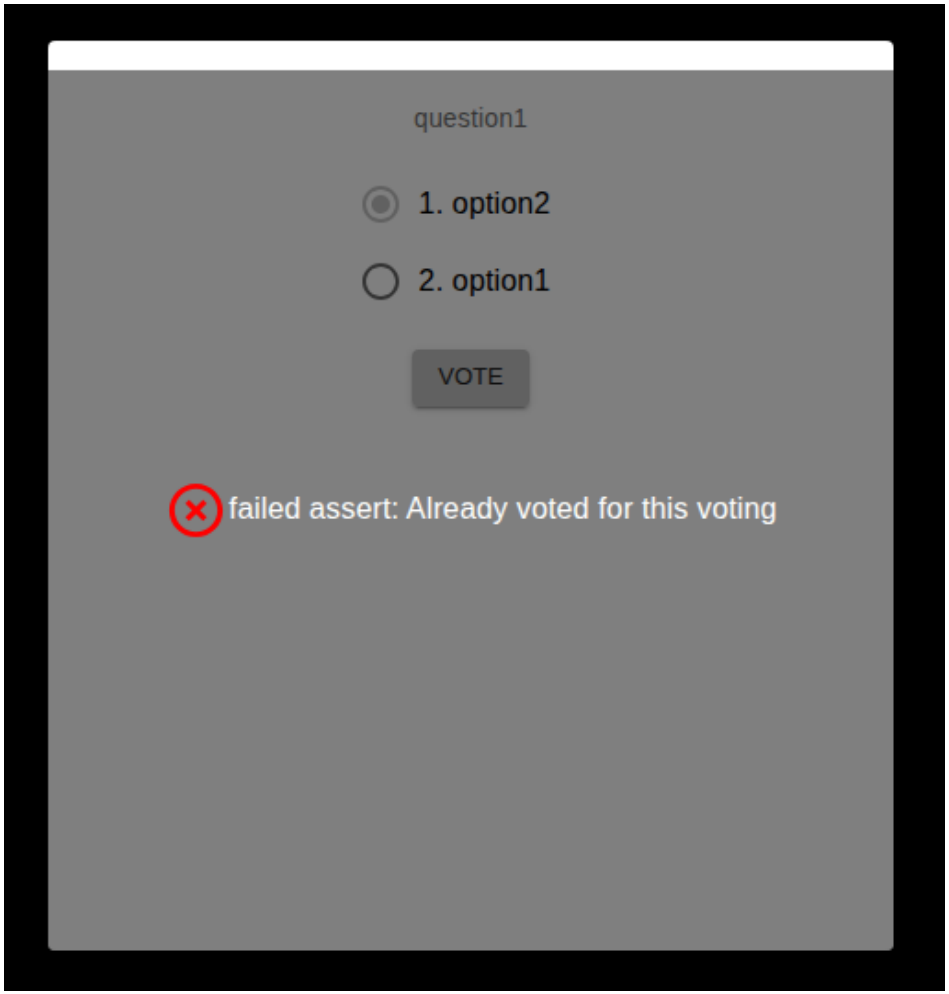

Figure: Double Voting

Use Cases

Voting - Invalid User

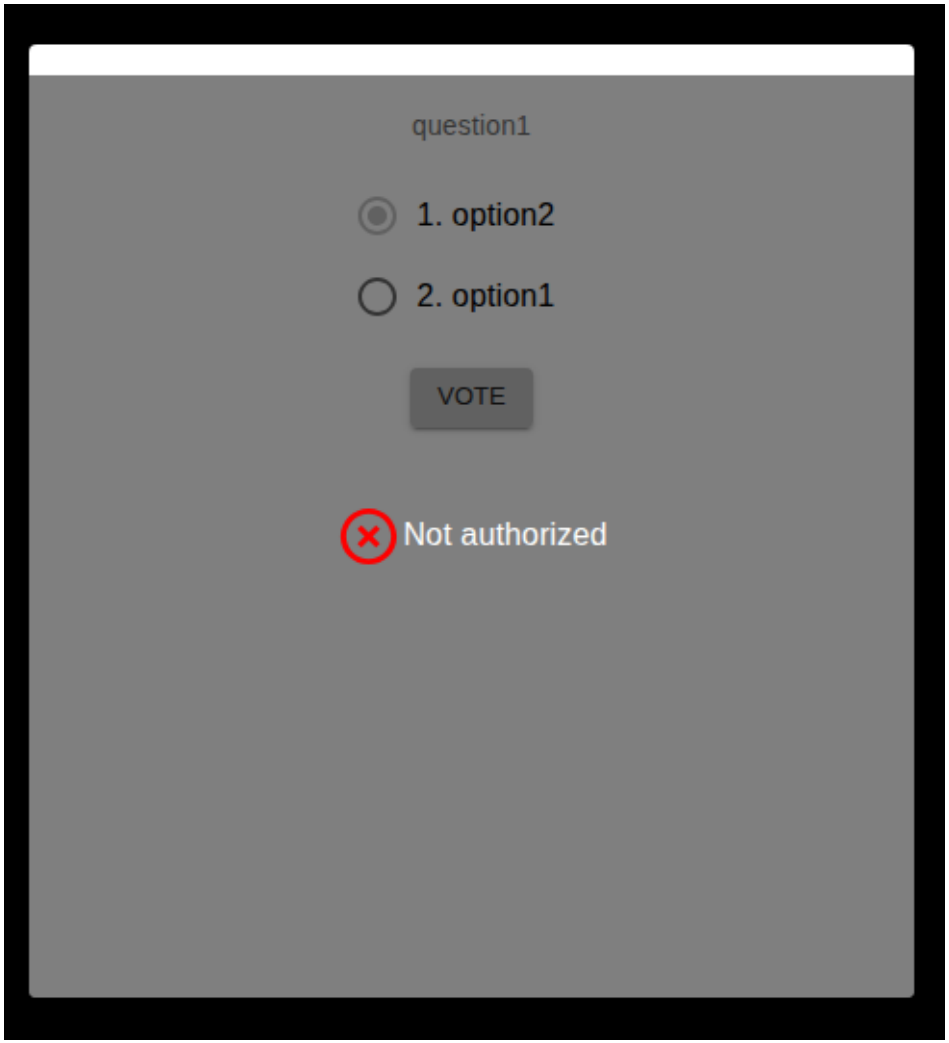

Figure: Invalid user

Use Cases

Publishing a vote - Overview of results

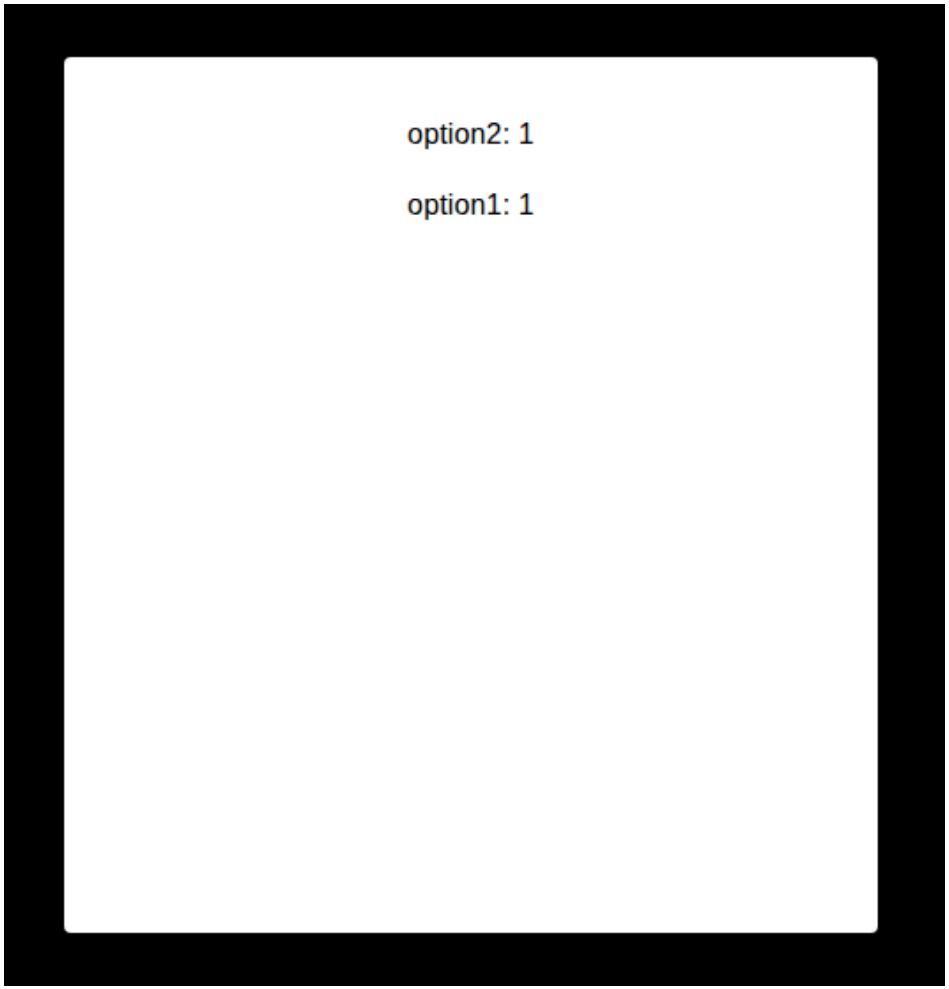

Figure: Results become visible after vote publication

Use Cases

Publishing a vote - Deadline expired

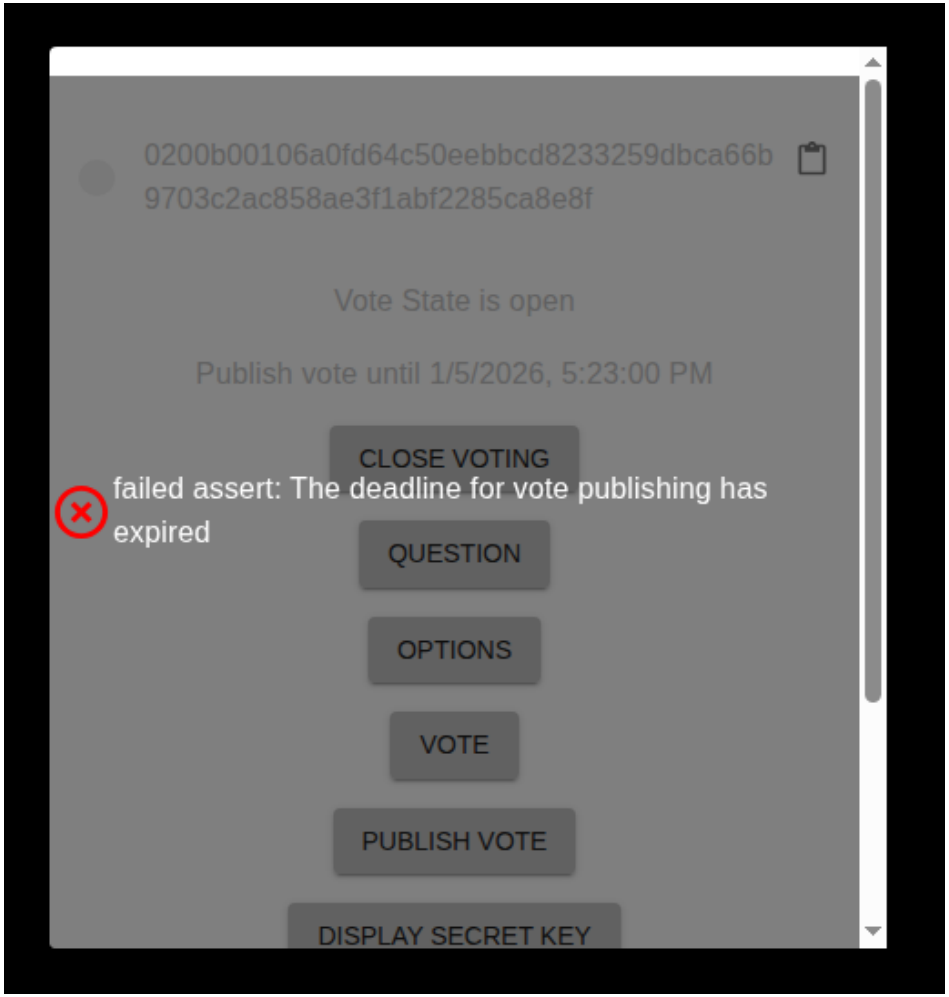

Figure: After the publication deadline, the vote is not registered

Use Cases

Publishing a vote - Changing a vote locally

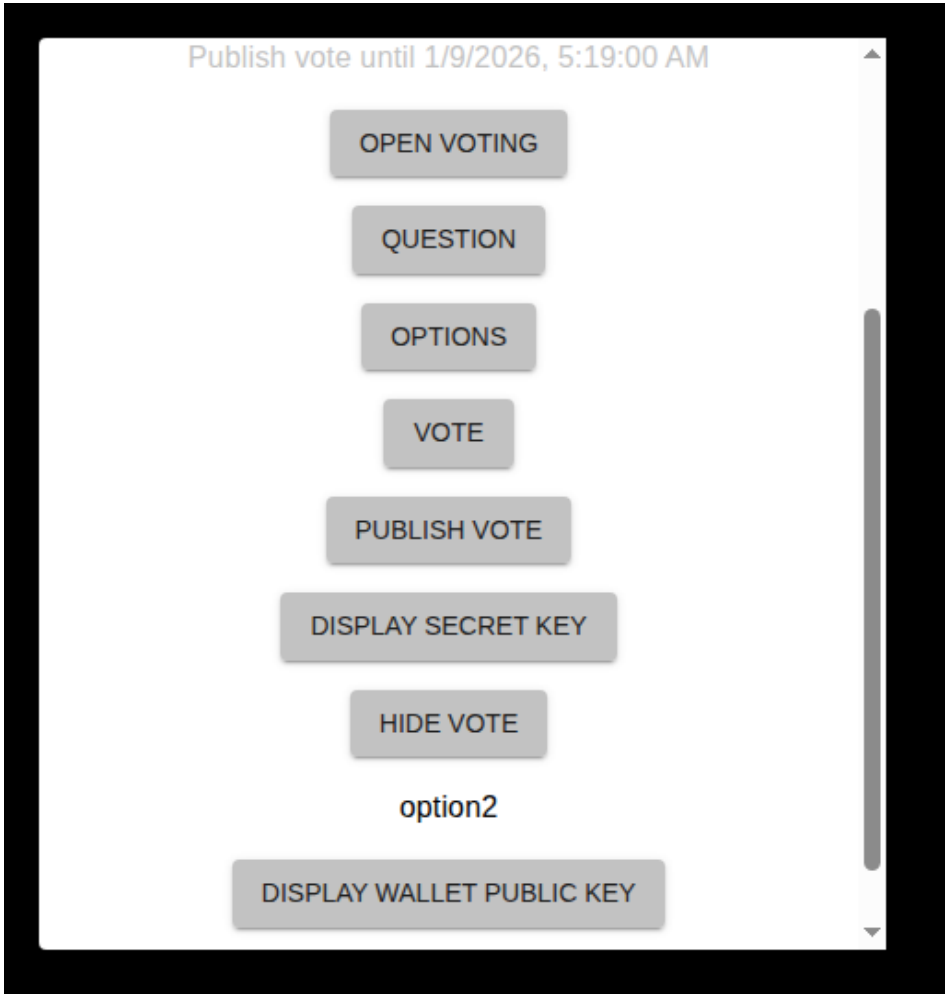

Figure: Change vote locally

Use Cases

Publishing a vote - Changing a vote locally

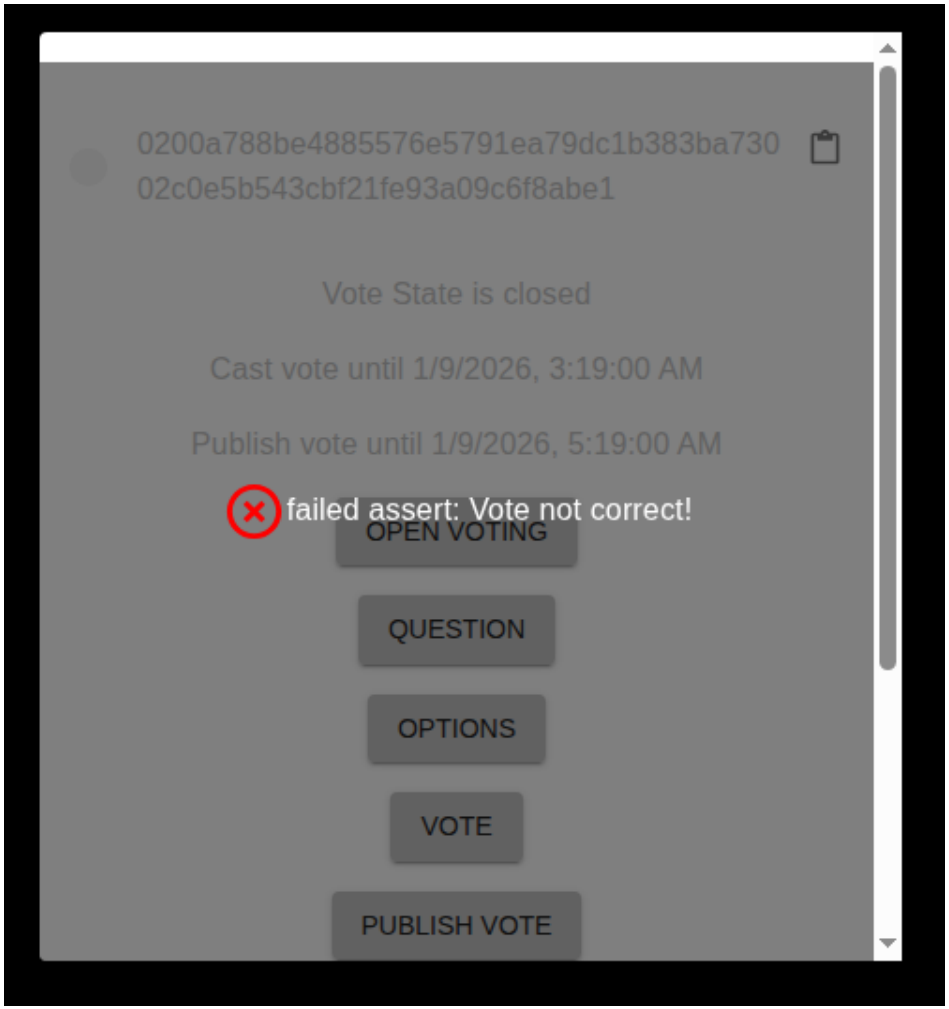

Figure: The user changed their vote after submitting it to another available one

Use Cases

Publishing a vote - Invalid vote

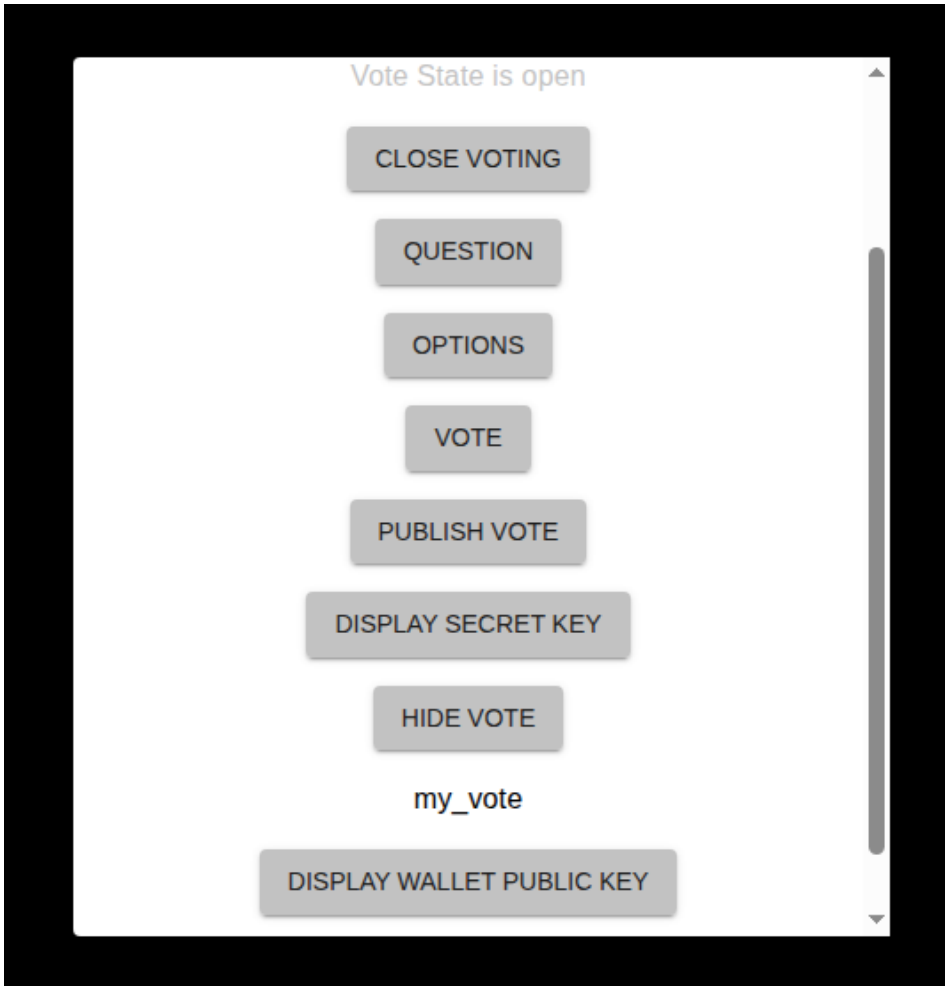

Figure: User votes for his own choice

Use Cases

Publishing a vote - Invalid vote

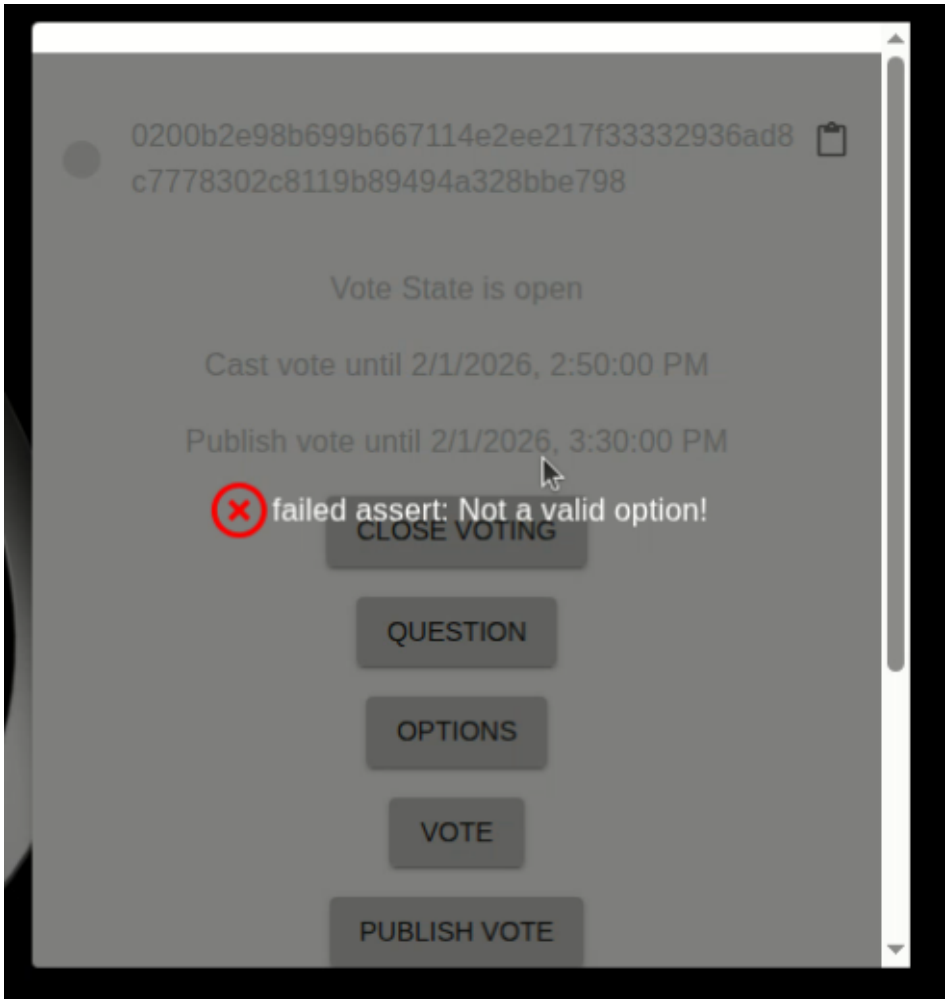

Figure: The user voted for his own choice and then published it

Limitations

- All students can vote in all votings
- One question per voting
- Central database - Increased centralization

Future Directions - Improvements

- Separation into votings per semester
- Multiple questions per voting
- Enhance interface via mobile app integration, such as ProofSpace
- Each voting a separate smart contract - Increased performance (This solution was tested but ultimately abandoned)
- Expand application scope beyond student voting – e.g., faculty elections

Thank you very much for your attention! I am available to answer any questions

The code of the application can be found here:
`https://github.com/JimV4/VoteGuardian2`