

Spotlight

A web3 Reddit

The Team	3
Goal	3
The Pitch	3
Stack Overview	3
Currently Supported Features	4
Registration	4
Profile Management	4
ERC-20 token - Spotlight Reputation (RPT)	5
Post Creation	5
Post Verification	5
Editing Posts	5
Deleting Posts	6
Paywalling Posts	6
Comments	6
Upvotes / Downvotes	6
Tokenomics	7
Earnings and Burnings	7
Token Purpose	8
Decay Mechanism	8
How to prevent “gaming the system”	9
Contract Details	9
User Profiles	9
Posts	10
Frontend Details	14
Navigation Route	14
Login Page	15
Feed Page	16
Post Detail Page	17
Home Page	19
Responsive Layout	20
Quality Assurance	20
Future Features	21
Post Media	21
Off-chain Storage	21
Communities, DAOs, and Moderation	22

Encryption/Decryption of Paywall Posts	22
Paywalls and Content Monetization	22
Search and Sorting	22
Expanding Comments	23
Deeper Robustness for RPT	23
Decentralized Hosting	23
Lessons Learned	23
Web3 Tooling, their responsiveness and impact on UX	23
More “gaming-the-system” concerns	24
General Solidity Learnings	24
Appendix	24
Spotlight Codebase	24
Solidity Contracts	25
Unit Tests	25
NextJS App	25
Spotlight-Bot	25
References	25
Paywall Post Sequence Diagrams	26

The Team

Team member	Role
Garfield (Chengke) Deng	Frontend Lead
Juan Gutierrez	System Architect
Yixuan Li	Backend Lead
Robin (Hung-Ching) Liu	QA
Louis Zhao	Team Manager

Goal

This document aims to provide an in depth walkthrough of the feature sets, design decisions and implementation details of our project. The earlier stages of the report provide high level overviews but as it progresses, the aim is to provide more intricate and precise implementation specifications of the aforementioned feature set.

The Pitch

For writers, artists, and creators who share their original work on social media platforms like Reddit, it can be challenging to gain proper credit, earn revenue, and achieve lasting recognition for their contributions.

Spotlight is a decentralized, Reddit-style social media platform that rewards creators when their content is deemed helpful or entertaining by the community. Users earn reputation, represented through an ERC-20 token, primarily through community upvotes on their posts, but any engagement with the platform will grant users a tiny bit of reputation. Reputation is dynamic and gradually diminishes over time, ensuring that all members of the community have an equal opportunity to shine in the spotlight.

Stack Overview

The Spotlight DApp is currently supported by 3 main pillars:

- A Solidity smart contract - which makes it compatible with any EVM blockchain

- This acts as our backend server and main storage mechanism for the majority of user generated content
- A NextJS application which provides the frontend UI/UX, integrates with a user's wallet, and interfaces with the Spotlight smart contract.
 - Wallet and contract interactions facilitated by wagmi[1] through viem[2]
- web3.storage[3] provides access to IPFS decentralized storage with help from UCAN[4]

The DApp was constructed using the framework scaffold-eth[5] which allowed us a quick setup by providing an environment that tightly bound the contract, the frontend that user's use to interact with the contract, and both their deployments. Local development was driven by scaffold-eth in combination with a local testnet driven by foundry's anvil tool[6].

Currently Supported Features

Registration

Like with any social media application, it begins with the registration of your unique profile. Your registration is inherently bound to your wallet address. Therefore, in order to be able to interact with Spotlight, let alone register, you must have an EVM compatible wallet, such as MetaMask or Phantom. Registration begins with you choosing any 32-byte username (NOTE: yes, emoji's are supported 😊). Username restrictions are

- You cannot have any empty username
- It cannot exceed 32 bytes in length
- Another profile cannot have already taken the username

In order to enforce username uniqueness, we keep a mapping of all active usernames, normalized to lowercase and hashed. This means we consider the username's "AAA", "aaa", "aAa" and all other variations to be equivalent. While this limits the username space, it can prevent confusion and further ensure a user's sense of uniqueness.

Profile Management

Your username and wallet are not the only things we use to help you cultivate your unique persona in Spotlight. There are currently three operations enabled for managing your profile:

- Update Username
- Change Profile Photo
- Delete Profile

With respect to the update username feature, the same constraints are applied during Spotlight registration. The "Delete Profile" operation also is fairly straightforward - this will remove your username from the tracked list and remove all of your posts from active storage. However, it is worth noting that, given the immutability of the blockchain, the user's content is never truly "deleted" - the dApp will simply stop rendering that particular address's content.

The change profile photo feature is more interesting, leveraging web3.storage's IPFS and UCAN protocol to store and serve profile photos. More on this later.

ERC-20 token - Spotlight Reputation (RPT)



Spotlight comes with an ERC-20 token that is used to measure reputation within the Spotlight ecosystem. As users post and engage with each other, they are issued varying amounts of RPT, depending on the type of engagement. We'll go into much greater detail in the [Tokenomics](#) section of this document, but it needs to be mentioned early on, as RPT may be referred to in the following sections.

Post Creation

The concept of "ownership" is one of the core concepts in web3. The way in which posts are created in Spotlight takes this to heart. When a user creates a post, the user is asked to sign the post with their wallet first. While many claim this is unnecessary, given that the post will inherently be bound to the transaction when the post is stored on-chain, this was a forward thinking design decision.

To ensure signature uniqueness, the data of the signature is the aggregation of the title of the post, its content, and a random number. This uniqueness is extremely important. Given that, on-chain, we're not dealing with a traditional data store that has auto-incrementing indices, we leverage the initial signature as the index of a post. This index is more-often than not used as a key in Solidity contract mappings to ensure $O(1)$ access to post lookups. The signature generation follows the ERC-191[7] standard of a signed message hash. This is available to EVM compatible wallets and also Solidity contracts, which allows us to verify that the post did indeed come from the person claiming to be the creator at creation time. In other words, we can reject storing of posts should someone attempt to impersonate another user.

Post Verification

What good is a signature if you don't verify it? As mentioned, all posts are stored with the signature of their content. That signature is specifically generated by the user's wallet, which means that if you have the address of the creator, then you can verify the signature. When rendering posts, Spotlight always verifies the content against the signature. A badge () is present indicating that verification has passed. All that said, there's no need to trust the badge rendered. All posts are available for reading and can be verified independently by users if they wish. If a posts fails verification, user's will see the following icon instead  .

Editing Posts

We allow for user's to make modifications to their uploaded content if they deem necessary. Updates to posts will be immediately reflected in the DApp, however, we do not request a signature for the update. **THIS IS FOR DEMONSTRATION PURPOSES ONLY.**

The intention is that edits to content will require another signature to ensure verifiability.

Deleting Posts

We allow users to “delete” posts which removes them from storage in the Spotlight smart contract. Deleted posts will be reflected immediately by the DApp, but like deleted profiles, because they are currently stored on-chain, they are never truly deleted.

Paywalling Posts

Because wallets generally leverage asymmetric public/private keys, in many instances, these pairs can not only be used for **signatures**, but also **encryption**. More specifically: encryption with a public key that can only be decrypted with the associated private key. We leveraged this idea in order to solve the problem of: *“how do we keep posts exclusive while publicly available on the blockchain?”* MetaMask provides the [eth-sig-util\[8\]](#) library which provides some utilities facilitating encryption of data with a wallet's public key, which in turn, also makes it so that only that user's private key can decrypt the data. We allow creators to make content exclusive by creating a “Paywall Post” which will encrypt the post with their public key and store it on-chain. They can decrypt it at any point they like.

Other users can see this post and opt to purchase this content directly from the creator. For now, all posts cost a flat fee of 0.1 ETH. A purchaser provides their public key and pays the fee which goes to the Spotlight contract. Our contract acts as an escrow for these purchases. **(NOTE:** No methods exist on the contract for extraction or transference of ETH except between the purchaser and the creator in the context of a specific piece of content).

For now, creators can accept or reject offers to purchase their content. If they reject the offer - the purchaser is given back their funds. If the creator accepts the offer, then the dApp decrypts the post with the creator's private key, re-encrypts the post with the purchaser's previously provided public key, and this re-encrypted copy is stored on chain. The purchaser, in their feed, can now decrypt and view the purchased post whenever they wish.

If a purchaser no longer wishes to purchase the post, they can cancel the purchase at any time, so long as it is **before** the creator has accepted the purchase.

Comments

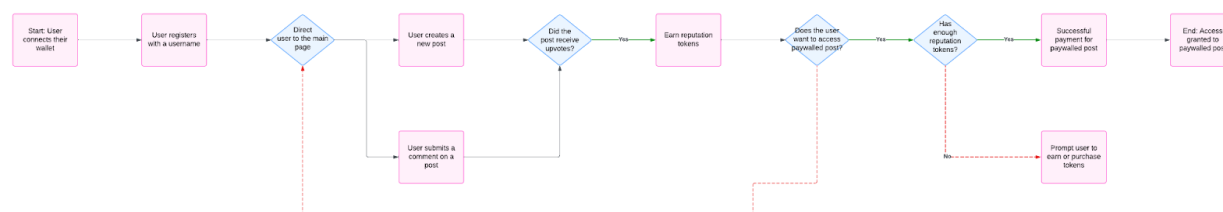
The comment system in Spotlight allows users to engage more deeply with posts, fostering a sense of community and discussion similar to traditional platforms like Reddit. Users can leave comments on posts they find interesting or relevant. Like Posts, comments are linked to the user's wallet address and stored on-chain ensuring decentralized storage and immutability.

Upvotes / Downvotes

The upvote/downvote system in Spotlight serves as the core mechanism for content ranking and reputation management. Inspired by traditional platforms like Reddit, it allows users to signal approval or disapproval of posts and comments, fostering community-driven content curation.

Users can currently upvote or downvote posts. Each vote is linked to the user's wallet address, ensuring accountability and transparency. A single user can vote only once per post with the option to reverse or remove their vote.

Upvotes and downvotes directly impact the Reputation Token (RPT) balance of the post creator. When a post is upvoted, the **creator of the post** is issued 100 RPT. This is the largest issuance of RPT in the ecosystem. The idea here is that the best way to earn RPT (reputation!) is to provide quality content that the community engages with in a positive manner. The next section will go into more detail about how RPT is earned and burned.



Tokenomics

As we discussed in class, one of the main challenges of a social media application, web3 or otherwise, is keeping the user base engaged and coming back. In traditional social media applications, while the content creators get some recognition and compensation for extremely popular content, one thing that remains common is that the content consumers get no rewards for their engagement with the ecosystem. An old adage is a recurring theme in most web2 social media applications: *“If you’re not paying for the product, **you** are the product.”* In the following sections we’ll explore how to earn RPT, how it affects the DApp, and disincentives to prevent “gaming the system.”

Earnings and Burnings

We mentioned in [Upvotes / Downvotes](#) one way to earn RPT. The full list of burning/earning is as follows:

- Create a post
 - Creator issued 1 RPT
- Upvote a post
 - Post creator issued 100 RPT
 - Upvoter issued 1 RPT
- Downvote a post
 - Post creator loses 5 RPT
 - Downvoter issued 1 RPT
- Add a comment
 - Commenter issued 1 RPT
- Purchase a post
 - Purchaser issued 1 RPT

Now that we've covered how to gain RPT in Spotlight, let's discuss what the token does for users in Spotlight.

Token Purpose

NOTE: The following are features we were not able to implement but were the driving motivations behind introducing an ERC-20 token to measure reputation of creators and their posts.

The primary purpose of the token was to affect a new way of driving content visibility. Traditionally, the more “upvotes” or “likes” a piece of content generated, the more visible it becomes. Instead, Spotlight does not have to be bound only to “number of likes” as the “metric of visibility.” Users with higher RPT have higher visibility on their posts. We still have the number of upvotes and downvotes, and can thus provide users the traditional sorting of posts based on these metrics as well, however, just because you have a lot of likes does not necessarily mean you're a **consistently** reliable content creator. (More on why “consistently” is important in the next section.)

Because RPT is an ERC-20 token, we have the added benefit of it being transferable. A user could directly transfer their RPT to another user and **immediately** have an impact on their content's visibility. Traditional social media requires another user to boost their content using re-posts or the like, which can detract from the content or creator being boosted. With RPT transfer, the focus continues to remain on the content and the creators. There is currently no UI support for this, but it is available by default of the ERC-20 interface of RPT.

Another component of RPT allows for tiered rewards seen in other web3 applications to further deepen engagement with its user base, such as access to exclusive NFTs.

Decay Mechanism

You might rightly ask yourself: *“But if visibility is based on the amount of RPT you have, doesn't that mean a select few would constantly be visible?”* In our case, the answer is no, because we built into our contract an inherent token decay mechanism. We currently [decay 1% of your RPT every 15 minutes](#). There are 3 primary motivations behind introducing a decay mechanism.

First and foremost, we wanted the token to embody an enablement of the ecosystem and its users as opposed to being a driver of equity. That said, posts with high engagement that generated a large amount of RPT for the creator could be used to infer a kind of monetary “value” of the content, but the intention was never for RPT to be bought/sold on exchanges.

Next, digging more into the ethos of “enabling the ecosystem and users,” by having a decay mechanism, it ensures that while your content may have been hot today, it makes no guarantees for tomorrow. If you want to stay relevant, you need to stay engaged and continue producing high quality content and engage with the ecosystem. We believe this would give the “stickiness” effect for users - to keep that RPT, you need to keep coming back!

Finally, by adding a decay mechanism, it allows for a completely dynamic ecosystem of content. There's always something new bubbling up to the top. It actually was the source of the platform's namesake: different users can get their 15 minutes in the spotlight.

How to prevent “gaming the system”

All the values mentioned above for issuance, burning, and decay rate are purely for demonstration purposes. In reality, these values are not tuned enough to prevent bad actors from generating RPT inorganically (i.e. not from true community engagement) and taking over the spotlight (🤪).

However, we believe that these are enough levers that should suffice for mitigating such behavior. At the end of the day, a user upvoting a post should always be the largest generation and earning of RPT. All other engagement rewards (commenting/etc) should always pale in comparison to the amount an upvote on a post generates. We want to reward engagement, but it must be balanced with making it that inorganically generating RPT is extremely expensive. At the end of the day, any engagement with Spotlight will at minimum cost the user gas fees. While they still need to be tuned, we believe this should be enough to ward off misuse of the platform.

The decay rate and downvotes also play an important role here as well. Even if someone had the funds to behave in this manner, the community could very quickly bury the creators' content by simply downvoting them into the background.

Contract Details

The following section provides an overview of the data structures and methods that drive the features enumerated above within the UI. While there are multiple contracts that power Spotlight, all features are driven by methods accessed through the Spotlight contract, located in [Spotlight.sol](#). All methods listed below are said entry points in [Spotlight.sol](#) and most certainly call other contracts, but they will not be enumerated here in the interest of brevity. All of our contracts can be found [here](#).

User Profiles

Data structures

The [user profile struct](#) stores a single user's username and optionally the IPFS CID of their avatar, should they choose to upload one. The `reputation` field is meant for serialization purposes: whenever a user's profile is requested, we look up their current RPT and include it in the response.

Unset

```
struct Profile {
    string username;
    string avatarCID;
    uint256 reputation;
}
```

[mapping\(address => Profile\) internal profiles](#)

This mapping stores the mapping of a user's wallet address to their profile.

mapping(bytes32 => bool) internal normalized_username_hashes

This mapping stores the normalized hashes of existing usernames and facilitates enforcement of username uniqueness.

Public Methods

modifier usernameValid(string memory _username)

Modifier to ensure that a username meets the length requirements.

modifier onlyRegistered()

Modifier to ensure that only registered users can perform certain actions.

Calls `is_registered` on `msg.sender`

function isRegistered(address a) public view returns (bool)

Check if an address is registered with a profile, using `profiles` mapping

function registerProfile(string memory _username) public usernameValid(_username)

Given a `_username`, register `msg.sender` to Spotlight. The username is validated and checked for uniqueness after normalization (case-insensitive)

function getProfile(address a) public view returns (Profile memory)

Retrieve a profile by address from `profiles`. The profile must exist for the specified address.

function updateUsername(string memory _newUsername)

public onlyRegistered usernameValid(_newUsername)

Update the username for the profile of the `msg.sender`. Their profile must exist.

function updateAvatarCID(string calldata _cid) public onlyRegistered

Update the profile avatar CID on IPFS for `msg.sender`. Their profile must exist.

function deleteProfile() public onlyRegistered

Delete the profile `msg.sender`. Their profile must exist.

Posts

Data structures

The `Post struct` contains all the metadata necessary to render and verify a post. At creation, the `id` field and the `signature` field are the same.

Unset

```
struct Post {
    address creator;
    string title;
    string content;
```

```

    bytes id;
    bytes signature;
    bool paywalled;
    uint256 nonce;
    uint256 createdAt;
    uint256 lastUpdatedAt;
    uint256 upvoteCount;
    uint256 downvoteCount;
}

```

The [Comment struct](#) is simpler.

```

Unset
struct Comment {
    address commenter;
    string content;
    uint256 createdAt;
}

```

We also have a [PendingPurchase struct](#), meant to track which posts are still awaiting settlement. This struct tracks the wallet address of the purchaser, the post ID they wish to purchase, and the public key that should be used to encrypt the content upon the creator's acceptance of the purchase.

```

Unset
struct PendingPurchase {
    address purchaser;
    bytes postId;
    string pubkey;
}

```

[bytes\[\] internal communityPostIDs](#)

A list of all post IDs created by all users in the community.

[mapping\(address => bytes\[\]\) internal profilePostIDs](#)

Mapping which provides all post IDs for a given user's wallet address

[mapping\(bytes => PostLib.Post\) internal postStore](#)

Mapping which provides the `Post` struct (aka - the actual post content) for a given post ID

mapping(bytes => PostLib.Comment[]) internal postComments

Mapping which provides a list of `Comment` structs for a given post ID

mapping(bytes => mapping(address => bool)) public upvotedBy
mapping(bytes => mapping(address => bool)) public downvotedBy

Mapping which provides a true or false value, indicating whether a given wallet address has upvoted or downvoted the given post ID. The key ordering is `postID` followed by `address`.

mapping(bytes => mapping(address => string)) internal purchaserPublicKeys

Mapping which stores the public key of a wallet address to be used for a given post ID. The key ordering is `postID` followed by `address`.

mapping(address => PendingPurchase[]) internal pendingPurchases

Mapping providing the list of `PendingPurchase` structs for a given creator's wallet address

mapping(address => mapping(bytes => PostLib.Post)) purchasedPosts

Mapping providing the purchased `Post` struct, generated by the creator after accepting the purchaser's payment, which only the purchaser can decrypt.

Public Methods

function createPost(
string memory _title,
string memory _content,
uint256 _nonce,
bytes calldata _sig,
bool _paywalled
) public onlyRegistered

Create a post for the given content, where the `creator` is `msg.sender`. Their profile must exist. The given signature must be valid. Grants 1 RPT to `msg.sender` as a reward for engagement.

function getPostsOfAddress(address _addr) public view returns (PostLib.Post[] memory)

Get a list of `Post` structs for the given address. Read from `profilePostIDs`

function getPost(bytes calldata _post_sig) public view returns (PostLib.Post memory)

Get the `Post` struct for a given post ID. Read from `postStore`

function getCommunityPosts() public view returns (PostLib.Post[] memory)

Get all posts in the community. Reads `communityPostIDs` to populate the list of `Post` structs to return.

function editPost(bytes calldata _id, string calldata newContent)
public onlyRegistered

Update the `content` field for a given post ID that belongs to `msg.sender`. Their profile must exist. The post ID must exist. They must be the `creator` of the `Post`.

`function deletePost(bytes memory _id) public onlyRegistered`

Delete the `Post` for a given post ID of `msg.sender`. Their profile must exist. The post ID must exist. They must be the `creator` of the `Post`.

`function upvote(bytes calldata _id) public onlyRegistered`

Track `msg.sender` gave an upvote for the target post ID. Their profile must exist. The post ID must exist. They may not have already upvoted the post. Grants 100 RPT to the post creator. Grants 1 RPT to `msg.sender` as a reward for engagement. If `msg.sender` had previously downvoted this post, reverse those states.

`function downvote(bytes calldata _id) public onlyRegistered`

Track `msg.sender` gave a downvote for the target post ID. Their profile must exist. The post ID must exist. They may not have already downvoted the post. Burns 5 RPT from the post creator. Grants 1 RPT to `msg.sender` as a reward for engagement. If `msg.sender` had previously upvoted this post, reverse those states.

`function upvotedBy(bytes calldata _id, address _addr) public view returns (bool)`

Returns true or false if the given address has upvoted the given post ID.

`function downvotedBy(bytes calldata _id, address _addr) public view returns (bool)`

Returns true or false if the given address has downvoted the given post ID.

`function addComment(bytes calldata _id, string calldata _content)`

`public onlyRegistered`

Stores a `Comment` and adds it to `postComments` for the given post ID. `msg.sender` must have a profile. The post ID must exist. Grants 1 RPT to `msg.sender` as a reward for engagement.

`function getComments(bytes calldata _id)`

`public view returns (PostLib.Comment[] memory)`

Returns the list of `Comment` structs for a given post ID.

`function isPurchasePending(bytes calldata _id) public view returns (bool)`

Return true or false if `msg.sender` has a pending purchase that has yet to be settled for a given post ID. **NOTE:** While we check `msg.sender`, this is a read only function and there are no fees associated with it.

`function purchasePost(bytes calldata _id, string calldata _pubkey)`

`public payable onlyRegistered nonReentrant`

Track that `msg.sender` wants to purchase the given post ID. The public key to use for the post encryption is also provided. `msg.sender` must have a profile. `msg.value` must satisfy the minimum fee of 0.1 ETH. The post ID must exist. The `Post` must have the property `paywalled`

set to `true`. Grants 1 RPT to `msg.sender` as a reward for engagement. This function is explicitly [non-reentrant](#).

`function getPendingPurchases() public view returns (Posts.PendingPurchase[] memory)`

Get the list of all `PendingPurchase` structs for `msg.sender`. **NOTE:** The frontend handles filtering which `PendingPurchase` objects to render based on which post ID is currently viewed on the page. **NOTE:** While we check `msg.sender`, this is a read only function and there are no fees associated with it.

`function declinePurchase(bytes calldata _id, address payable _purchaser)
public onlyRegistered nonReentrant`

Untracks that `purchaser` wanted to purchase the given post ID. `msg.sender` must have a profile. `purchaser` must have a `PendingPurchase` for the post ID. The post ID must exist. The `Post` must have the property `paywalled` set to `true`. The 0.1 ETH fee is returned to the `purchaser`. This function is explicitly [non-reentrant](#).

`function acceptPurchase(bytes calldata _id, address _purchaser, string memory
_content)
public onlyRegistered nonReentrant`

Settle the transaction between the `purchaser` and `msg.sender`. `msg.sender` must have a profile. `purchaser` must have `PendingPurchase` for the given post ID. The post ID must exist. The `Post` must have the property `paywalled` set to `true`. `msg.sender` must be the `Post.creator`. The new `content` must be stored in `purchasedPosts`, so the `purchaser` can access their encrypted copy. The 0.1 ETH fee must be transferred to `msg.sender`. This function is explicitly [non-reentrant](#).

`function getPurchasedPost(bytes calldata _id) public view
returns (PostLib.Post memory)`

Get the purchased, decryptable copy of a `Post` for a given post ID for `msg.sender`. **NOTE:** While we check `msg.sender`, this is a read only function and there are no fees associated with it.

Frontend Details

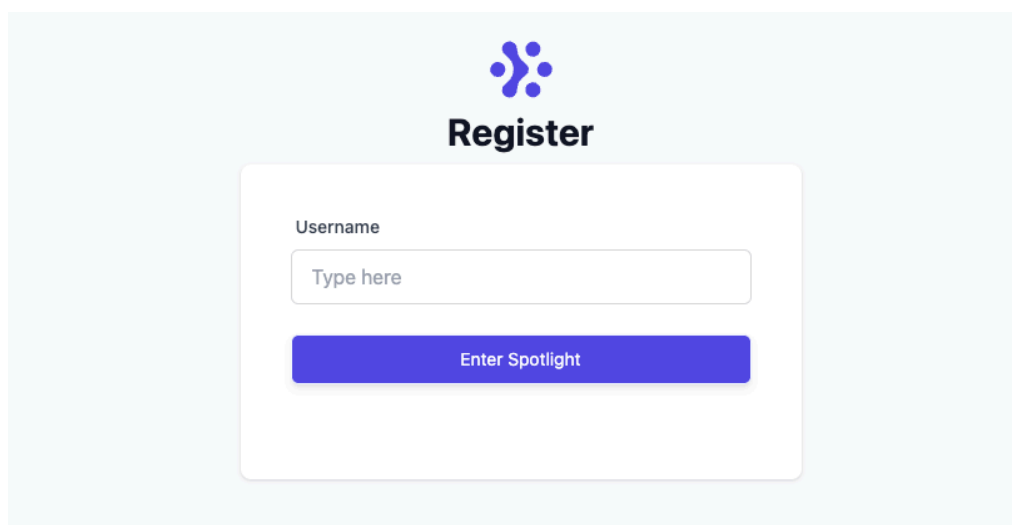
Navigation Route

The routing structure of the website ensures smooth navigation between pages, with each route corresponding to specific user actions and page views. Below is a breakdown of the main navigation flow and URLs:

From	To
Login Page (/)	Feed Page

Feed Page (/feed)	Home Page
	Post Detail Page
Home Page (/home)	Feed Page
	Post Detail Page
Post Detail Page (/feed/viewPost?postSig={sig})	Feed Page
	Home Page

Login Page



The image shows a 'Register' form on a light blue background. At the top is a purple logo consisting of four dots arranged in a square pattern. Below the logo is the word 'Register' in bold black text. The form itself is a white rounded rectangle containing a 'Username' label, a text input field with the placeholder 'Type here', and a blue button labeled 'Enter Spotlight'.

On the login page, users can register by creating a unique username. The system checks the availability of the username in real time. If the chosen username is already taken, a pop-up notification appears at the top of the page, informing the user of the conflict. Upon successful registration, users are automatically redirected to the feed page on subsequent visits, ensuring a seamless experience.

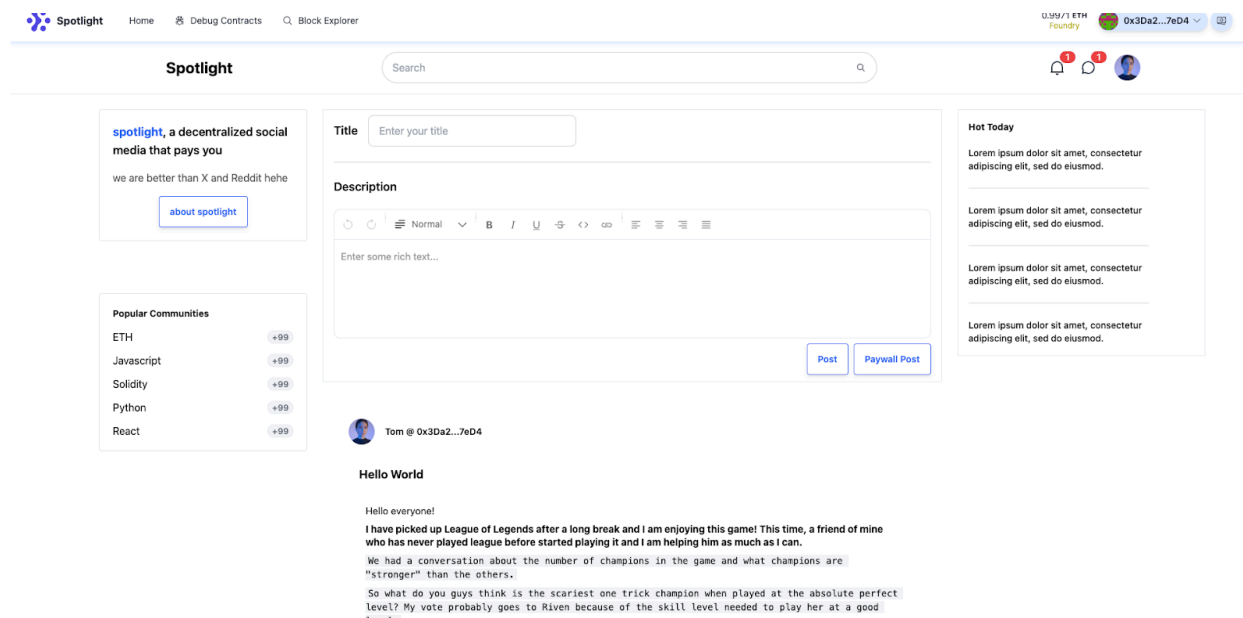
Supported Features :

- Register profile
- Auto login based on wallet address

Contract Used :



Contract Name	Function Name	Usage
Spotlight	registerProfile	Register a user

Feed Page



On the feed page, users can create posts by entering a title and body text. If the user is connected via MetaMask, they have the option to create paywall posts, which require a token for others to view. The post editor supports rich text formatting using the 'Lexical.js' library. After creating a post, users are automatically redirected to the post detail page to view their newly created post.

On the feed page, users can browse community posts. If no posts are available, a hint message is displayed. Users can engage with posts by upvoting or downvoting, promoting interaction within the community. By clicking on a post, users are navigated to the post detail page, where they can leave comments. Additionally, users can easily navigate to the home page by clicking on the avatar in the top right corner.

At the bottom of each post, a badge is displayed to indicate its verification status. A green badge () is shown for verified posts, while a red badge () is displayed for unverified posts, providing users with a clear indication of the post's authenticity.

Supported Features :


- Create a post
- Create a paywall post
- View community posts
- Navigate to the post detail page
- Navigate to the home page
- Upvote and downvote
- Verify the signature of posts

Contract Used :



Contract Name		Function Name	Usage
Spotlight	Post	createPost	Create a post and reward with reputation token
	Reputation	engagementReward	
Spotlight	Post	upvote	Upvote a post and reward with reputation token
	Reputation	engagementReward	
Spotlight	Post	upvote	Downvote a post and reward with reputation token
	Reputation	engagementReward	
Post		getCommunityPosts	Get the community posts

Post Detail Page

[← Back](#)



Tom
0x3Da224D548F7A01957049815b04fa1d4A987eD4

Hello World




Hello everyone!

I have picked up League of Legends after a long break and I am enjoying this game! This time, a friend of mine who has never played league before started playing it and I am helping him as much as I can.


We had a conversation about the number of champions in the game and what champions are "stronger" than the others.

So what do you guys think is the scariest one trick champion when played at the absolute perfect level? My vote probably goes to Riven because of the skill level needed to play her at a good level.

Thank you in advance.


 (0) |
 (0) |


Comments (1)



Add a comment...

Submit



Tom
0x3Da224D548F7A01957049815b04fa1d4A987eD4
This is a comment
2024/12/1 20:18:00

On the post detail page, users can view the full content of the post and can upvote or downvote it. They can also leave comments in the comment section below. If the user is the

post owner, they can edit or delete the post using the button in the top-right corner. Clicking the edit button will open a panel and editor, allowing the owner to modify the title and body of the post. Users could go back to their previous page by clicking the 'back' button in the top-left corner.

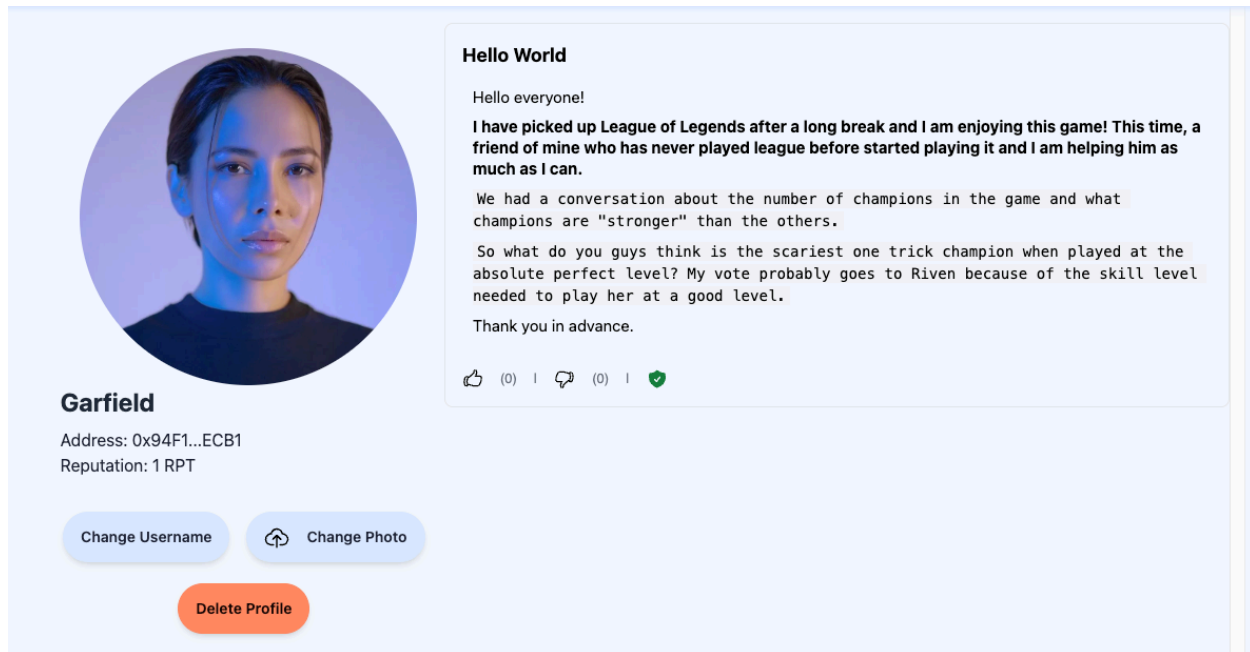
Supported Features :

- Browse the whole post
- Upvote and downvote
- Comment on the post
- Edit and delete the post (Only Owner)
- Verify the signature of posts
- Pagination for comments
- Navigate to the home page
- Return to previous page

Contract Used:

Contract Name		Function Name	Usage
Spotlight	Posts	addComment	Create a comment and reward user with reputation token
	Reputation	engagementReward	
Spotlight		getProfile	Display user avatar, name, and address
Posts		getComments	Display all comments of this post
Posts		getPost	Get the post and display it
Posts		editPost	Edit owner's post
Posts		deletePost	Delete owner's post

Home Page



On the homepage, users can browse their posts, change their username, update their avatar, and delete their account. They can access the homepage by clicking their avatar in the top-right corner of any page. When users update their avatar, the image is securely stored on IPFS via web3.storage. Users can view the details of their posts by clicking on them, which will navigate them to the post detail page.

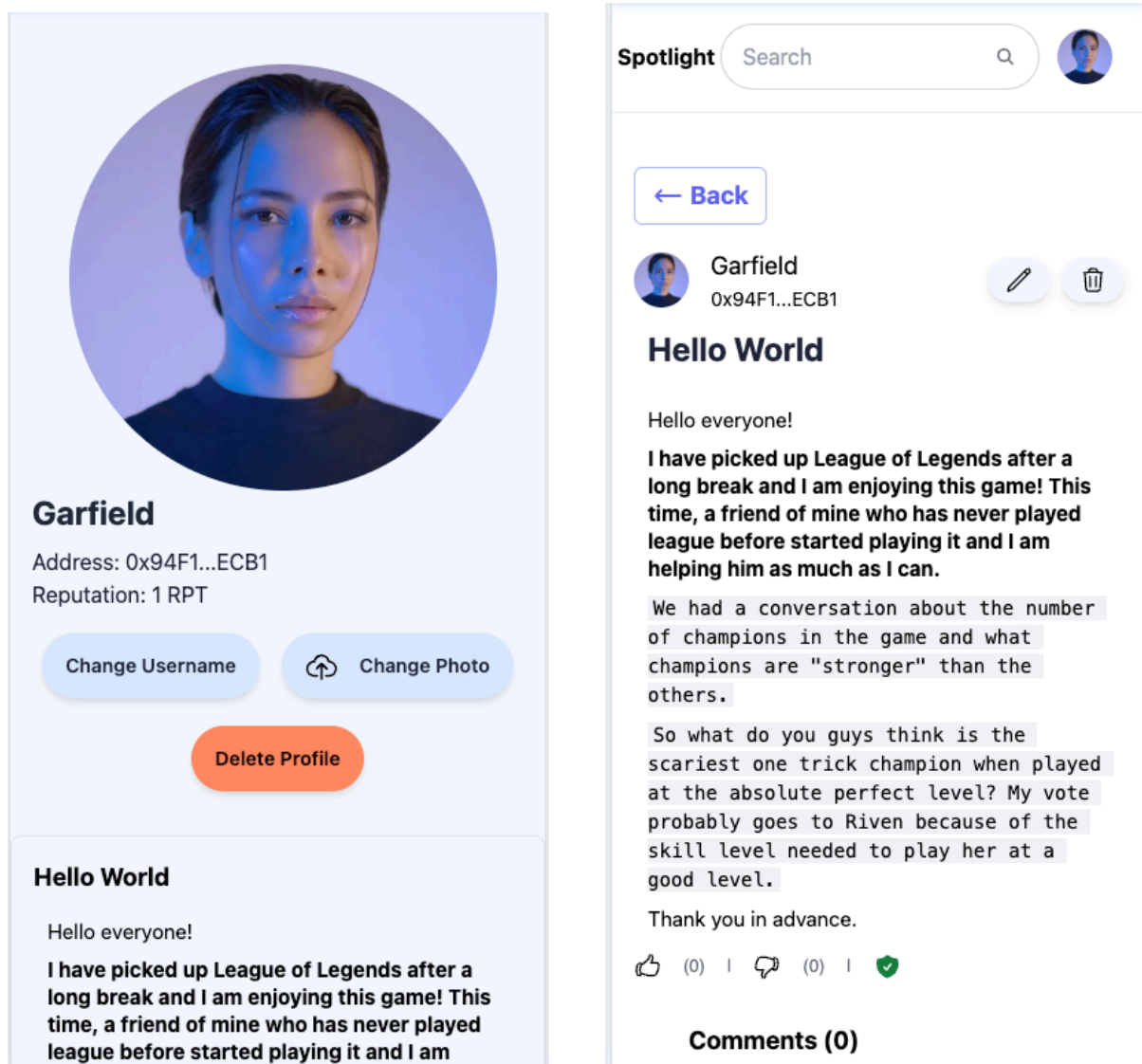
Supported Features:

- User posts display
- Change username
- Change user avatar
- Delete profile
- Navigate to the post detail page

Contract Used:

Contract Name	Function Name	Usage
Posts	getPostsOfAddress	Display user's post
Spotlight	updateAvatarCID	Update user's avatar
Spotlight	updateUsername	Update username
Spotlight	deleteProfile	Delete account

Responsive Layout



Our website is designed with a fully responsive layout, ensuring optimal display and usability across a wide range of devices, including mobile phones. We prioritize a seamless user experience by making sure that all body content is easily accessible and properly displayed, without compromising functionality or design. Through the use of media queries, we customize the layout, frame, and positioning of elements to adapt to different screen sizes, providing a smooth and intuitive browsing experience on both small and large devices.

Quality Assurance

Our primary method of ensuring we kept our DApp stable and functional was by keeping our test coverage exceptionally high. Our current coverage report is:

File	% Lines	% Statements	% Branches	% Funcs
PostLib.sol	100% (3/3)	100.00% (6/6)	100.00% (0/0)	100% (2/2)
Posts.sol	100% (121/121)	97.63% (165/169)	97.14% (34/35)	100% (22/22)
Reputation.sol	100% (63/63)	94.74% (90/95)	73.68% (14/19)	100% (15/15)
Spotlight.sol	100% (82/82)	99.11% (111/112)	92.31% (12/13)	100% (29/29)
Total	100% (269/269)	97.38% (372/382)	88.24% (60/68)	100% (68/68)

A number of tests go over the same code paths multiple times, as our contract unit tests were written in a manner that described expected behaviors of interactions, as opposed to testing low-level implementation details. This was instrumental when we hit the 24KB contract size limit and were required to separate our contracts. We could do so confidently with a robust test suite. All of our unit tests can be found [here](#).

Another useful tool was a [bot we built](#) in python that would interact with our local testnet. It triggers most supported read & write contract operations at random. This helped us identify some unexpected edge cases.

Future Features

As with most projects, our plans and ambitions exceeded the timeline allotted. The following section goes over features we felt were valuable to the long term success of Spotlight, but simply did not have time to ship in a functional state or even get started on them.

Post Media

While we currently only support text based posts, which for the most part, are trivial for storing on-chain, the eventual goal for Spotlight is to support any media - images, videos, audio, etc. This is why the signature component of our `Post` struct is so important: no matter what data or where it is stored, we can always verify its creator's authenticity.

Off-chain Storage

This was something discussed from the very beginning of the project that we knew would be an issue for true scalability, not only for storage, but for gas fees. Ideally, our Spotlight contract `Post` struct would only need to store CID pointers to IPFS locations of the post content. By only storing CID pointers it would significantly lower fees because we'd be storing significantly less data when compared to the entire content size off-chain. It would also allow for better contract storage optimization through more compact data packing, which would further reduce gas fees. Profile avatars were the first step in providing of a proof-of-concept for usage of IPFS via web3.storage, however, as we'll review in our [Lessons Learned](#) section,

web3.storage is still VERY slow due to how the UCAN protocol works. This had a noticeable (painful) impact on user experience.

Communities, DAOs, and Moderation

An early feature that we did not get a chance to implement was the idea of communities. Conceptually, a community is simply a logical grouping of posts, but our intent was much more sophisticated than that. A community could be solely managed by its creator, similar to a reddit mod. But a more interesting twist we thought of was if each community was its own DAO. The community DAO would be responsible for self-moderation. Voting and engagement would be facilitated by Spotlight's RPT. Some communities could be more "private" and require an invitation approved by the DAO, whereas others may be completely open. Again, any form of "privacy" would require dependency on a reliable encryption/decryption scheme.

Not only should communities moderate themselves, but users will need to be able to curate the content they see with additional features such as "Follow User" or "Block User." The "Block" feature would also be applicable to communities - just because something is popular, doesn't mean you're necessarily interested in it.

Encryption/Decryption of Paywall Posts

This feature is built on some methods that MetaMask and others have said are deprecated and will most likely be removed in the future. The primary reason has to do with the fact that data of the public key is being leaked with each use, which is very reasonable. In order to properly future-proof this feature, one alternative we've been following is [ERC-5630: "New approach for encryption / decryption"](#) however, this is still only a draft. Should the encryption/decryption methods be fully removed from the wallet providers, our paywall feature would need to be shelved until further notice.

Paywalls and Content Monetization

As mentioned, our paywall implementation is a flat fee of 0.1 ETH which is hardcoded into the Spotlight contract. We'd like to move away from a "one-size fits all approach" and instead, allow creators to set their own fee, modify the fee, have discounts for specific user types & communities, and allow other users to suggest prices (i.e. "haggling").

We also would like to eventually have a dashboard for creators to better visualize and measure the success of their exclusive content. Adjacent to this, we'd like to build a kind of notification system for creators of when purchases of their content occur. Currently the encryption/decryption happens in the wallet, which is why our "paywall protocol" requires the creator to settle the post purchase. Ideally, the content is issued directly to the purchaser without this extra step, but this, too, is dependent on a more robust encryption/decryption scheme.

Search and Sorting

As mentioned in our [Tokenomics](#) section, the primary function of RPT would be to facilitate a new form of post sorting. Aside from reputation based sorting, we want to support sorting posts by their created date and also by the amount of engagement (count of upvotes

and downvotes). Even within engagement there are other sorting options such as “most liked” and “least liked” that could be provided.

Another ubiquitous social media feature is “search.” For Spotlight it would involve the ability to search for users, communities, and posts. An approach we thought of was potentially using a web2 data store like MongoDB which has some good search capabilities.

Expanding Comments

Comments are another area that we were able to just barely scratch the surface of. The directions of comments includes upvote and downvote for comments, which would also provide some RPT engagement rewards and nesting comments, giving that sense of better organized conversations which can be seen on many occasions in reddit.

Deeper Robustness for RPT

As mentioned in our [Tokenomics](#) section, we have a number of levers available to us when it comes to issuance, burning, and decay rate of RPT, but they are all hard coded and static. A more interesting take on this would be a more dynamic set of rules around the supply of RPT based on

A more straightforward feature to be implemented that we believe would have a powerful community impact would be adding RPT transference into the UI. These functions are only available on the ERC-20 token contract methods.

Decentralized Hosting

We’ve discussed decentralized storage for our user’s content, but the app itself could go further than where it is today. We’re currently running our demo on a GCP hosted instance. Instead, it would be nice to lean even more into the web3 ecosystem by leveraging hosting solutions with tools like ENS and IPFS as the CDN[\[9\]](#).

Lessons Learned

Web3 Tooling, their responsiveness and impact on UX

During the semester we revisited multiple times about the idea that decentralization has a direct impact on speed - generally, things get slower. When developing locally, transactions are immediately confirmed. It’s snappy, the UX feels great. We did an experiment where we deployed to Sepolia and the experience was...painfully slow. Research has been done about extra milliseconds can have a detrimental effect on conversion rates in e-commerce[\[10\]](#). If there’s any hope for social media to be successful in a web3 ecosystem, things need to operate at significantly faster speeds. This delay was felt even using the Polygon network ([contract address](#)).

In the vein of speed and responsiveness, while web3.storage has excellent tooling, it too was also very slow. We did our MVP demo using it and Kele inquired as to why our post

creation was so slow. It would take 5 - 10 seconds to get the data uploaded, and this was before we took into account transaction time (again, negligible on local, not so much on SepoliaETH). It was then that we opted to keep posts on-chain for the time being.

Once again, the user experience was abysmal when in the context of social media. That said, we could have opted to host our own IPFS node and interface directly with it but the interesting thing that web3.storage brings to the table is the usage of the UCAN[11] protocol and its ability to delegate “upload” capabilities to end-users in a transparent manner[12]. In fact, this is what we’re using to drive the “Upload Profile Photo” feature in Spotlight. The delegation of only “upload” capabilities is quite useful, otherwise, in IPFS, we would have to start implementing permissions of who can delete content. Maybe this would be the way to go in the future, but this was not something we had time to more deeply explore.

More “gaming-the-system” concerns

While we have addressed some ways we believe our [Tokenomics](#) should help keep the ecosystem healthy with respect to preventing RPT farming, there is always the concern of “whales.” Some with a large amount of ETH could create many wallet addresses and thus many accounts, and begin liking their own posts, thus artificially boosting their RPT.

Community DAOs could help combat these types of RPT farmers by proposing their own safety mechanisms with proposals such as disallowing posts from the user, heavy losses of RPT, or prevention of RPT earnings for some period of time. Piggy backing off the [Robustness for RPT](#) section, downvoting could become more sophisticated by keeping track of the “rate of downvotes” so as to be able to apply the following logic: *“if you are receiving many downvotes in a short period of time, Spotlight will dynamically start to increase the amount of RPT burned, as the community is trying to indicate that the particular post or user is a bad actor and needs more extreme measures to ensure ecosystem health.”*

General Solidity Learnings

There were a few things that we learned during the development of our smart contracts

- Easy gas optimizations: custom errors instead of strings through [require](#)[13]
- Contract size limits and workarounds
 - This cost us a few days of development work as we had carefully split apart existing functionality across multiple contracts.
- Overflow protections are already built into our compiler version of Solidity!

Appendix

Spotlight Codebase

<https://github.com/juannyG/coms6998/tree/dev>

Solidity Contracts

<https://github.com/juannyG/coms6998/tree/dev/packages/foundry/contracts>

- [Spotlight.sol](#) - primary entry point to to any DApp interactions
- [Reputation.sol](#) - RPT; ERC-20 token responsible for reputation tracking
- [PostLib.sol](#) - Library holding common structures and functions specific to posts
- [Posts.sol](#) - Separate contract responsible for storing and managing posts
- [Events.sol](#) - Events that the contract(s) can emit
- [SpotlightErrors.sol](#) - Errors that the contract(s) can return

Unit Tests

<https://github.com/juannyG/coms6998/tree/dev/packages/foundry/test>

NextJS App

<https://github.com/juannyG/coms6998/tree/dev/packages/nextjs>

Spotlight-Bot

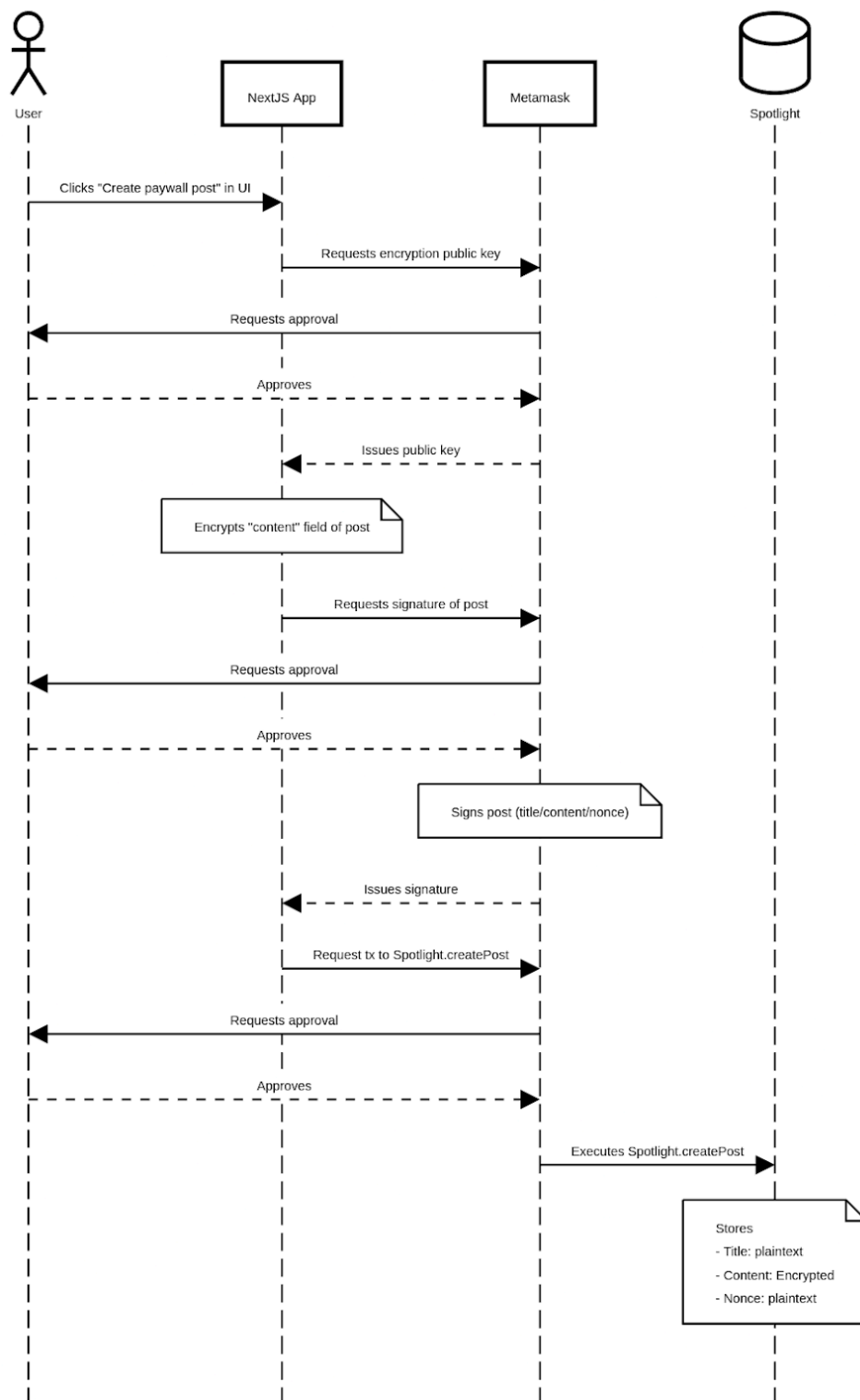
<https://github.com/juannyG/coms6998/tree/dev/spotlight-bot/bot>

References

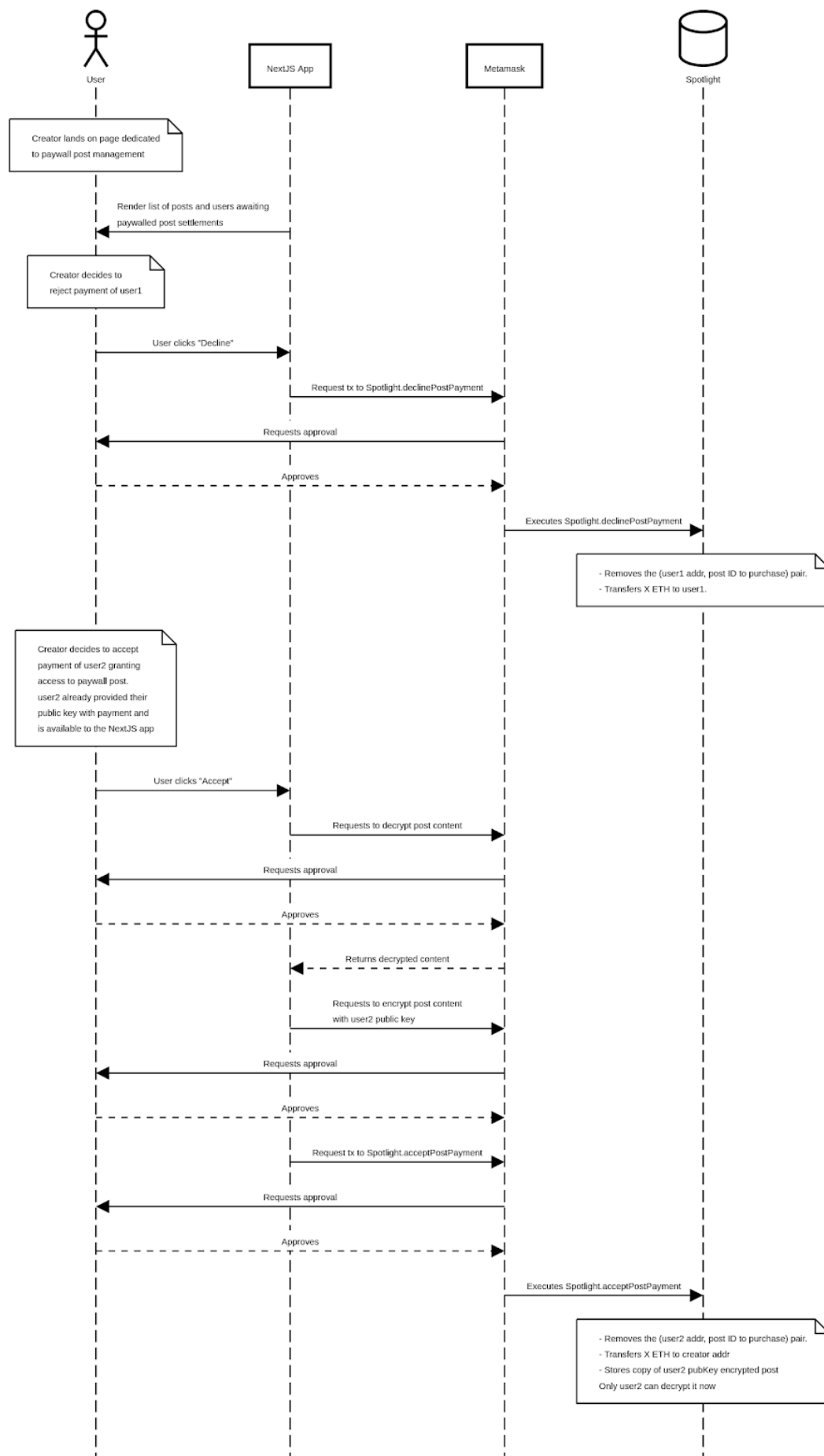
- [1] <https://wagmi.sh/react/getting-started>
- [2] <https://viem.sh/docs/introduction>
- [3] [web3.storage](#)
- [4] [UCANs and web3.storage](#)
- [5] [Scaffold-ETH 2](#)
- [6] [Foundry Book - Anvil Reference](#)
- [7] [ERC-191: Signed Data Standard](#)
- [8] [eth-sig-util - A small collection of Ethereum signing functions](#)
- [9] [Link a domain | IPFS Docs](#)
- [10] [The Impact of Web Pages' Load Time on the Conversion Rate of an E-Commerce Platform](#)
- [11] [User Controlled Authorization Network \(UCAN\) Specification v0.9.2](#)
- [12] [App owned accounts - transparent to the user](#)
- [13] [Custom Errors in Solidity](#)

Paywall Post Sequence Diagrams

Creating paywalled post



Satisfying Paywalled Post Payments



Paying & Canceling Payment for Paywalled Post

