Quests NFT Technical Specification

Overview

A key feature of Quests.xyz, the platform for creating a socially-verified portfolio of quests, is the creation of a commemorative Quest NFT.

This token is minted and distributed to all contributors at the end of a quest and represents a unit of work in their portfolio. These quests contribute to a user's profile on the platform, where they can showcase all the quests they've led and/or contributed to and all the social verification they've received for that work.

The purpose of this document is to define the technical specification for the system of creating and managing NFTs that represent completed quests. It starts by laying out the assumptions made in the design of the system, including how to treat transaction fees and contract interactions. It then provides a high-level overview of the system's architecture. In describing the system's architecture, this document explains how the on-chain components of the system interact with each other and how the overall on-chain system will interact with the off-chain web application. This document then goes on to describe the main set of features of the system conceptually and how they technically interact in the system's architecture. The final section of this document provides a description of the important methods in the smart contract API.

Assumptions

The design of the Quests NFT system makes a couple of novel assumptions about how the system should work. Most of these assumptions stem from a base assumption, which this section describes first. It then goes on to outline the other assumptions that influenced the system's design.

Most transactions are created and executed by a set of programmable bot wallets.

The primary assumption made in the design of this system is that most transactions with the smart contracts are made by a set of programmable bot wallets operated by Quests.xyz. While this is a significant deviation from how smart contract systems are typically designed, this assumption is made on the basis that it will lead to a better end-user experience, especially for non-crypto-native users.

These programmable wallets will be managed and cycled by the Quests.xyz team, such that one is never in use for a long period of time. This will mitigate the risk of exposing the private keys for one of the bot wallets. Additionally, the system will have a multi-sig wallet that acts as a super-admin user, that is able to remove wallet addresses from the system, should any of the private keys be exposed.

Gas is an infrastructure cost.

Stemming from the previous assumption, the design of the Quest NFT system assumes that blockchain transaction fees, commonly referred to as gas, are an infrastructure cost borne by the system and not the end user. This is inspired by systems like Proof of Attendance Protocol (POAP), where users never directly invoke and pay for transactions themselves. This leads to a better end-user experience in the design and implementation of the front end. It also makes having an EOA wallet and being crypto-native optional for using the Quests.xyz platform.

Verification of wallet ownership happens in the front-end.

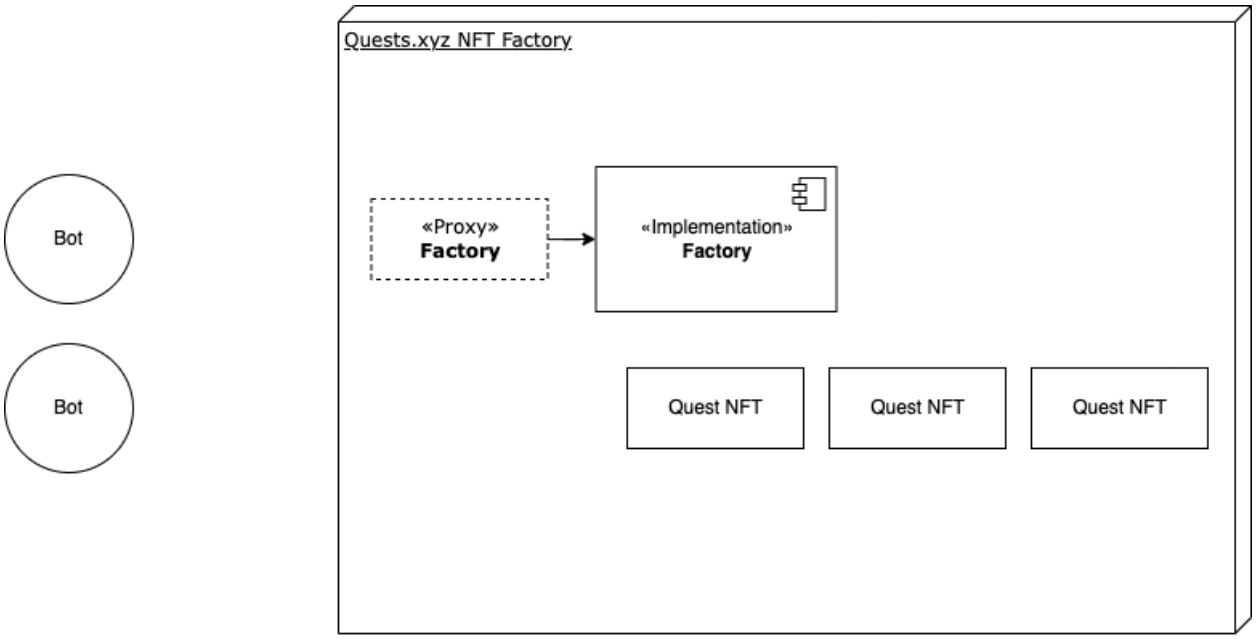
The system of smart contracts described in this specification is not responsible for verifying ownership of a wallet. This is explicitly handled by the web application. Transactions created by end users are relayed to this system through the programmable wallets described above.

Not all quests are represented as NFTs; not all users will have an Ethereum wallet.

While all quests will be stored in the web application database and be visible on a user's profile, not all users will have a wallet or choose to mint an NFT for their quest. This system is only concerned with quests created by users who do have an Ethereum wallet and want to create an NFT to represent them on-chain.

System Architecture

The smart contract system has two main components: the factory contract and the quest NFT contract. Outside systems can interact with both the factory and the quest NFT. Instances of the quest NFT are created by the factory.



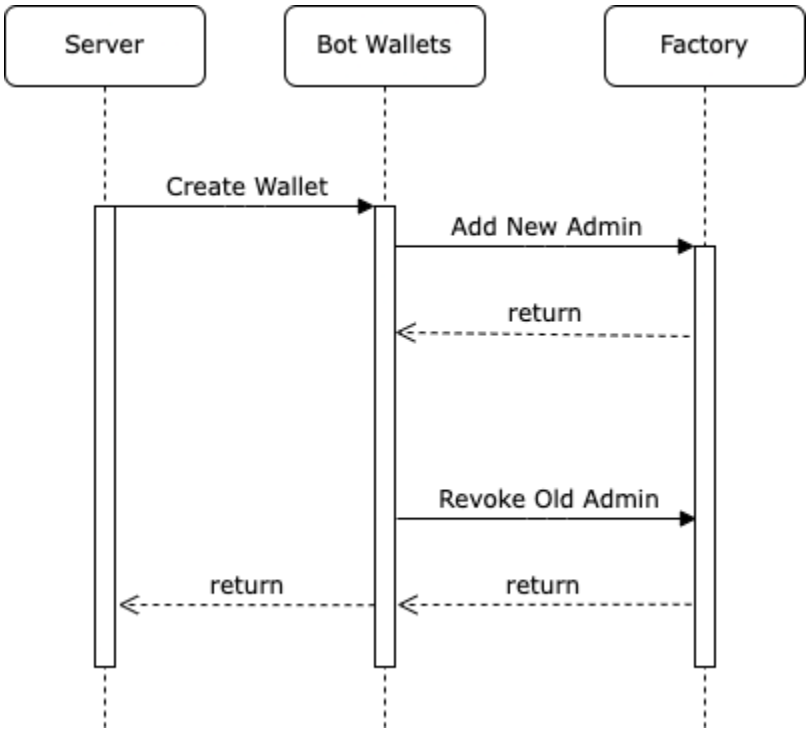
The factory will be implemented with the Universal Upgradeable Proxy Standard (UUPS) pattern described in EIP-1822 and the Proxy Storage pattern described in EIP-1967. That means that the Factory will be made up of two contracts: the proxy and the implementation, the latter of which can be changed by the contract owner. In following this pattern, the business logic related to creating a quest NFT is stored in the current implementation and the data is stored in the Proxy contract. This preserves data as the implementation is upgraded.

The logic for admin roles will be implemented in the Factory. This will control which addresses can invoke the create action in the Factory. It will also control the set of addresses that can make changes to the NFT contracts created by the factory.

Feature Description

The Quest NFT contract system, described above, will need to support a couple of key features that will interact with different parts of the system. The key features that need to be supported are described below, including how different components of the system interact in fulfilling that feature.

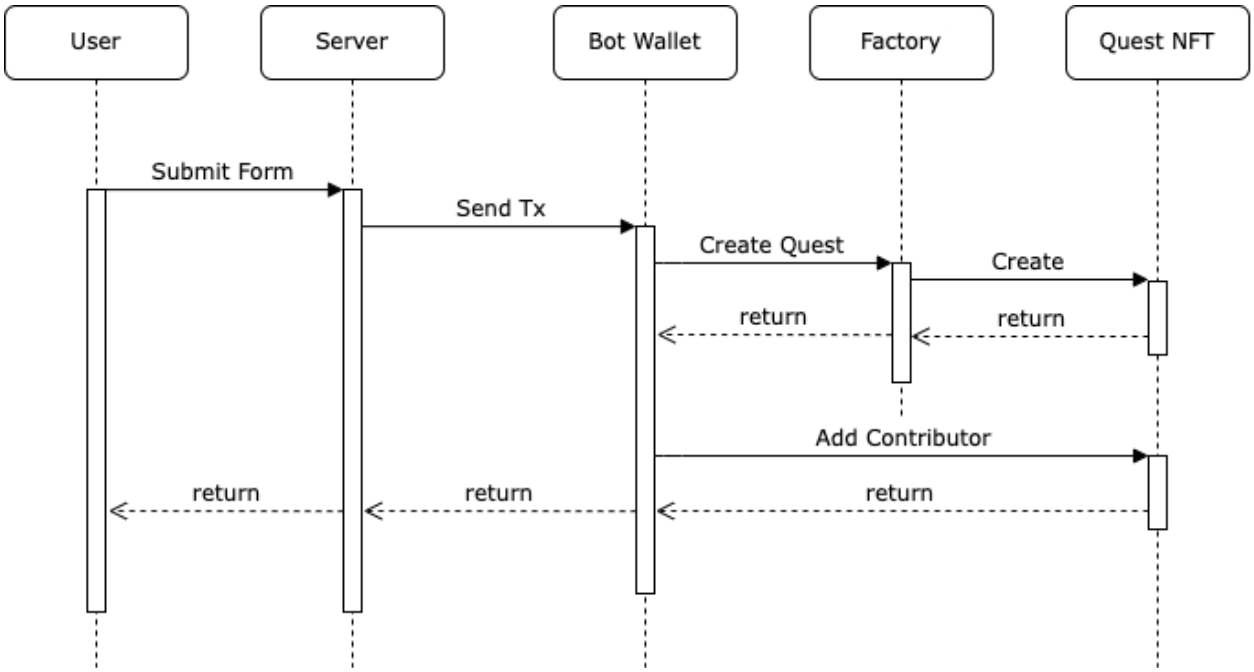
Adding an admin bot wallet



Periodically, the system will cycle an admin wallet so that one is never in use for a long period of time. This will mitigate the risk of the private keys for these programmable bot wallets ever being exposed.

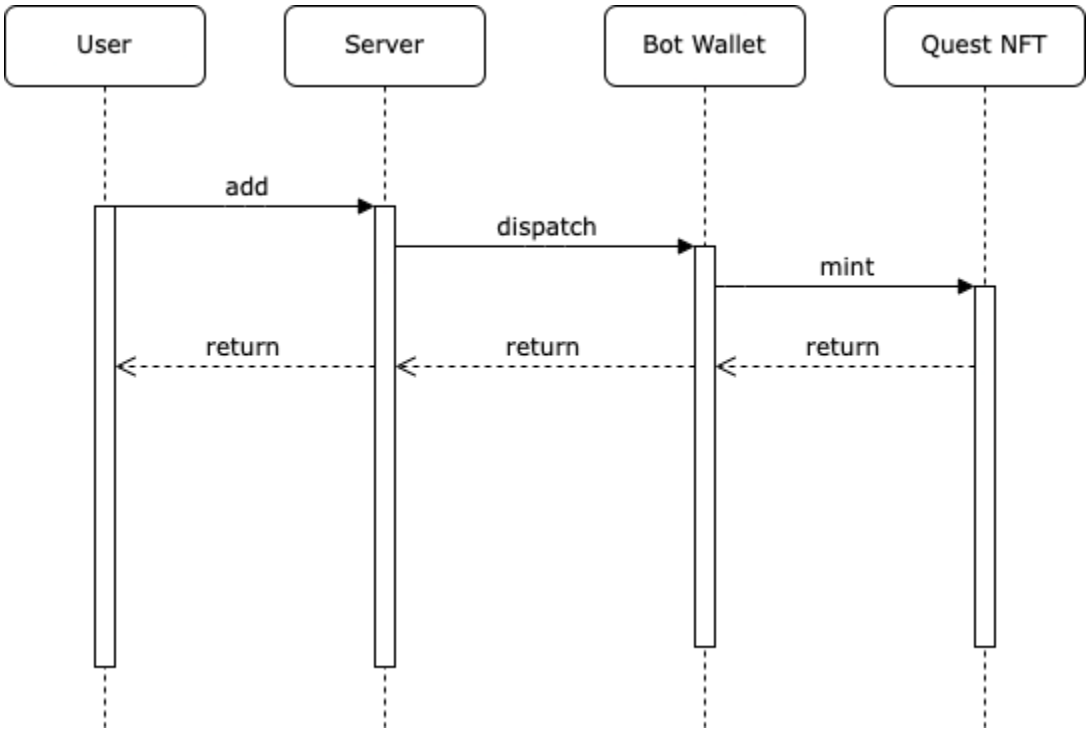
To cycle out an old admin wallet, first a new wallet should be created. The multi-sig Owner should then send a transaction to the Factory invoking the method that adds a wallet address as an admin on the Factory contract. When the transaction is complete, the admin role on the old wallet address should be revoked, also by the multi-sig Owner. If possible, the contract should ensure that there is at least one admin wallet at all times. It should also make it possible to add or remove a list of addresses in one transaction.

Creating a quest



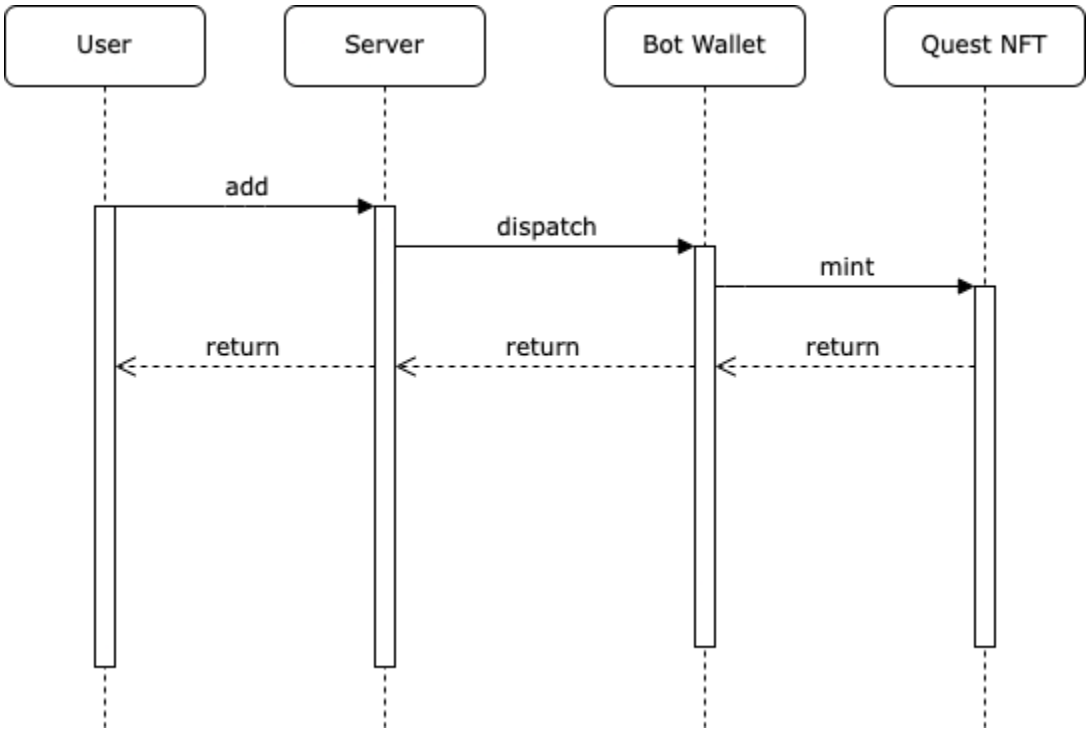
When a user completes a quest on the front-end platform, they'll have the option to create a commemorative Quest NFT. The web application will handle the logic for how this works and will send a message to one of the bot wallets when a Quest NFT should be created. When a bot receives a message to create an NFT, it will send a transaction to the factory to invoke the `createQuest` method, including: the quest name, the tokenURI for the image, the quest symbol, and an array of contributor wallet addresses. The Factory will create an instance of the Quest NFT contract and pass through the data provided.

Adding a contributor



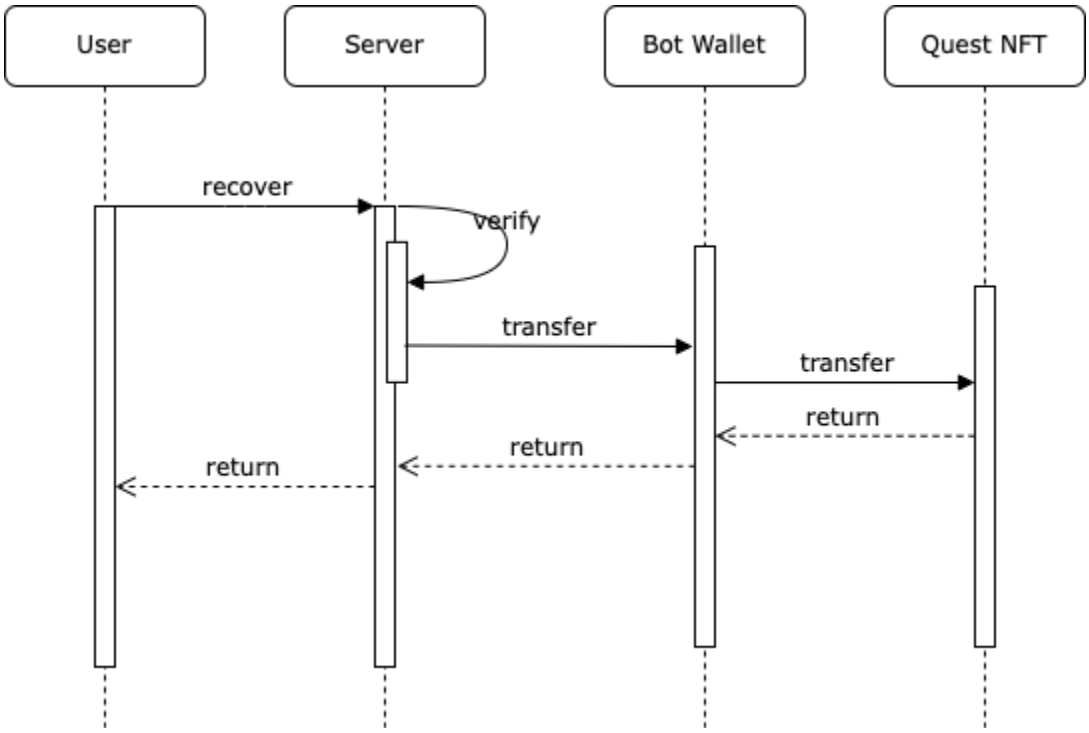
Generally speaking, contributors will be added to quests when the Quest NFT is created. There will be cases where a contributor is added after contract creation. In this case, the web application will send a message to a bot wallet, which will send a transaction to the Quest NFT to invoke the `mint` method with the Ethereum address for the new contributor. The Quest NFT method will verify that the transaction sender is an admin (looking this up on the Factory). If the sender is authorized, a new token ID will be added to the Quest NFT contract and assigned to the supplied address.

Removing a contributor



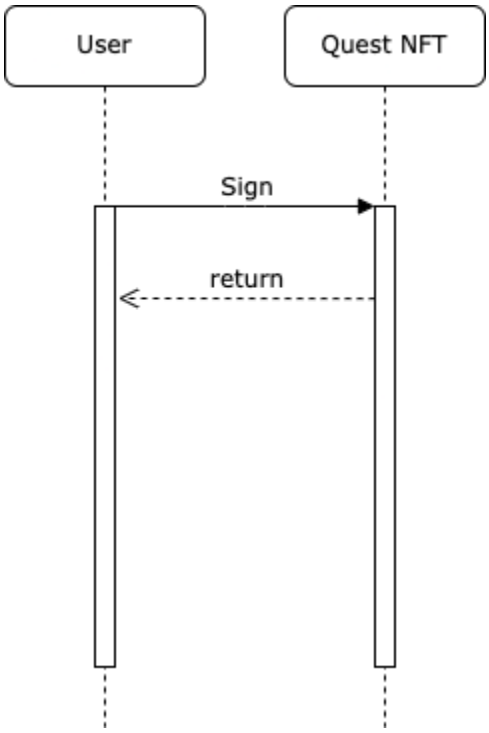
There will be cases where a contributor should be removed from a Quest NFT. The logic for when and why that should happen will be in the web application. When the web application is verified that a user should be removed from a Quest NFT, it will send a message to the bot wallet. The wallet will then send a transaction to the Quest NFT to burn the token associated with the user (the `burn` method).

Recovering Quest NFTs



Users of the Quests platform will need to recover their Quest NFTs, should they lose access to their wallet. The web application will verify that a recovery request is valid and that the new address is associated with the same user. After this verification, the web application will send a message to a bot wallet to transfer a Quest NFT token ID to the new address. If a user has multiple Quest NFTs they need to recover, the bot wallets will need to send one transaction to each NFT contract separately.

Praising (signing) a quest



Users of the Quests platform have the ability to praise a quest (verify that they witnessed and saw the contributions described by the quest). For users who have connected their wallet to their account in Quests, this means signing a message with that wallet with the quest they'd like to praise and an optional message.

Smart Contract Specification

Factory.sol

The Factory contract, responsible for creating instances of the Quest NFT, should implement the methods described below. These will be implemented on top of methods inherited from Ownable2Step and AccessControl, which are required for implementing ownership and administrative roles to the contracts.

```
modifier onlyOwnerOrAdmin()
```

We will need a way to ensure that some methods can only be called by the contract owner (a multi-sig wallet controlled by the Quests.xyz team) or an admin (a bot wallet managed by the application's server). This modifier should build on top of the `_checkOwner()` method provided by Ownable2Step and the `_checkRole()` method provided by AccessControl.

```
function createQuest(string memory _name, string memory _symbol,
string memory _tokenUri, address[] memory _contributors) external
onlyOwnerOrAdmin
```

The Factory contract needs a method that can be called to create a Quest NFT. This method should only be callable by the contract Owner or one of the bot wallets with the Admin role. It should accept and pass through the parameters required by Quest#constructor. Finally, it should store the address of the created Quest and emit an event that can be indexed off-chain.

Quest.sol

The Quest contract, and ERC-721 commemorating the completion of a quest, should implement the methods described below. It should inherit from an ERC-721 implementation, implementing the additional functionality described here.

```
constructor(string memory _name, string memory _symobol, address[]
memory _contributors, string memory _tokenURI) ERC721(_name, _symbol)
```

The Quest contract constructor needs to accept the token name and symbol, as per the ERC721 specification. It should also accept a list of contributors, which it then mints a token to. Finally, it should accept the token URI for the NFT's image (so an image appears in a user's wallet, on OpenSea, and wherever else they might view their NFTs).

```
function mint(address _to) external onlyAdmin
```

The mint method is how contributors are added to the Quest NFT. This method should only be callable by one of the admin wallets. It should create a token (with a token ID) for the address

passed in and emit an event that the contributor was added. This method should revert if a wallet address already holds a token (has already been added as a contributor).

```
function burn(uint256 _id) external onlyAdmin
function burnByAdress(address _from) external onlyAdmin
```

The burn method is how a contributor is removed from the Quest NFT. This method should only be callable by one of the admin wallets. The standard burn implementation is to accept the ID of the token that should be deleted. We want to add a burnByAddress method that takes a user's wallet address, looks up the ID for their token, then calls the standard burn method with that token ID.

```
function transferFrom(address _from, address_to, uint256 _id) public
override onlyAdmin
function safeTransferFrom(address _from, address_to, uint256 _id)
public override onlyAdmin
function safeTransferFrom(address _from, address_to, uint256 _id,
bytes calldata data) public override onlyAdmin
```

The above three methods should follow the standard signature expected from the ERC-721 standard, with one exception: they should only be callable by one of the admin wallets. A user should not be able to transfer their own tokens. Otherwise, the behavior should be the same as that expected in the EIP-721 standard (it should transfer the token with the given id from the _from address to the _to address).

```
function tokenURI(uint256 _id) public view override returns (string
memory)
```

The tokenURI view function should be implemented for returning the token URI for a given token ID. This should behave similarly to other common NFT contracts that use token URIs for storing image locations. It should be compatible with OpenSea, Rainbow, and other platforms for viewing NFTs.

```
function setTokenURI(string memory _newTokenURI) public onlyAdmin
```

The setTokenURI method is what the web application server will use to update the image associated with a Quest NFT, by overwriting the current token URI. It should only be callable by an admin wallet.

Conclusion

An important aspect of Quests.xyz, the platform for creating a portfolio of verified quests, is the ability to create a non-fungible token (NFT) to commemorate the completion of a quest.

This document laid out the technical details that make creating the commemorative Quest NFTs possible. The first section outlined the assumptions made in the design of this system, including the particularly important assumption that most interactions with the system would be by managed bot wallets. The second section then provided an overview of the system architecture, showing how the on-chain part of the system worked internally and how this on-chain component interacts with the off-chain web application. The third section described each core feature of the system and how they interacted with the different components of the system's architecture. Finally, the document laid out a draft API for the two smart contracts that make this system possible.