Atelier 6 : Mini Social Media

Département Génie Informatique, Faculté des Sciences et Techniques de Tanger,
Université Abdelmalek Essaâdi, Tanger, Maroc
Chibani Fahd

1 Introduction

This report outlines the design and implementation of a mini social media platform built using a Solidity smart contract.The contract stores all post data on the blockchain, providing an immutable and transparent record of user interactions. The system consists of the following key components :

- A smart contract deployed on the Ethereum blockchain, which handles post creation, deletion, editing, and liking/disliking.
- A frontend application built with HTML, CSS, and JavaScript that communicates with the Ethereum blockchain using Web3.js.
- MetaMask for managing user accounts and transactions.

2 Solidity smart contract

2.1 Contract Structure

The core of the mini social media platform is the `SocialMedia` smart contract, which is written in Solidity version 0.8.0. The contract consists of the following key components :

2.1.1 Post Struct

The `Post` struct represents a single post on the platform, containing the following fields :

- `id` : A unique identifier for the post
- `author` : The address of the user who created the post
- `content` : The text content of the post
- `likes` : The number of likes the post has received
- `dislikes` : The number of dislikes the post has received
- `createdAt` : The timestamp of when the post was created
- `modifiedAt` : The timestamp of the last modification to the post

2.1.2 State Variables

The contract maintains the following state variables :

- `posts` : An array to store all posts on the platform
- `userPosts` : A mapping that associates each user's address with a list of post IDs they have created
- `postLiked` : A mapping that tracks whether a specific user has liked a specific post
- `postDisliked` : A mapping that tracks whether a specific user has disliked a specific post
- `postCount` : A counter that keeps track of the total number of posts created

2.1.3 Events

The contract defines the following events to log important actions :
— `PostCreated` : Emitted when a new post is created
— `PostUpdated` : Emitted when a post is edited
— `PostDeleted` : Emitted when a post is deleted
— `PostLiked` : Emitted when a post is liked
— `PostDisliked` : Emitted when a post is disliked

2.2 Key Functions

The SocialMedia contract provides the following functions :

2.2.1 `createPost(string memory content)`

This function allows users to create a new post with the given content. It adds the new post to the `posts` array, maps the post to the user's address in `userPosts`, and emits a `PostCreated` event.

```
function createPost(string memory content) public {
    Post memory newPost = Post({
        id: postCount,
        author: msg.sender,
        content: content,
        likes: 0,
        dislikes: 0,
        createdAt: block.timestamp,
        modifiedAt: 0
    });
    posts.push(newPost);
    userPosts[msg.sender].push(postCount);
    emit PostCreated(postCount, msg.sender, content, block.timestamp);
    postCount++;
}
```

2.2.2 `getPost(uint256 postId)`

This function retrieves a specific post by its ID, ensuring that the post exists before returning it.

```
function getPost(uint256 postId) public view returns (Post memory) {
    require(postId < postCount, "Post does not exist");
    return posts[postId];
}
```

2.2.3 `getAllPosts()`

This function returns the entire array of posts stored in the contract.

```
function getAllPosts() public view returns (Post[] memory) {
    return posts;
}
```

2.2.4 `editPost(uint256 postId, string memory newContent)`

This function allows the author of a post to update its content, provided that the post exists and the caller is the author. It updates the post's content and `modifiedAt` field, and emits a `PostUpdated` event.

```
function editPost(uint256 postId, string memory newContent) public {
    require(postId < postCount, "Post does not exist");
    Post storage post = posts[postId];
    require(msg.sender == post.author, "Only the author can edit this post");

    post.content = newContent;
    post.modifiedAt = block.timestamp;
    emit PostUpdated(postId, newContent, block.timestamp);
}
```

2.2.5 deletePost(uint256 postId)

This function allows the author of a post to delete it, provided that the post exists and the caller is the author. It removes the post from the `posts` array and emits a `PostDeleted` event.

```
function deletePost(uint256 postId) public {
    require(postId < postCount, "Post does not exist");
    Post storage post = posts[postId];
    require(msg.sender == post.author, "Only the author can delete this post");

    delete posts[postId];
    emit PostDeleted(postId, msg.sender);
}
```

2.2.6 likePost(uint256 postId)

This function allows a user to like a post, provided that the post exists and the user has not already liked the post. It increments the post's `likes` count and records the user's like in the `postLiked` mapping, then emits a `PostLiked` event.

```
function likePost(uint256 postId) public {
    require(postId < postCount, "Post does not exist");
    require(!postLiked[postId][msg.sender], "You have already liked this post");

    Post storage post = posts[postId];
    post.likes++;
    postLiked[postId][msg.sender] = true;
    emit PostLiked(postId, post.likes);
}
```

2.2.7 dislikePost(uint256 postId)

This function allows a user to dislike a post, provided that the post exists and the user has not already disliked the post. It increments the post's `dislikes` count and records the user's dislike in the `postDisliked` mapping, then emits a `PostDisliked` event.

```
function dislikePost(uint256 postId) public {
    require(postId < postCount, "Post does not exist");
    require(!postDisliked[postId][msg.sender], "You have already disliked this
        post");

    Post storage post = posts[postId];
    post.dislikes++;
    postDisliked[postId][msg.sender] = true;
    emit PostDisliked(postId, post.dislikes);
}
```

2.3 Workflow Diagram

The following diagram illustrates the workflow of the mini social media platform :

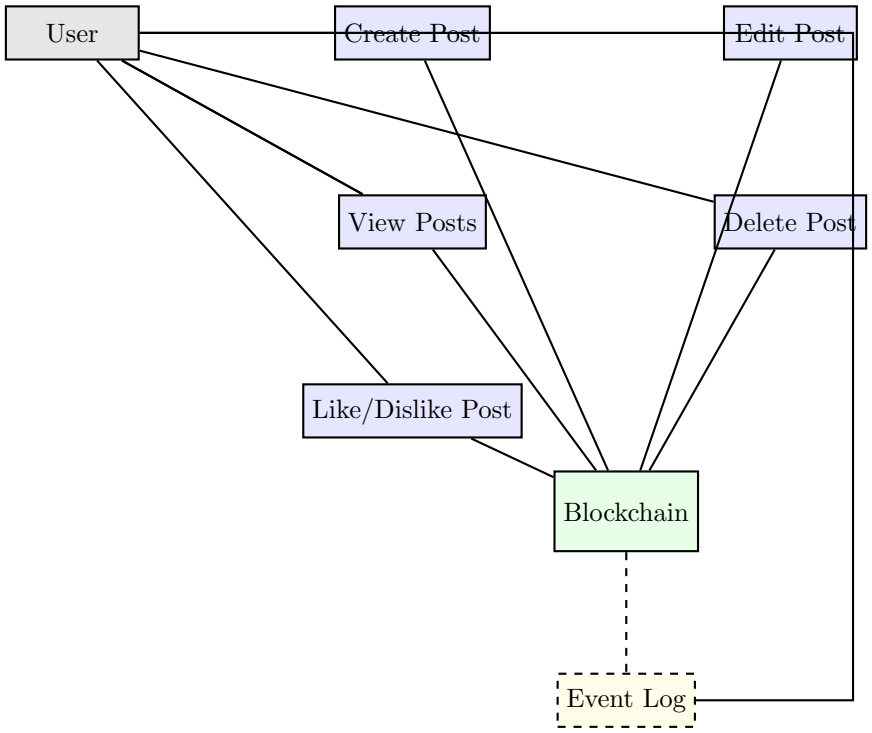


FIGURE 1 – Enhanced Workflow for Mini Social Media Platform

The workflow diagram illustrates the interactions between the user and the SocialMedia smart contract deployed on the blockchain. The key steps are :
1. User creates a new post, which is then stored on the blockchain.
2. User can view the posts, which are retrieved from the blockchain.
3. User can edit their own posts, which updates the corresponding data on the blockchain.
4. User can delete their own posts, which removes the post data from the blockchain.
5. User can like or dislike posts, which updates the like/dislike counts on the blockchain.

All user actions are directly reflected in the state of the SocialMedia contract on the blockchain, ensuring a transparent and immutable record of the platform's activity.

2.4 Conclusion

The mini social media platform built using the SocialMedia smart contract provides a decentralized solution for creating, managing, and interacting with social media posts. The contract's design ensures that post data is stored permanently and transparently on the blockchain, while also allowing users to edit, delete, and react to posts in a secure and controlled manner.

3 Fontend application

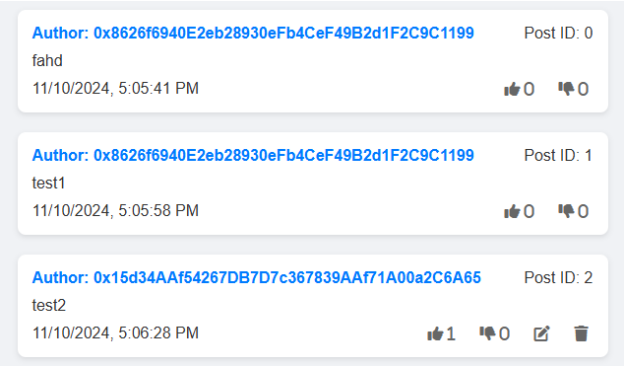
The purpose of this project is to create a decentralized social media platform where users can interact with posts in a blockchain-powered environment. The platform leverages Web3.js for interacting with Ethereum smart contracts and MetaMask for account management.

— Viewing all posts created by any user.

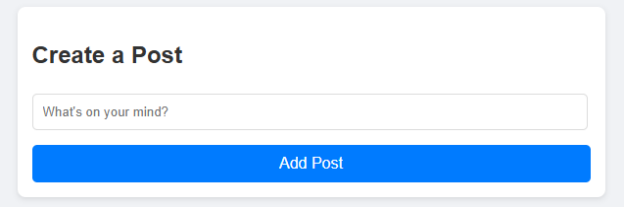
3.1 Frontend Implementation

The frontend is designed with simplicity and responsiveness in mind. It provides users with the following functionalities :

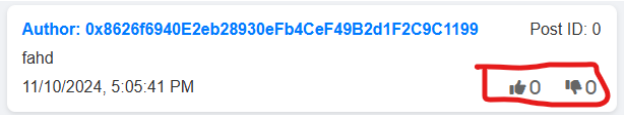
— Viewing all posts created by any user.



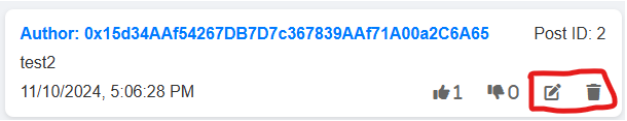
— Creating new posts with content input by the user.



— Liking and disliking posts.

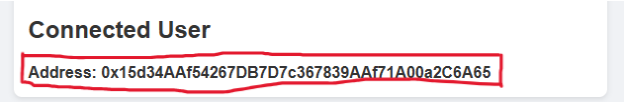


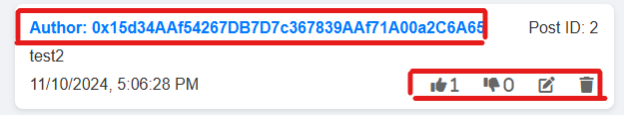
— Editing and deleting posts (only by the author of the post).



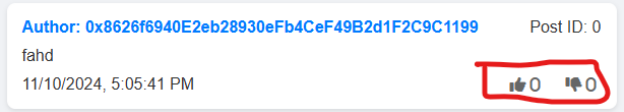
3.2 Smart Contract Interaction

The smart contract allows users to interact with posts in the following ways :
— **createPost :** Users can create a new post by providing content.
— **editPost/deletePost :** Users can edit and delete their post's content.





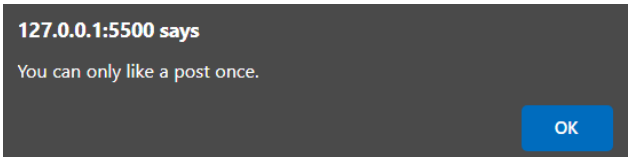
— **likePost/dislikePost :** Users can like or dislike posts.



The contract ABI and address are defined in the frontend JavaScript code, enabling Web3.js to communicate with the contract.

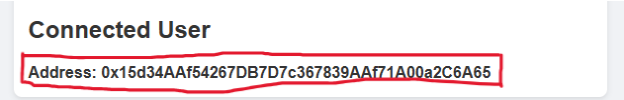
3.3 Error Handling

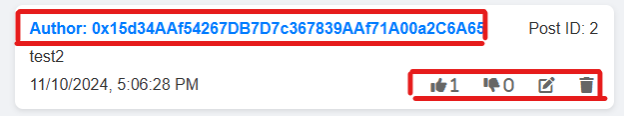
Error handling is crucial to ensure the DApp functions smoothly even in adverse conditions. The following error handling mechanisms have been implemented :
— **MetaMask Installation Check :** The application checks whether MetaMask is installed before attempting to interact with the Ethereum network.
— **Account Switching :** The DApp listens for account changes in MetaMask and updates the displayed address accordingly.
— **Post Creation Error :** If an error occurs during post creation (e.g., network issues), the user is notified with a failure message.
— **Like/Dislike Post Error :** If a user tries to like or dislike a post more than once, an alert is displayed to inform them of the restriction.

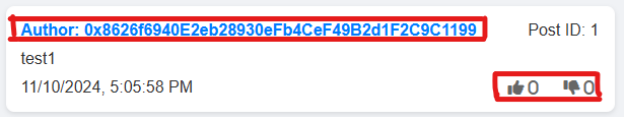


— **Edit and Delete Post Error :** Users can only edit or delete their own posts. If they try to edit or delete another user's post, an alert will notify them of the issue.

Users can't edit or delete others posts like we see in figures below







3.4 Code Example : Error Handling in Post Creation

Here is an example of the error handling implemented in the `createPost` function :

```
async function createPost() {
  const content = document.getElementById("newPostContent").value;
  try {
    await contract.methods.createPost(content).send({ from: currentAccount });
    document.getElementById("addPostStatus").innerText = "Post added successfully!";
    document.getElementById("addPostStatus").style.display = "block";
    loadAllPosts();
  } catch (error) {
    console.error(error);
    alert("Failed to add post.");
  }
}
```

Result :

4 Conclusion

The Social Media DApp demonstrates how blockchain technology can be applied to social media platforms. The integration of smart contracts with Web3.js and MetaMask allows for secure and decentralized interaction, providing users with control over their data and content.