Software Engineering 1 - Decentralised Voting System

Zan Patryk
Czarnecki Jerzy
Małachowski Mateusz
Paciorek Michał

December 2024

Contents

1 Introduction

1.1 Goal of the System

The goal of this project is to design and implement a decentralised voting system using blockchain technology, specifically Ethereum. The purpose of the system is to provide a secure, transparent, and tamper-proof platform to conduct polls and voting processes. Using blockchain technology, the system ensures trustless interactions, immutability of votes, and the elimination of single points of failure. The overarching objective is to showcase how decentralised applications (dApps) can modernize voting systems by addressing issues of trust, accessibility, and scalability.

1.2 General Concept

The decentralised voting system operates entirely on blockchain-based smart contracts to manage every aspect of the voting process, including role assignments, poll creation, and vote count. The system involves the following key components:

- **User Authentication:** Users log in to the platform using their MetaMask accounts, which ensures secure authentication and interaction with the blockchain.
- **Role Management:** There are three types of roles in the system:
  - **Admin:** Responsible for assigning the *manager* role to selected users.
  - **Manager:** Authorized to create polls with unique access codes.
  - **User:** Regular participants who can join polls using the codes provided and cast votes.
- **Poll Creation and Voting:** Managers can create polls, each identified by a unique code. Users can join polls by adding these codes, which allows them to participate in voting. Votes are recorded as blockchain transactions, ensuring immutability and transparency.
- **NFT Minting:** Upon completion of a poll, participants can mint nonfungible tokens (NFTs) containing the poll results. These NFTs serve as a verifiable record of the result of the poll and a reward for participation. They can be later uploaded to retrieve data about ended poll (such as results).
- **Smart Contract Automation:** All operations, including role assignments, poll creation, vote submissions, and result calculations, are automated using smart contracts to ensure reliability and fairness.

1.3 Key Features

The system includes the following core features:

- **Blockchain Integration:** Utilization of Ethereum blockchain for secure and decentralised operations.
- **MetaMask Integration:** Simplified login and interaction with the blockchain through the MetaMask wallet.
- **Role-Based Access Control:** Hierarchical role management for administrators, managers, and users.
- **Transparency and Immutability:** Poll results are tamper-proof and publicly verifiable on the blockchain.
- **NFTs for Poll Results:** Incentivization and long-term verifiability through NFT-based poll results.
- **Scalability:** Potential to integrate Layer 2 solutions, such as zkSync, to improve scalability and reduce transaction costs.

This system exemplifies the transformative potential of decentralised technologies to build secure, transparent, and efficient voting mechanisms.

2 Requirements

2.1 User Stories

1. **Voting Participation and NFT Minting**

   As a user, I want to be able to cast my vote in a decentralised voting system so that my participation is recorded immutably on the blockchain.

   Acceptance Criteria:

   - I can see all active polls I am eligible for.
   - I can cast my vote securely using my blockchain wallet.
   - After the voting ends, I am able to mint an NFT that confirms my participation in the voting process.

2. **Vote Participation History**

   As a user, I want to view my voting history, including all the polls I have participated in, so that I can keep track of my involvement.

   Acceptance Criteria:

   - I can see a list of all past polls I have participated in.
   - Each entry includes details like poll name, date, and option to mint NFT.
   - I can verify the authenticity of the participation NFTs in my wallet and show the outcome.

3. **User Ticket Creation**

   As a user, I want to be able to create support tickets when I encounter issues or have questions regarding the voting system so that I can get assistance.

   Acceptance Criteria:

   - I can fill out a form to create a ticket, providing details about the issue or request.
   - I can view a list of all tickets I've submitted and track their statuses.

4. **Poll Creation**

   As a manager, I want to create new polls with specific parameters (e.g., start/end time, eligible participants, type of poll), so that I can initiate a voting process for a decentralised decision.

   Acceptance Criteria:

   - I can define the details of the vote, including title, description, voting duration, and type.
   - Once the poll is created, I receive a code, which allows voters to join the poll.
   - Once created, the poll is deployed on the blockchain.

5. **Adding Participants to Poll**

   As a manager, I want to add eligible participants to a vote pool, so that only those people can participate in the voting process.

   Acceptance Criteria:

   - I can share a unique code used to join the poll.
   - The system should prevent adding the same participant twice.
   - I have the ability to include myself in the vote pool.

6. **Ending Voting Process**

   As a manager, I want to end the voting process manually or let it end automatically when the deadline passes, so that the results can be finalized.

   Acceptance Criteria:

   - I can manually trigger the end of a voting process.
   - Voting results are finalized, and NFTs can be minted by users.
   - I receive a summary of the vote outcome after the voting ends.

7. **Vote Pool Management History**

   As a manager, I want to view the history of all the polls I have created and managed, so that I can keep track of the decisions I have facilitated.

   Acceptance Criteria:

   - I can access a list of all the polls I've created, both active and completed.
   - Each poll includes details such as start and end time, participants, and results.

8. **Manager Ticket Creation**

   As a manager, I want to create tickets when I encounter system issues or need help with managing vote pools or polls, so that I can get technical support or escalate problems.

   Acceptance Criteria:

   - I can create and submit tickets for issues related to vote creation, participants, or results.
   - I can view a list of all submitted tickets, with their status and responses.

9. **Admin Management of Roles**

   As an admin, I want to add or remove other users as admins or managers, so that I can control who has the authority to create and manage voting pools.

   Acceptance Criteria:

   - I can view a list of all current admins and managers.
   - I can promote or demote any user to the role of admin or manager.
   - Changes to roles should be recorded on the blockchain for transparency.

10. **Access to All Voting Events**

    As an admin, I want access to the details of all voting pools in the system, regardless of whether I created them or not, so that I have full oversight of the voting activities.

    Acceptance Criteria:

    - I can view details (status of poll, number of participants, name, start/end date) of all active and historical voting pools, except the outcome of polls.

11. **Admin Ticket Oversight**

    As an admin, I want to view, manage, and resolve tickets created by users and managers so that I can ensure that issues are addressed and system functionality remains smooth.

    Acceptance Criteria:

    - I can view all open tickets submitted by users and managers.
    - I can respond to tickets and update their statuses to keep users informed.

2.2 Non-Functional Requirements

The decentralised voting system must meet the following non-functional requirements to ensure its robustness, scalability, and usability:

- **Scalability:** The system should support a large number of users and concurrent transactions without significant performance degradation. This includes accommodating the growth of polls and participants.
- **Performance:** The system should process transactions and smart contract interactions efficiently, ensuring low latency for actions such as voting, role assignment, and poll creation.
- **Security:** The system must ensure the integrity and confidentiality of user data. It should employ secure authentication mechanisms through MetaMask and robust smart contract designs to prevent vulnerabilities like reentrancy or unauthorized access.
- **Reliability:** The system should guarantee availability and fault tolerance. Users should be able to access the system at any time, and it should recover gracefully from failures.
- **Usability:** The user interface must be intuitive and accessible, enabling seamless interactions for all roles (users, managers, admins) via MetaMask integration.
- **Interoperability:** The system should work seamlessly with blockchain networks, wallets like MetaMask.
- **Maintainability:** The system must be designed for easy updates and modifications to user operations (KYC verification for different clients) and the frontend without disrupting ongoing operations.
- **Transparency:** The system should maintain transparency by making the poll results and voting process accessable for poll participants while preserving voter anonymity.
- **Cost-Effectiveness:** The system should optimize gas usage within the constraints of the Ethereum mainnet to reduce transaction using gas-optimization techniques (such as low-level calls)
- **Compliance:** The system should adhere to relevant legal and ethical standards for decentralised systems and data protection.

3 Design Documentation

3.1 Architecture Design

The system consists of three main parts—the user, the smart contract, and the database. The *user* is the customer who performs actions or interacts with the service. This includes logging in using their MetaMask accounts, participating in polls, and minting NFTs upon poll completion. Users can be assigned roles, such as *manager* or *admin*, which govern their permissions within the system. Managers, for example, are granted the ability to create polls with unique codes, which are then shared with other users, enabling them to participate and cast their votes. The *smart contract* serves as the backbone of the entire system, handling transactions, enforcing the role-based permissions, and managing all interactions related to the voting and NFT minting processes. The contract ensures transparency and eliminates the need for centralized control, with all data stored securely on the blockchain. The *database* is used solely to store and retrieve support tickets that may arise and require resolution by the system administrators.

The user interacts with the system via the graphical interface of the application on the front end, which follows structured message schemas to communicate with the back end. The application seamlessly integrates blockchain technology, allowing users to directly interact with smart contracts for activities such as voting, role management, and NFT minting. In almost all cases, the system retrieves data directly from the blockchain, thus minimizing the need to store any significant data on the server itself. For example, the poll results and role-based configurations are stored on-chain, ensuring trust and immutability. After a poll concludes, participants can mint NFTs, which include metadata detailing the poll results, offering both a digital collectible and a transparent record of the event. This comprehensive architecture combines decentralization, user empowerment, and minimal reliance on centralized infrastructure to deliver a robust and transparent solution.

3.2 Message Schemas

The following message schemas outline the API endpoints, request/response formats, and error handling mechanisms.

3.2.1 Authentication

Connect with MetaMask

- **Endpoint:** `GET /connect`
- **Description:** Initiates the connection process with the user's MetaMask wallet. Verifies wallet ownership and retrieves role-based access from the blockchain.
- **Response:**

Listing 1: Successful Response Example

```
{
    "message": "Wallet connected successfully.",
    "user_address": "0x123abc...",
    "role": "user"
}
```

- **Error Response:**

Listing 2: Error Response Example

```
{
    "error": "Failed to connect with MetaMask. Please try again."
}
```

3.2.2 User Endpoints

View Active Polls

- **Endpoint:** `GET /polls`
- **Description:** Fetches a list of active polls available for participation.
- **Response:**

Listing 3: Active Polls Response Example

```
[
    {
        "poll_id": 301,
        "poll_name": "Board Elections 2024",
        "poll_options": ["Candidate A", "Candidate B", "Candidate C"],
        "start_date": "2024-12-01",
        "end_date": "2024-12-10",
        "status": "Active"
    },
    ...
]
```

Join a Poll Using Code

- **Endpoint:** `POST /polls/join`
- **Description:** Allows users to join a poll using a special access code.
- **Request Body:**

Listing 4: Join Poll Request Example

```
{
    "poll_code": "ABCD1234"
}
```

- **Response:**

Listing 5: Join Poll Success Response Example

```
{
    "message": "Successfully joined the poll."
}
```

- **Error Response:**

Listing 6: Join Poll Error Response Example

```
{
    "error": "Invalid poll code or transaction failed."
}
```

Cast a Vote

- **Endpoint:** `POST /polls/{poll_id}/vote`
- **Description:** Allows users to cast a vote in a specific poll.
- **Request Body:**

Listing 7: Cast Vote Request Example

```
{
    "vote_option": "Candidate A",
    "wallet_signature": "user-wallet-signature"
}
```

- **Response:**

Listing 8: Cast Vote Success Response Example

```
{
    "message": "Vote successfully submitted."
}
```

- **Error Response:**

Listing 9: Cast Vote Error Response Example

```
{
    "error": "Transaction failed. Please check your wallet connection."
}
```

Mint Participation NFT

- **Endpoint:** `POST /polls/{poll_id}/mint`
- **Description:** Allows users to mint an NFT to represent their participation.
- **Response:**

Listing 10: Mint NFT Success Response Example

```
{
    "message": "NFT minted successfully.",
    "nft_details": "https://nft-platform.example.com/nft/123"
}
```

- **Error Response:**

Listing 11: Mint NFT Error Response Example

```
{
    "error": "Failed to mint NFT. Please try again later."
}
```

3.2.3 Manager Endpoints

Create a Poll

- **Endpoint:** `POST /polls`
- **Description:** Creates a new poll for users to join.
- **Request Body:**

Listing 12: Create Poll Request Example

```
{
    "poll_title": "Annual Committee Elections",
    "poll_description": "Voting for the 2024 committee members",
    "poll_options": ["Candidate A", "Candidate B", "Candidate C"],
    "start_time": "2024-12-01T10:00:00Z",
    "end_time": "2024-12-10T17:00:00Z",
}
```

- **Response:**

Listing 13: Create Poll Success Response Example

```
{
    "poll_id": 301,
    "poll_code": "ABCD1234",
    "message": "Poll created successfully."
}
```

- **Error Response:**

Listing 14: Create Poll Error Response Example

```
{
    "error": "Transaction failed. Could not create poll."
}
```

End Voting Process

- **Endpoint:** `POST /polls/{poll_id}/end`
- **Description:** Ends a poll and locks further voting.
- **Response:**

Listing 15: End Poll Success Response Example

```
{
    "poll_id": 301,
    "ended_automatically": Yes,
    "message": "Poll has ended."
}
```

- **Error Response:**

Listing 16: End Poll Error Response Example

```
{
    "error": "Failed to end poll. Please try again."
}
```

View Poll Management History

- **Endpoint:** `GET /managers/polls`
- **Description:** Fetches the poll history managed by a manager.
- **Response:**

Listing 17: Poll History Response Example

```
[
    {
        "poll_id": 301,
        "poll_title": "Annual Committee Elections",
        "start_time": "2024-12-01T10:00:00Z",
        "end_time": "2024-12-10T17:00:00Z",
        "poll_options": ["Candidate A", "Candidate B", "Candidate C"],
        "participant_count": 500
    }
]
```

Create a Support Ticket

- **Endpoint:** `POST /tickets`
- **Description:** Creates a new support ticket related to a managed poll.
- **Request Body:**

Listing 18: Create Support Ticket Request Example

```
{
    "issue_description": "Technical issue with poll creation",
    "related_poll_id": 301,
    "attachments": ["attachment2_url"]
}
```

- **Response:**

Listing 19: Create Support Ticket Success Response Example

```
{
    "ticket_id": 401,
    "message": "Ticket created successfully."
}
```

- **Error Response:**

Listing 20: Create Support Ticket Error Response Example

```
{
    "error": "Failed to create ticket. Please try again."
}
```

3.2.4 Admin Endpoints

Manage User Roles

- **Endpoint:** `POST /users/roles`
- **Description:** Adds or removes roles for users.
- **Request Body:**

Listing 21: Manage User Roles Request Example

```
{
    "target_wallet_address": "0x456def...",
    "new_role": "manager" // Options: admin, manager, user
}
```

- **Response:**

Listing 22: Manage User Roles Success Response Example

```
{
    "message": "Role updated successfully."
}
```

- **Error Response:**

Listing 23: Manage User Roles Error Response Example

```
{
    "error": "Transaction failed. Role change unsuccessful."
}
```

View All Polls

- **Endpoint:** `GET /polls/all`
- **Description:** Retrieves a list of all polls in the system.
- **Response:**

Listing 24: View All Polls Response Example

```
[
    {
        "poll_id": 301,
        "poll_title": "Annual Committee Elections",
        "status": "Completed",
        "participant_count": 500
    }
]
```

Manage Support Tickets

- **Endpoint:** `GET /tickets`
- **Description:** Fetches all support tickets for review.
- **Response:**

Listing 25: View Support Tickets Response Example

```
[
    {
        "ticket_id": 400,
        "creator_role": "user",
        "issue_description": "Unable to submit vote",
        "status": "Pending"
    }
]
```

Respond to a Support Ticket

- **Endpoint:** `POST /tickets/{ticket_id}/respond`
- **Description:** Responds to a support ticket.
- **Request Body:**

Listing 26: Respond to Support Ticket Request Example

```
{
    "response": "Issue is being investigated."
}
```

- **Response:**

Listing 27: Respond to Support Ticket Success Response Example

```
{
    "message": "Response added successfully."
}
```

- **Error Response:**

Listing 28: Respond to Support Ticket Error Response Example

```
{
    "error": "Failed to respond to the ticket. Please try again."
}
```

3.3 Diagrams

This section contains diagrams related to various parts of the system, which provide a graphical overview of the key functionalities implemented in system components.

3.3.1 Use Case Diagram

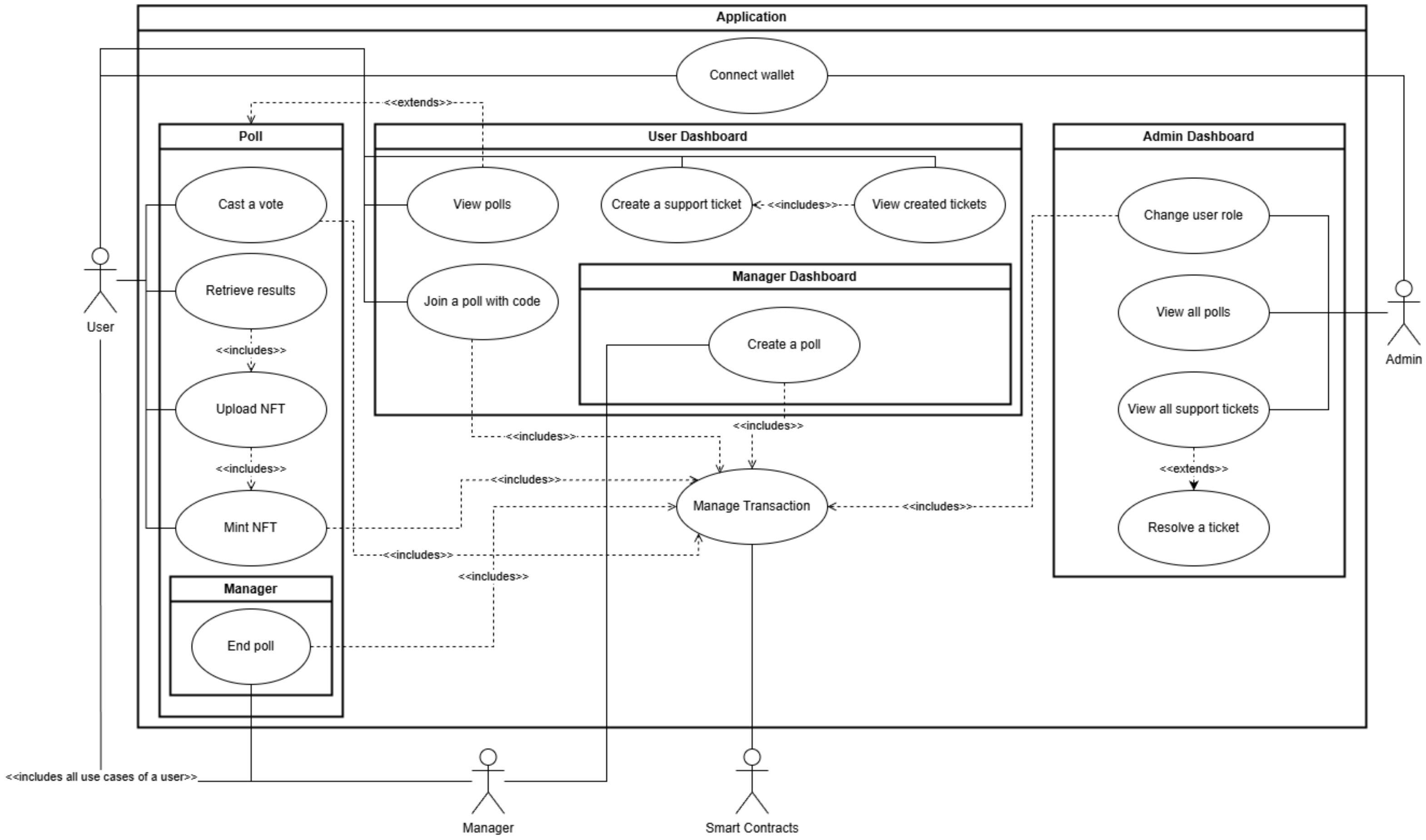

Figure 1: Use Case Diagram

This use case diagram represents the interactions between various actors and the decentralised voting system. Users log in using their MetaMask accounts, enabling authenticated access to the system.

- **Actors:**
  - **User:** A standard participant who interacts with the system. Users can:
    - View polls and join a poll using a unique code provided by the manager.
    - Cast a vote and retrieve the results of the poll.
    - Upload and mint an NFT after the poll concludes, where the NFT contains the poll's results.
    - Connect their wallet to interact with the blockchain.
  - **Manager:** A privileged user role assigned by the admin. Managers can:
    - Create polls with unique codes.
    - End polls manually or after the voting period is over.
  - **Admin:** The administrator manages user roles and system configurations. Admins can:
    - Change user roles (e.g., assign manager roles).
    - View all polls and support tickets.
    - Resolve support tickets as needed.
  - **Smart Contracts:** These handle all blockchain-based interactions, including voting, role assignment, poll creation, and transactions, ensuring transparency and security.
- **Use Cases and Interactions:**
  - The system allows users to cast votes in a poll and view results.
  - The "*Upload NFT*" and "*Mint NFT*" processes are included as extensions to the "*Retrieve Results*" use case, allowing users to mint NFTs containing poll data.
  - Managers create polls, which include obtaining unique codes for user access.
  - Admins extend their control through actions such as viewing polls, resolving tickets, and changing roles.
  - All transactions are managed on the blockchain via smart contracts, ensuring immutability and trust in the voting process.

3.3.2 State Diagram

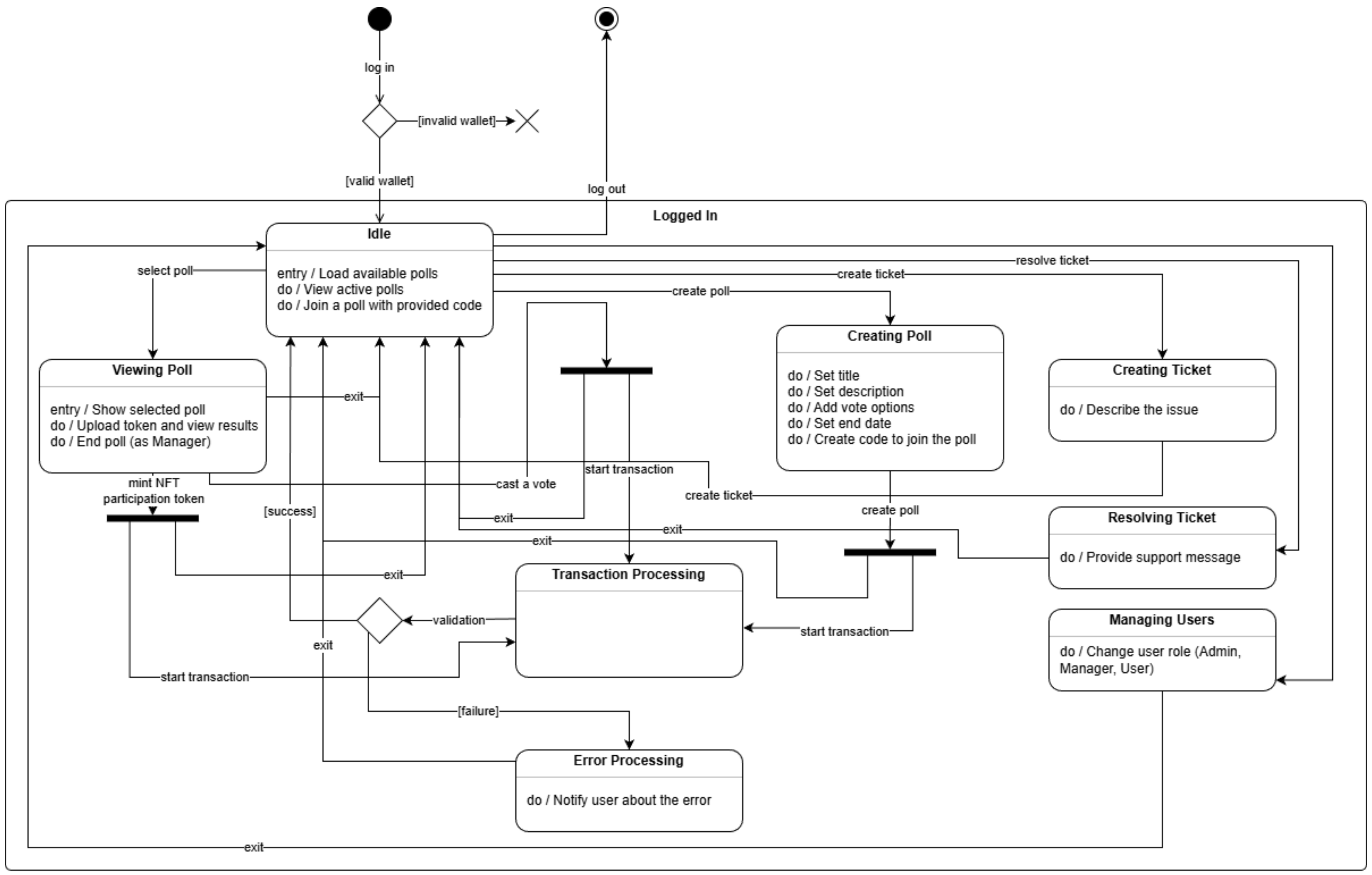

Figure 2: State Diagram

The state diagram illustrates the dynamic behavior of the decentralised voting system, focusing on various states the application can transition through, based on user interactions and system processes.

- **Initial State:**
  - Users begin at the *log in* state. If the wallet validation fails (invalid wallet), the user is blocked from further interaction.
  - A valid wallet allows the user to transition to the **Idle** state.
- **Idle State:**
  - Upon entering the idle state, the system loads available polls and displays active polls to the user.
  - Users can perform the following actions:
    - **View a Poll:** Users can select a poll to show results, upload tokens, and view poll outcomes. Managers have the privilege to *end the poll*.
    - **Create Poll:** Managers can set the title, description, vote options, end date, and generate a code for other users to join the poll.
    - **Create Support Tickets:** Users can describe issues by creating tickets.
    - **Cast a Vote:** Users can cast votes, initiating a transaction.
- **Transaction Processing State:**
  - Transactions initiated for voting, creating polls, or minting NFTs undergo validation.
  - On success, users exit the transaction and return to the idle state.
  - On failure, the system transitions to the **Error Processing** state.
- **Error Processing State:**
  - The system notifies users about the error and allows them to exit the state, returning to idle for further interactions.
- **Creating Ticket State:**
  - Users describe issues in the system, creating support tickets for the admin.
- **Resolving Ticket State:**
  - Admins address issues by providing support messages and resolving tickets.
- **Managing Users State:**
  - Admins can change user roles to assign permissions such as *Admin*, *Manager*, or *User*.
- **Mint NFT:**
  - After successful participation in a poll, users can mint an NFT containing the participation token.
- **Log Out and Exit:**
  - Users can log out at any time, transitioning to the final state.

The state diagram captures the transitions between the states, ensuring that all actions, such as poll management, ticket creation, and transactions, are logically sequenced and validated.

3.3.3 Activity Diagrams

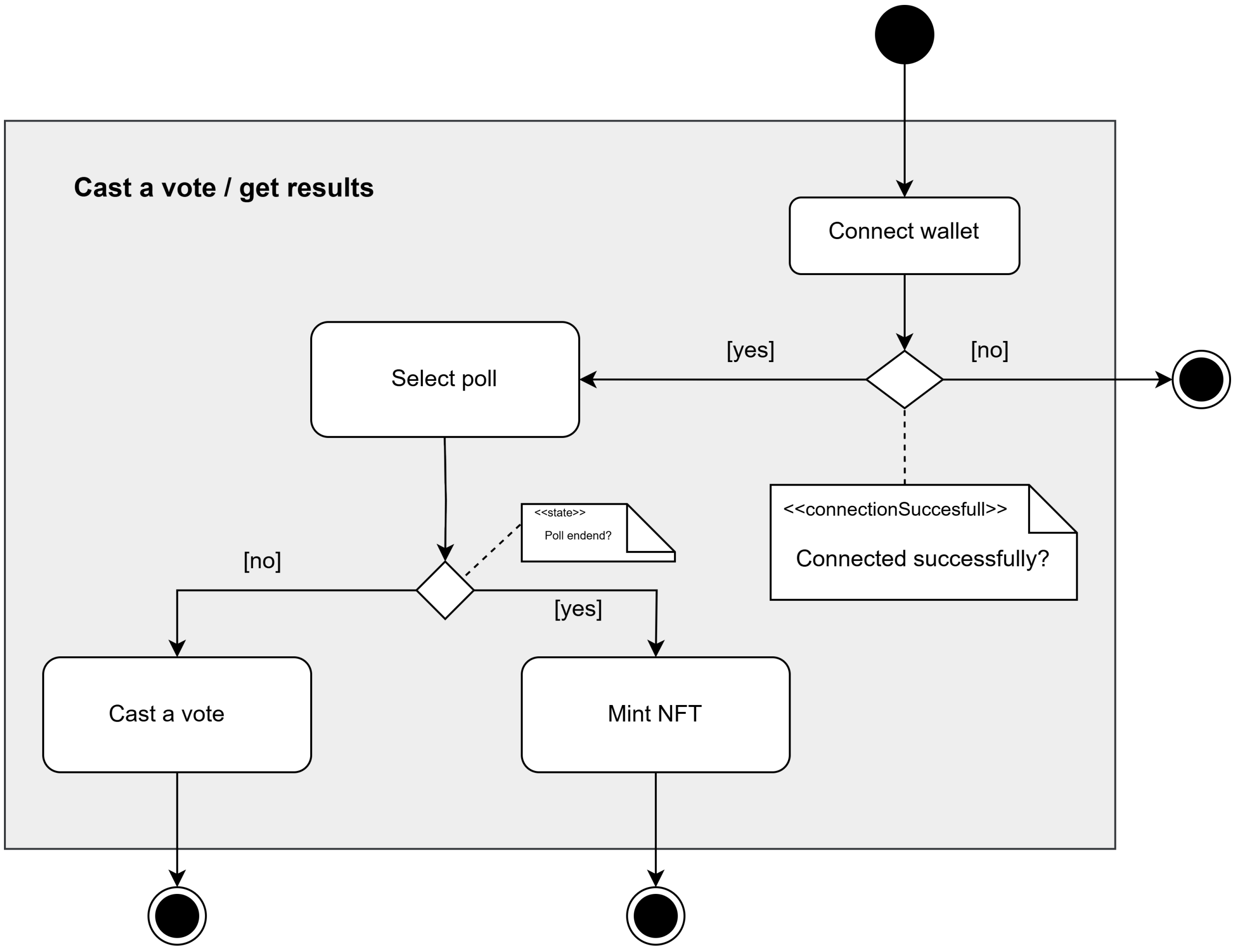

Figure 3: Activity Diagram

Cast a Vote / Get Results
The first activity diagram describes the **process for casting a vote and obtaining results** within the decentralised voting system. Below is the step-by-step explanation:

1. **Start the Process**: The process begins when the user initiates the voting flow.
2. **Connect Wallet**: The user connects their Metamask wallet.
    - If the wallet connection fails, the process terminates.
    - If the wallet connects successfully, the user proceeds to the next step.
3. **Select Poll**: The user selects a specific poll.
4. **Check Poll State**:
    - If the poll is still active (not ended), the user can **cast a vote**.
    - If the poll has ended, the user can **mint an NFT** containing the poll results.
5. **End the Process**: The process concludes once the user has cast a vote or minted an NFT.

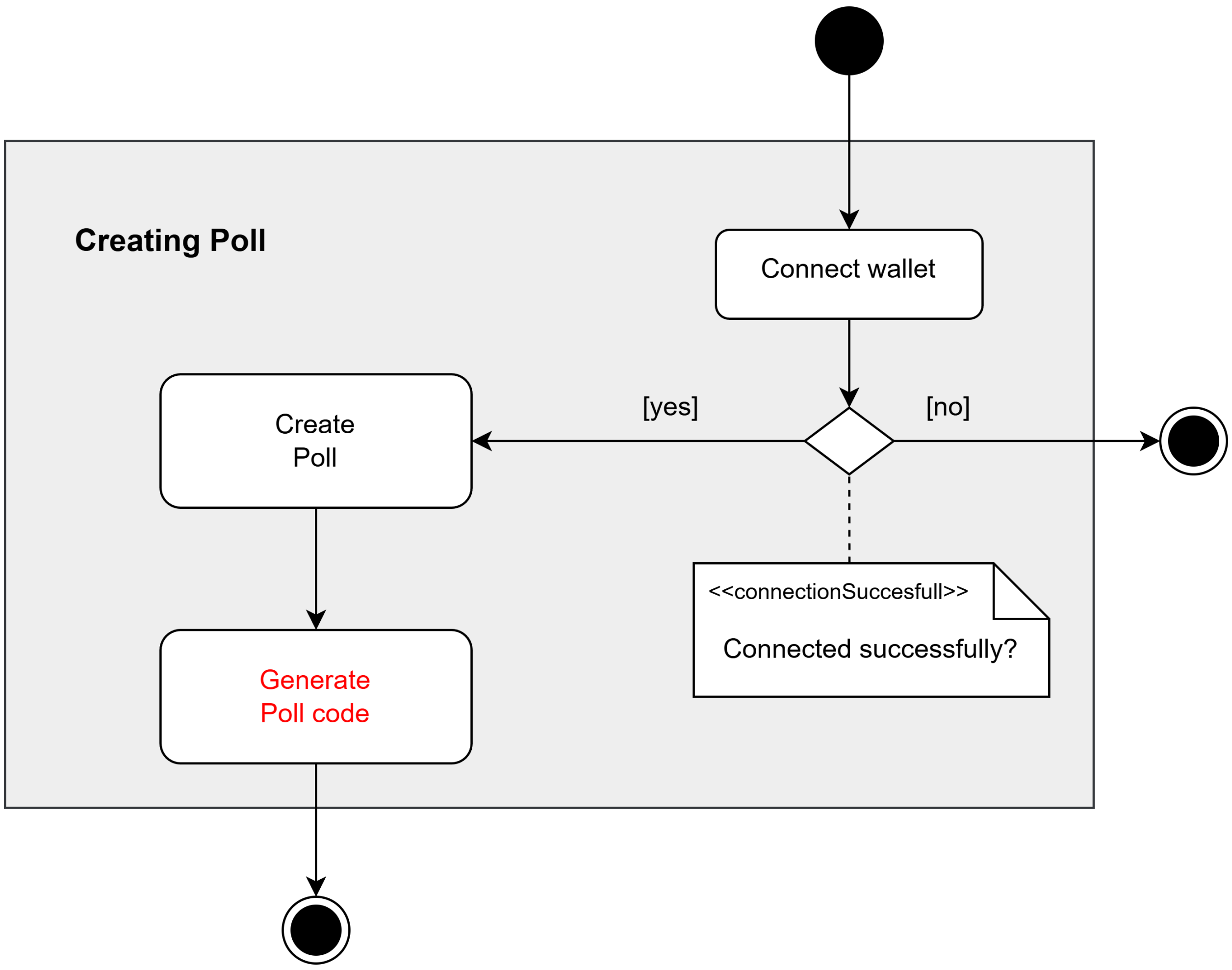

Figure 4: Activity Diagram

Creating a Poll
The second activity diagram outlines the **poll creation process**, which is restricted to users with the manager role. Here is the detailed breakdown:

1. **Start the Process**: The user initiates the poll creation flow.
2. **Connect Wallet**: The user connects their Metamask wallet.
    - If the wallet connection fails, the process ends.
    - If the wallet connects successfully, the user moves forward.
3. **Create Poll**: The manager creates a new poll.
4. **Add Participants**: The manager adds participants to the poll.
5. **End the Process**: Once participants are added, the poll creation process concludes.

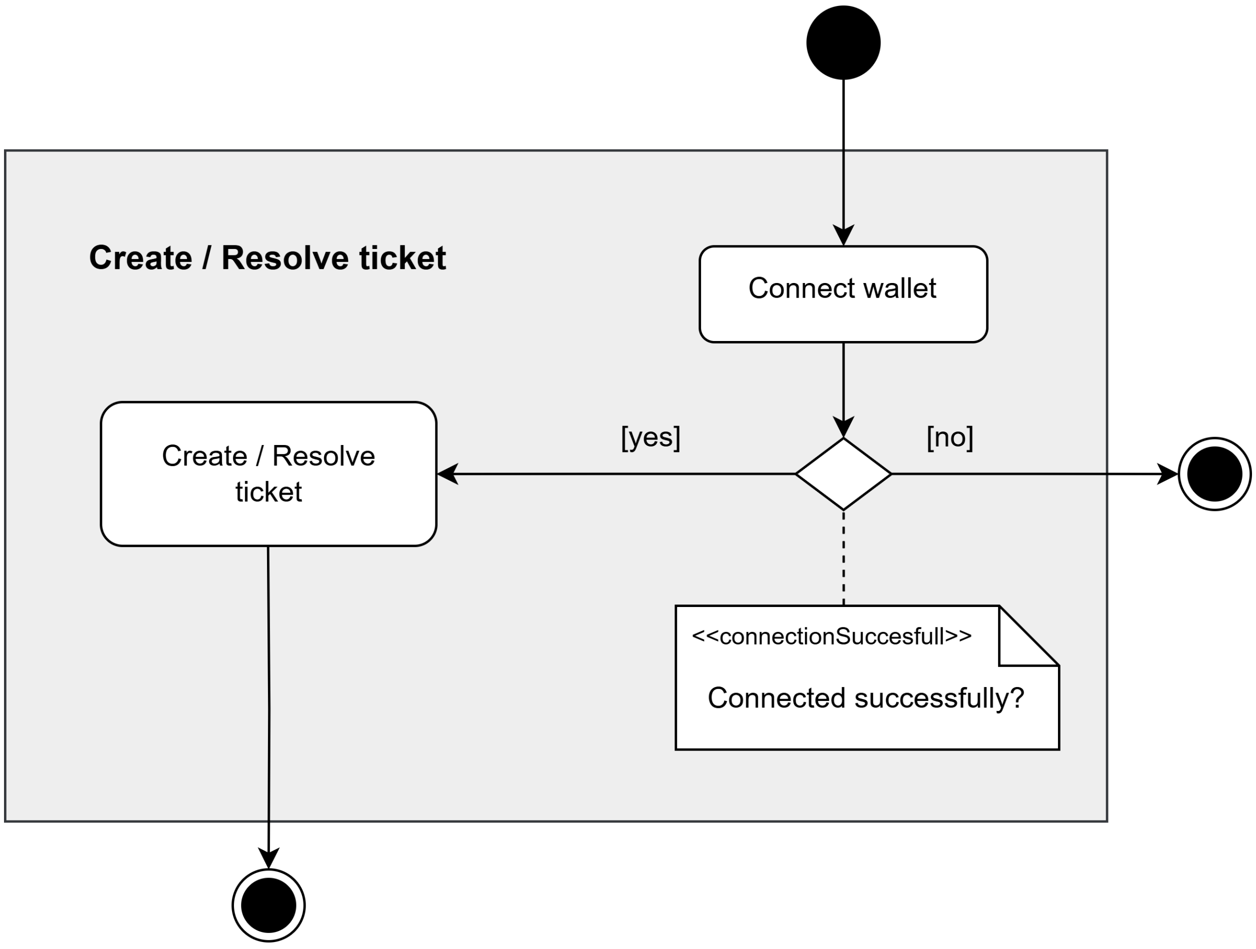

Figure 5: Activity Diagram

Create / Resolve Ticket
The third activity diagram describes the **ticket creation and resolution process** within the system. Below is the explanation:

1. **Start the Process**: The process starts when a user initiates a request to create or resolve a ticket.
2. **Connect Wallet**: The user connects their Metamask wallet.
   - If the connection fails, the process terminates.
   - If the wallet connects successfully, the user proceeds.
3. **Create / Resolve Ticket**: The user performs the desired action, either creating or resolving a ticket.
4. **End the Process**: The process concludes after the ticket has been created or resolved.

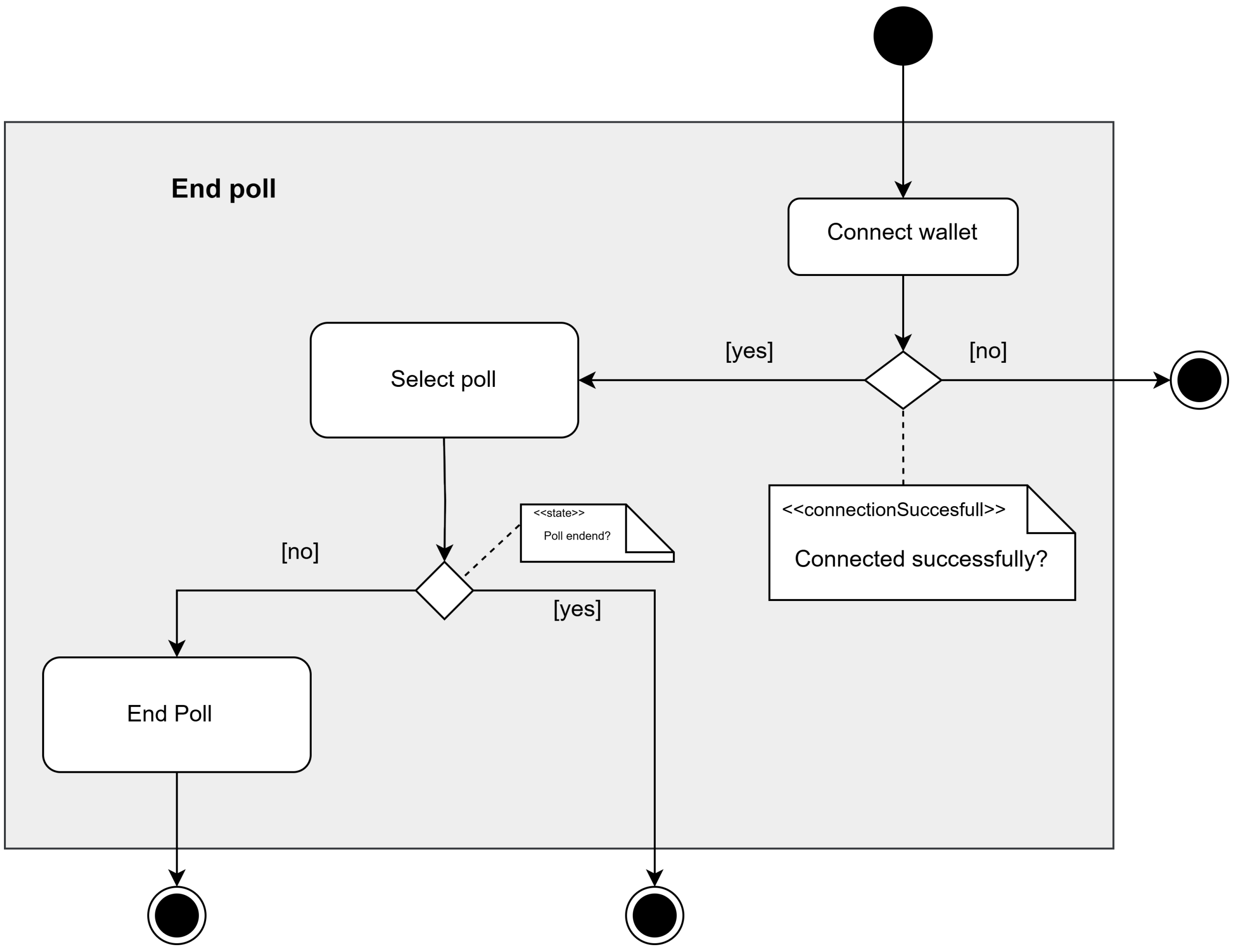

Figure 6: Activity Diagram

End Poll
This activity diagram explains the **process of ending a poll** in the system. Below is the explanation:

1. **Start the Process**: The process begins when the user initiates the request to end a poll.
2. **Connect Wallet**:
   - If the connection fails, the process terminates.
   - If the wallet connects successfully, the user continues.
3. **Select Poll**: The user selects the poll to end.
4. **Check Poll State**: The system checks if the selected poll has already ended:
   - If **yes**, the process ends.
   - If **no**, the user proceeds to end the poll.
5. **End Poll**: The poll is successfully ended. The participants are notified of this when viewing the poll.
6. **End the Process**: The process terminates after the poll has been ended.

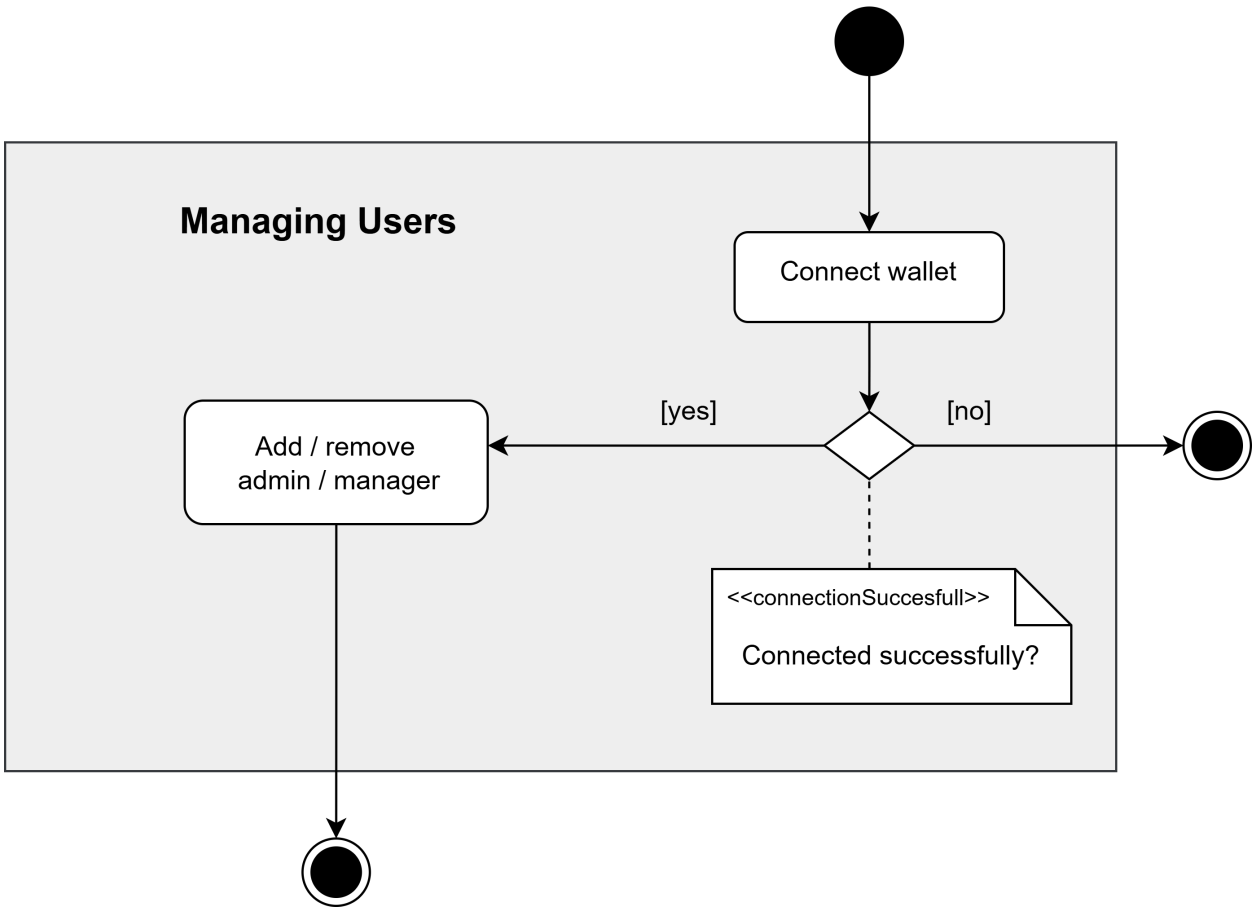

Figure 7: Activity Diagram

Managing Users
This activity diagram describes the **user management process**, including adding or removing administrators or managers. Below is the explanation:

1. **Start the Process**: The process begins when a user wants to manage roles in the system.
2. **Connect Wallet**:
   - If the connection fails, the process terminates.
   - If the wallet connects successfully, the user continues.
3. **Add/Remove Admin/Manager**: The user can perform actions to add or remove administrators or managers.
4. **End the Process**: The process concludes after the role changes have been applied.

3.3.4 Sequence Diagrams

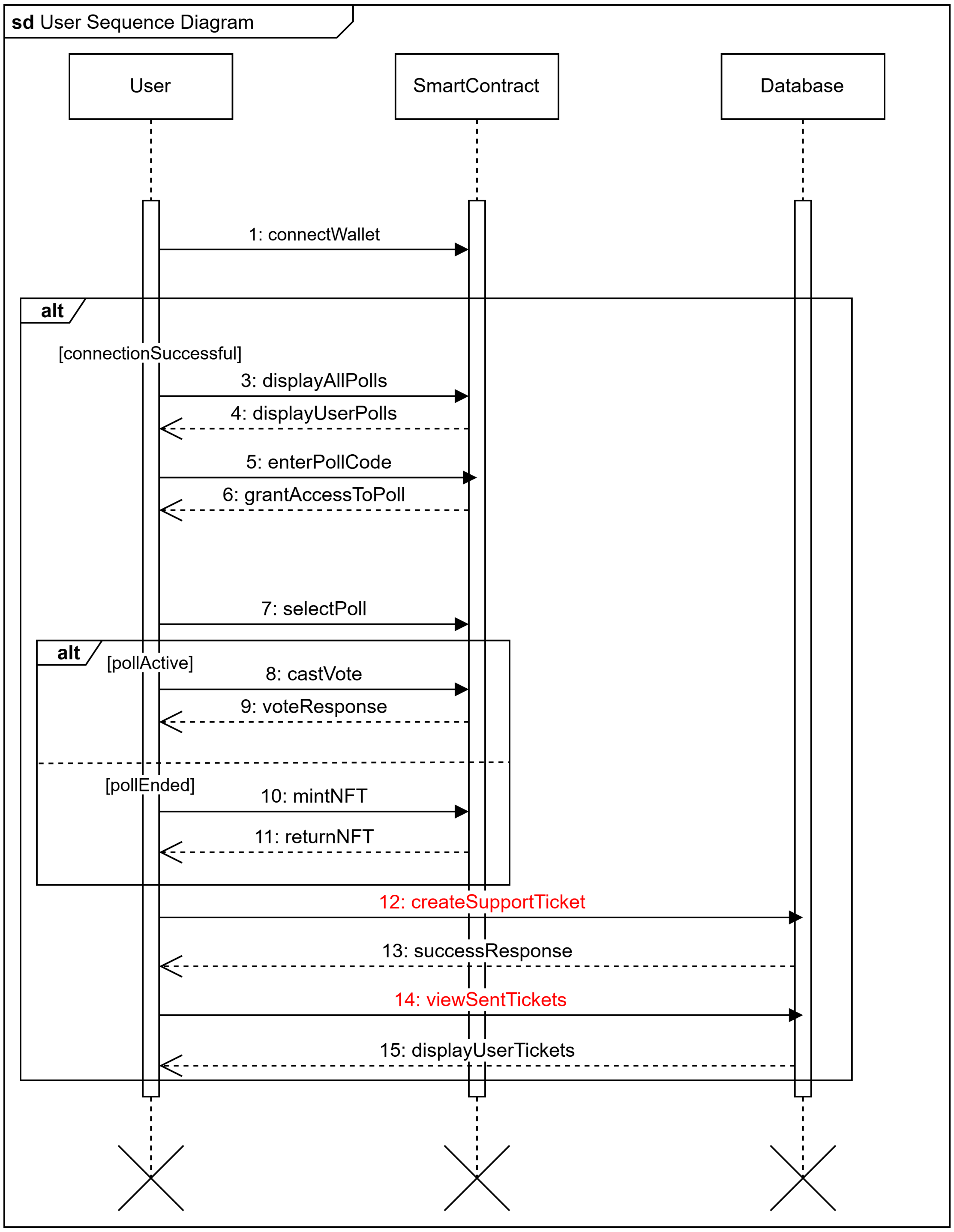

Figure 8: User Sequence Diagram

The sequence diagram showcases the interaction between the **User**, the **Smart Contract**, and the **Database** components.

The key steps in the sequence are outlined as follows:

1. **connectWallet:** The user initiates the process by connecting their MetaMask wallet to the decentralised application. This establishes a secure identity for the user.
2. **[connectionSuccessful]:** Once the connection is successfully established, the system allows further interactions.
3. **displayAllPolls:** The user requests to view all available polls.
4. **displayUserPolls:** The smart contract responds by retrieving and displaying the list of polls the user has access to.
5. **enterPollCode:** Users can input a unique poll code to gain access to a specific poll.
6. **grantAccessToPoll:** The smart contract validates the provided poll code and grants access to the poll.
7. **selectPoll:** The user selects a poll to participate in.
8. **[pollActive] castVote:** If the poll is active, the user submits their vote, which is handled by the smart contract.
9. **voteResponse:** The smart contract processes the vote and sends a confirmation response back to the user.
10. **[pollEnded] mintNFT:** After the poll ends, the user can mint an NFT representing the results of the poll.
11. **returnNFT:** The smart contract confirms the successful minting of the NFT and returns it to the user's wallet.
12. **createSupportTicket:** Users can create a support ticket for issues or inquiries related to the voting process.
13. **successResponse:** A confirmation is sent back to the user to indicate that the ticket has been successfully created.
14. **viewSentTickets:** Users can request to view previously submitted support tickets.
15. **displayUserTickets:** The system displays the list of support tickets to the user.

The sequence diagram includes two alternative (`alt`) conditions:

- **connectionSuccessful:** Ensures the user's wallet is connected before interacting with polls.
- **pollActive and pollEnded:** Differentiates the user's actions based on the poll status. If the poll is active, users can vote; if the poll has ended, users can mint NFTs representing the results.

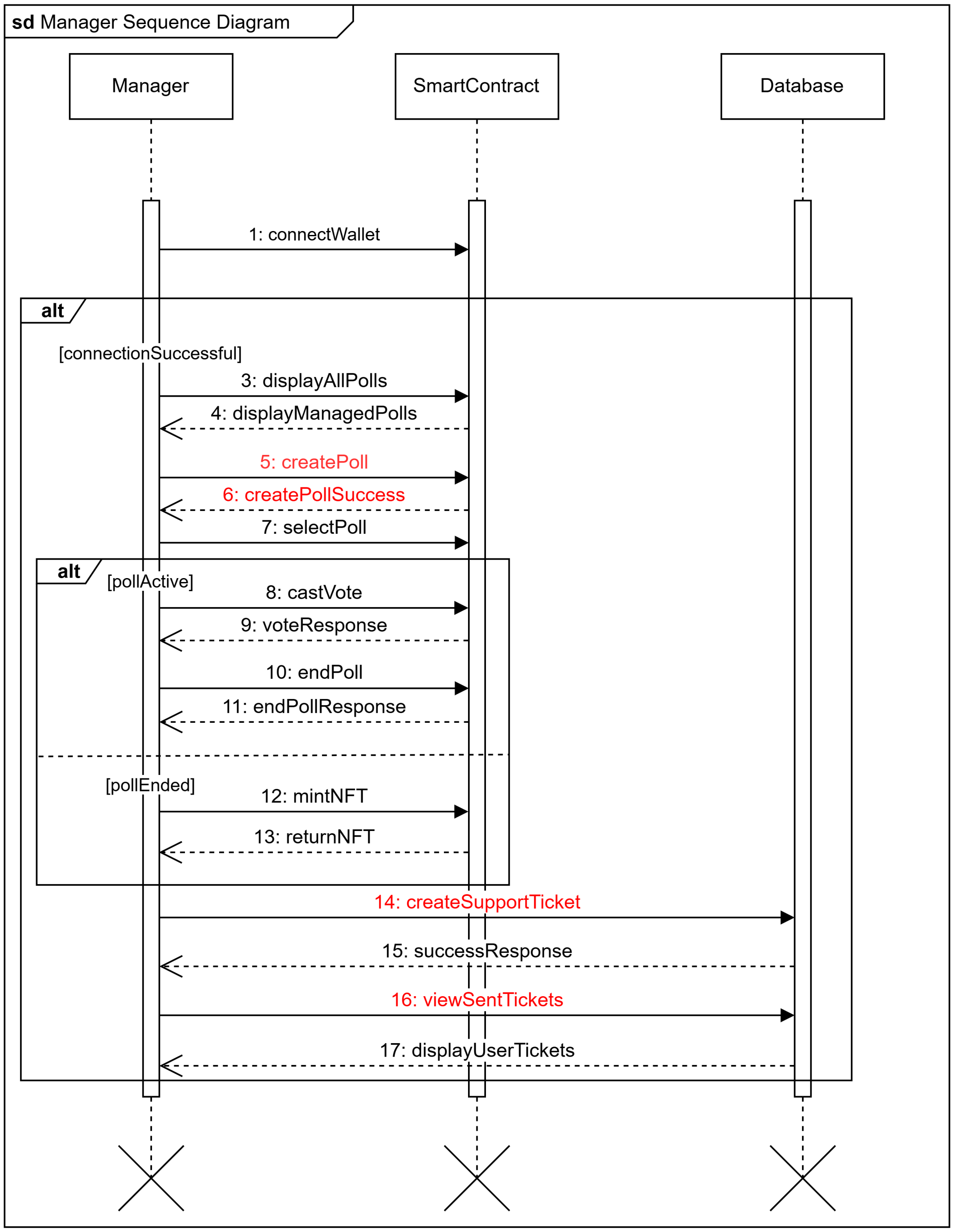

Figure 9: Manager Sequence Diagram

The sequence diagram illustrates the interaction between the **Manager**, the **Smart Contract**, and the **Database**. Managers, as privileged users, can view and manage polls, perform actions like ending polls, and participate in NFT minting after the poll concludes.

The key steps in the sequence are as follows:

1. **connectWallet:** The manager begins by connecting their MetaMask wallet to authenticate themselves in the decentralised system.
2. **[connectionSuccessful]:** Once the wallet connection is verified, the system enables further operations.
3. **displayAllPolls:** The manager requests to view all available polls in the system.
4. **displayManagedPolls:** The smart contract responds with the list of polls managed by the authenticated manager.
5. **createPoll:** The manager begins creating the poll
6. **createPollSuccess:** The smart contract processes the poll and returns success message when it is created.
7. **selectPoll:** The manager selects a specific poll to manage or interact with.
8. **[pollActive] castVote:** If the poll is still active, the manager can participate in the poll by submitting a vote.
9. **voteResponse:** The smart contract processes the vote and sends a response back to the manager.
10. **endPoll:** As part of their management privileges, the manager can end the poll, transitioning it to the *ended* state.
11. **endPollResponse:** The smart contract confirms that the poll has been successfully ended.
12. **[pollEnded] mintNFT:** After the poll concludes, the manager can mint an NFT containing the poll results.
13. **returnNFT:** The smart contract confirms the NFT creation and returns it to the manager's wallet.
14. **createSupportTicket:** The manager can submit a support ticket for issues or queries regarding the system.
15. **successResponse:** A confirmation response is sent back to the manager, indicating the successful creation of the ticket.
16. **viewSentTickets:** The manager can request to view all previously submitted support tickets.
17. **displayUserTickets:** The system displays the fetched support tickets to the manager.

The sequence diagram incorporates two alternative (`alt`) conditions:

- **connectionSuccessful:** Ensures that the manager's wallet is successfully connected before any further actions are allowed.
- **pollActive and pollEnded:** Differentiates the manager's actions based on the poll status. While the poll is active, the manager can vote or end the poll. Once the poll ends, the manager can mint an NFT containing the poll's final results.

The sequence highlights the manager's privileged role, particularly the ability to end polls and manage NFTs. All interactions, including votes and role-based actions, are securely handled by the smart contract to ensure transparency and immutability.

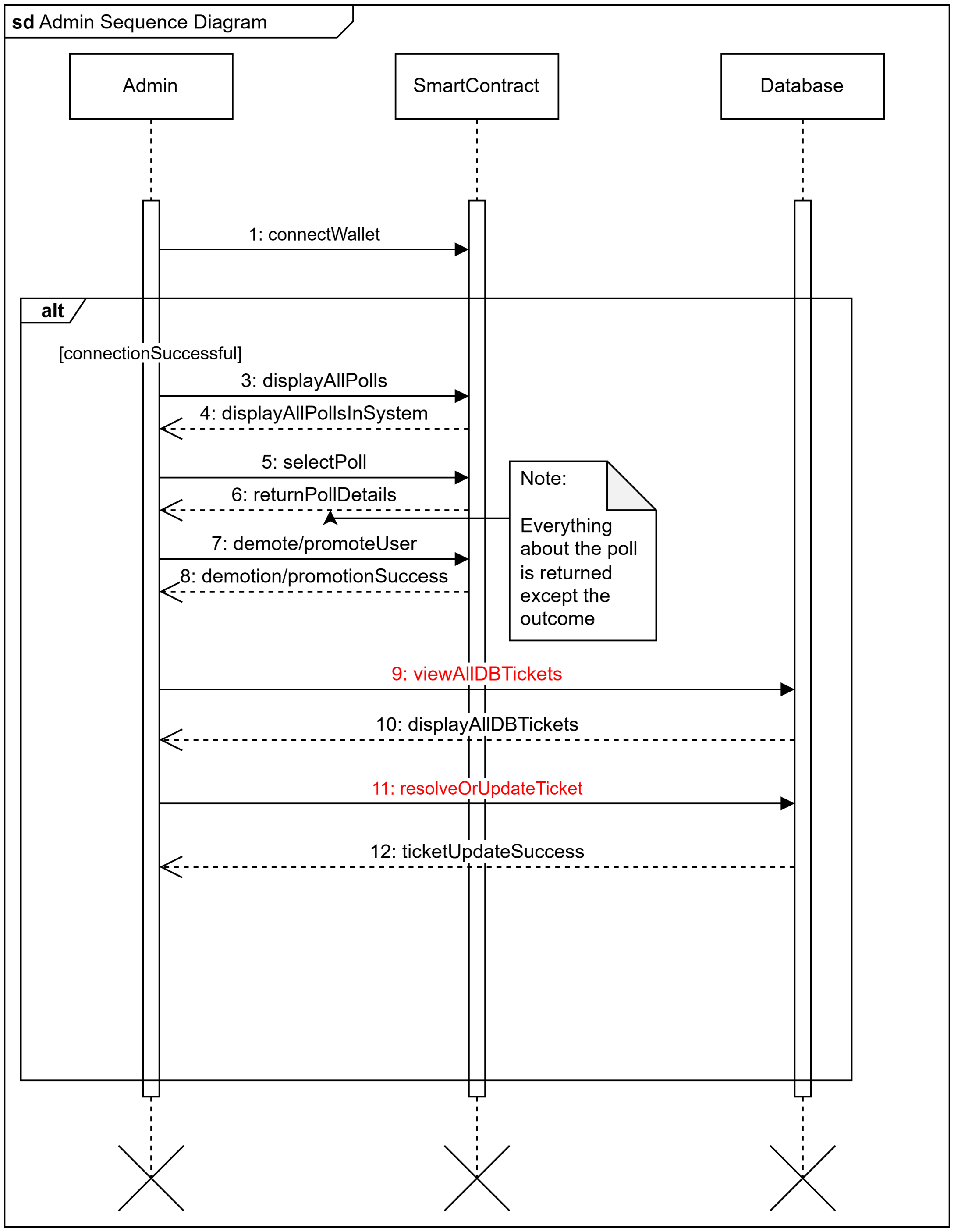

Figure 10: Admin Sequence Diagram

The sequence diagram illustrates the interaction between the **Admin**, the **Smart Contract**, and the **Database**. Admin users have privileged access to manage polls, oversee users, and handle support tickets within the system.

The key steps in the sequence are outlined as follows:

1. **connectWallet:** The admin begins by connecting their MetaMask wallet to authenticate themselves in the decentralised system.
2. **[connectionSuccessful]:** Once the connection is successfully verified, the system enables further administrative actions.
3. **displayAllPolls:** The admin requests to view all polls currently available in the system.
4. **displayAllPollsInSystem:** The smart contract responds with the details of all polls, providing visibility across the system.
5. **selectPoll:** The admin selects a specific poll to view its details.
6. **returnPollDetails:** The smart contract returns all poll-related information, excluding the outcome. *Note: A comment indicates that the poll's outcome is withheld for privacy or security reasons.*
7. **demote/promoteUser:** The admin can demote or promote a user (e.g., assign or revoke a managerial role).
8. **demotion/promotionSuccess:** The smart contract confirms the successful role update of the specified user.
9. **viewAllDBTickets:** The admin requests to view all support tickets stored in the database.
10. **displayAllDBTickets:** The retrieved tickets are displayed to the admin for review.
11. **resolveOrUpdateTicket:** The admin takes action to resolve or update a support ticket (e.g., mark as resolved, update status, or add notes).
12. **ticketUpdateSuccess:** A confirmation response is sent to the admin indicating that the support ticket has been successfully updated or resolved.

The sequence diagram includes the following alternative (`alt`) condition:

- **connectionSuccessful:** Ensures that the admin's wallet is successfully connected before any privileged administrative actions can take place.

The sequence highlights the admin's key responsibilities, such as:

- Viewing and managing all polls in the system.
- Promoting or demoting users based on roles (e.g., assigning or revoking managerial privileges).
- Viewing and resolving all support tickets stored in the database.

3.3.5 Class Diagrams

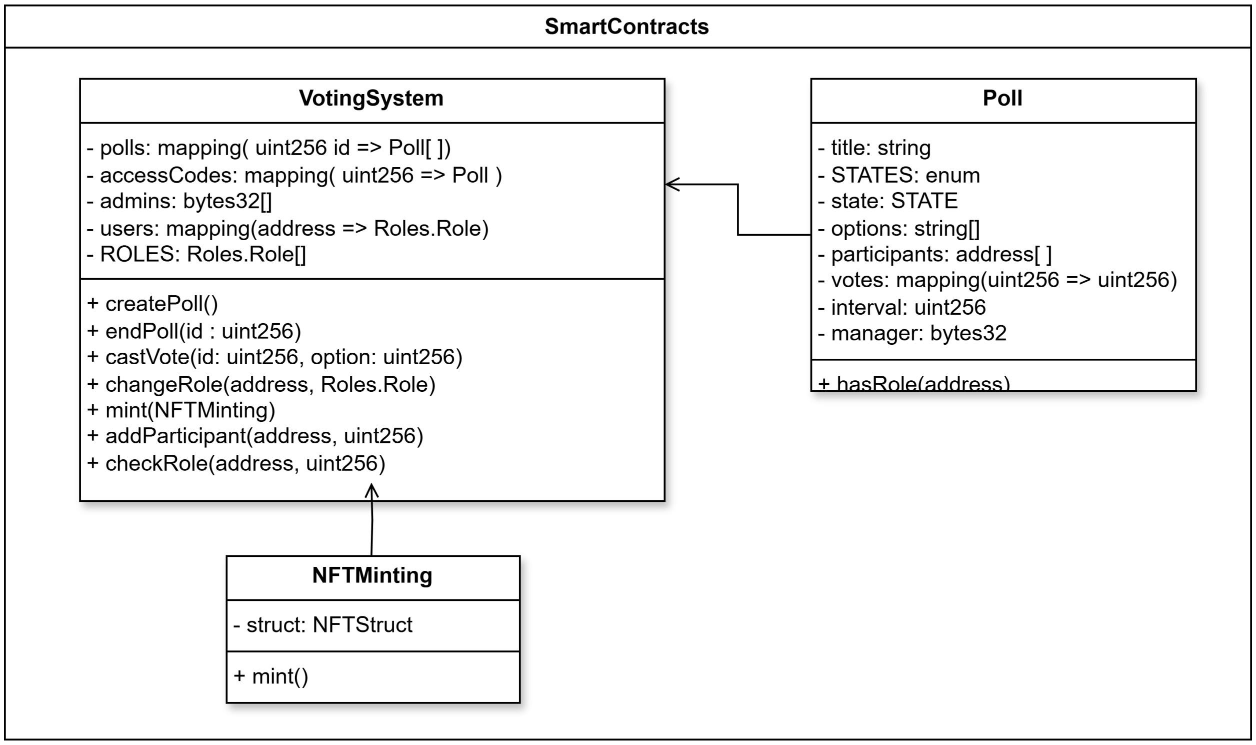
SmartContracts
VotingSystem
- polls: mapping(uint256 id => Poll[ ])
- accessCodes: mapping(uint256 => Poll)
- admins: bytes32[]
- users: mapping(address => Roles.Role)
- ROLES: Roles.Role[]
+ createPoll()
+ endPoll(id : uint256)
+ castVote(id: uint256, option: uint256)
+ changeRole(address, Roles.Role)
+ mint(NFTMinting)
+ addParticipant(address, uint256)
+ checkRole(address, uint256)
Poll
- title: string
- STATES: enum
- state: STATE
- options: string[]
- participants: address[ ]
- votes: mapping(uint256 => uint256)
- interval: uint256
- manager: bytes32
+ hasRole(address)
NFTMinting
- struct: NFTStruct
+ mint()

Figure 11: Class Diagram

The system is composed of three primary components, represented in the class diagram:

- **VotingSystem**: The main smart contract that orchestrates the voting process, manages roles, and ensures system integrity. It includes:
  - *State Variables*:
    - `polls`: A mapping storing polls by their unique ID.
    - `accessCodes`: A mapping for associating unique access codes with polls.
    - `admins`: A list of admin users identified by their `bytes32` values.
    - `users`: A mapping to manage user roles, where each address is assigned a `Roles.Role`.
    - `ROLES`: An array defining the various roles in the system.
  - *Functions*:
    - `createPoll()`: Allows managers to create a new poll.
    - `endPoll(id: uint256)`: Marks a poll as concluded.
    - `castVote(id: uint256, option: uint256)`: Enables users to cast their votes for a specific option in a poll.
    - `changeRole(address, Roles.Role)`: Updates the role of a user.
    - `mint(NFTMinting)`: Invokes the minting process for NFTs based on poll results.
    - `addParticipant(address, uint256)`: Adds a participant to a specific poll.
    - `checkRole(address, uint256)`: Verifies the role of a given user.

- **Poll**: Represents an individual poll within the system. Each poll contains:
  - *Attributes*:
    - `title`: The title of the poll.
    - `STATE`: An enum defining the various states of polls in the system.
    - `state`: The state of the poll.
    - `options`: A list of available voting options.
    - `participants`: An array of addresses of users participating in the poll.
    - `votes`: A mapping that records votes, where keys represent options and values represent the count of votes.
    - `interval`: A time period specifying the poll duration.
    - `manager`: The address of the manager overseeing the poll.
  - *Functions*:
    - `getResults()`: Returns the results of the poll.
    - `hasRole(address)`: Checks if a given address holds a specific role.
- **NFTMinting**: Handles the minting of NFTs for poll participants after a poll concludes. It includes:
  - *Attributes*:
    - `struct`: Defines the NFT data structure, which will include poll results and other metadata.
  - *Functions*:
    - `mint()`: Mints the NFT for eligible participants based on the poll data.

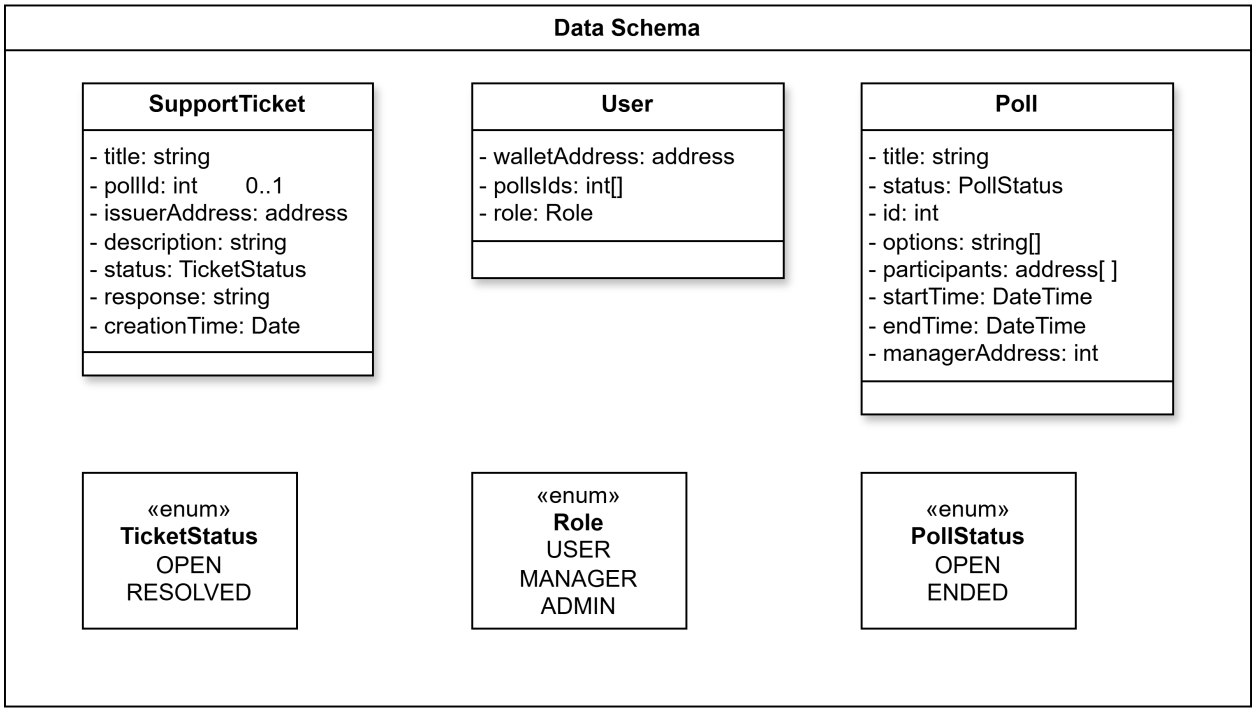

Figure 12: Class Diagram

The data schema for the decentralised voting system outlines the key entities and enumerations used to manage users, polls, and support tickets. The following components are presented in the schema:

- **SupportTicket**: Represents a ticket system that allows users to submit support requests or issues. It includes:
  - *Attributes*:
    - `title`: The title of the support ticket.
    - `pollId`: An optional field (0..1) that associates the ticket with a specific poll.
    - `issuerAddress`: The wallet address of the user who submitted the ticket.
    - `description`: A detailed description of the issue.
    - `status`: The current state of the ticket, using the `TicketStatus` enumeration.
    - `response`: A response or resolution provided for the ticket.
    - `creationTime`: The date and time when the ticket was created.
- **User**: Represents a user participating in the system. Each user has:
  - *Attributes*:
    - `walletAddress`: The wallet address of the user.
    - `pollsIds`: An array of IDs representing the polls the user is associated with.
    - `role`: The role of the user, defined by the `Role` enumeration.
- **Poll**: Represents an active or completed poll in the system. It includes:
  - *Attributes*:
    - `title`: The title of the poll.
    - `status`: The current status of the poll, as defined by the `PollStatus` enumeration.

    * `id`: A unique identifier for the poll.
    * `options`: A list of available voting options.
    * `participants`: An array of wallet addresses representing the participants in the poll.
    * `startTime`: The starting date and time of the poll.
    * `endTime`: The ending date and time of the poll.
    * `managerAddress`: The wallet address of the poll manager.

- **Enumerations**:
  - `TicketStatus`: Defines the status of a support ticket. Possible values are:
    * **OPEN**: The ticket is currently open and unresolved.
    * **RESOLVED**: The ticket has been resolved.
  - `Role`: Specifies the role of a user in the system. Roles include:
    * **USER**: A regular participant in the system.
    * **MANAGER**: A user who can manage polls.
    * **ADMIN**: A system administrator with elevated privileges.
  - `PollStatus`: Defines the status of a poll. Possible values are:
    * **OPEN**: The poll is currently active and accepting votes.
    * **ENDED**: The poll has concluded, and voting is closed.

This schema provides a clear structure for managing system data, including user roles, poll lifecycle management, and a support ticket system. It ensures data integrity and role-based access control, facilitating smooth operation of the decentralised voting system.

4 Glossary

Blockchain: A decentralised and distributed ledger technology that records transactions across multiple nodes in a secure, transparent, and immutable manner. In the context of this project, the blockchain ensures the integrity and transparency of the voting process.

Ethereum Mainnet: The main public network of the Ethereum blockchain, where real transactions occur, and where the smart contracts for this system will be deployed. It is distinguished from Ethereum testnets used for development purposes.

Transaction: A cryptographically signed operation submitted to the blockchain. Transactions can include sending Ether, deploying smart contracts, or invoking functions in a smart contract, such as voting in a poll or creating a poll.

Poll: A voting instance created by a user with the manager role. A poll includes a unique code (Vote Code) for identification, a set of possible options to vote on, and metadata such as the creator, participants, and current status.

Vote Code: A unique code generated upon the creation of a poll. This code is used by participants to join and interact with the poll. It ensures that only users with the correct code can participate in the voting process.

Vote Pool: A list of participants who are eligible or have already participated in a poll. The vote pool is dynamically updated as users join and cast their votes.

Support Ticket: A mechanism for users to report issues or request assistance. In this system, support tickets might be used to handle cases such as errors in poll creation or difficulties in accessing a poll.

NFT (Non-Fungible Token): A unique digital asset stored on the blockchain that represents ownership or proof of participation. NFTs in this system will be minted to commemorate participation in a poll and will include poll results as metadata.

NFT Minting: The process of creating a new NFT on the blockchain. After a poll concludes, participants can mint NFTs that encapsulate the results of the poll and their involvement as immutable records.

NFT Uploading: The process of uploading NFT from the user wallet, in order to retrieve the results and data about specific poll.

Smart Contract: A self-executing program deployed on the blockchain, with code that defines rules and procedures for specific operations. In this project, smart contracts govern voting, role-based permissions, and the minting of NFTs.

MetaMask Wallet: A browser-based cryptocurrency wallet used to interact with the Ethereum blockchain. Users log in to the system using their MetaMask accounts, which serve as their identity and provide access to blockchain operations.

Poll State/Status: The current condition or phase of a poll. Examples include "OPEN" (accepting votes), "ENDED" (voting concluded, results finalized and NFTs available for minting). The poll state is managed by the smart contract.