Blockchain Project
Phase III

Project Details:
Title: AdChain - A Decentralized Digital Advertising Marketplace

Team Members:
Team Member 1: . Ayesha Humaera - ayeshahu@buffalo.edu - 50465149
Team Member 2: Faizuddin Mohammed - faizuddi@buffalo.edu - 50471170

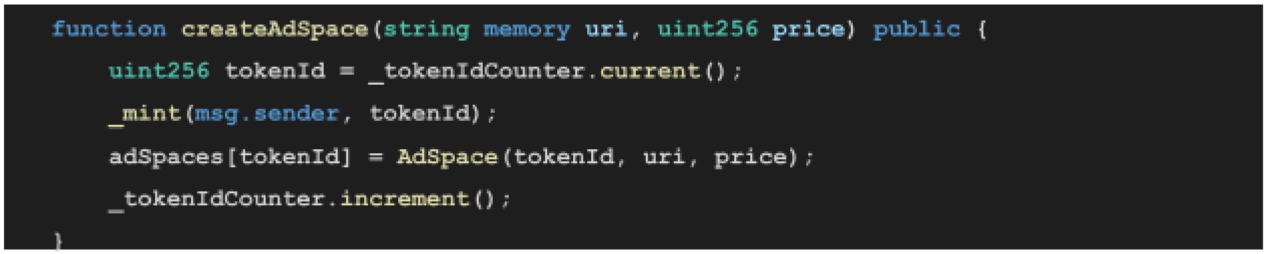
Issue(s) addressed:

AdChain is a blockchain-based platform that aims to address several issues that exist in the traditional model of digital advertising. One major issue is the lack of transparency in the ad buying process, which can lead to inefficiencies and waste. AdChain solves this by representing advertising space as NFTs, which provides a clear and immutable record of ad ownership and display. Another issue is the prevalence of ad fraud and non-human traffic, which can result in wasted ad spend and lower engagement. AdChain helps combat this by using blockchain technology to verify ad impressions and track engagement, making it more difficult for bad actors to manipulate ad metrics. Finally, AdChain could offer more targeted advertising options, based on factors such as user demographics or browsing history, which could increase the effectiveness of digital advertising and help businesses to reach their desired audiences more effectively.

Abstract

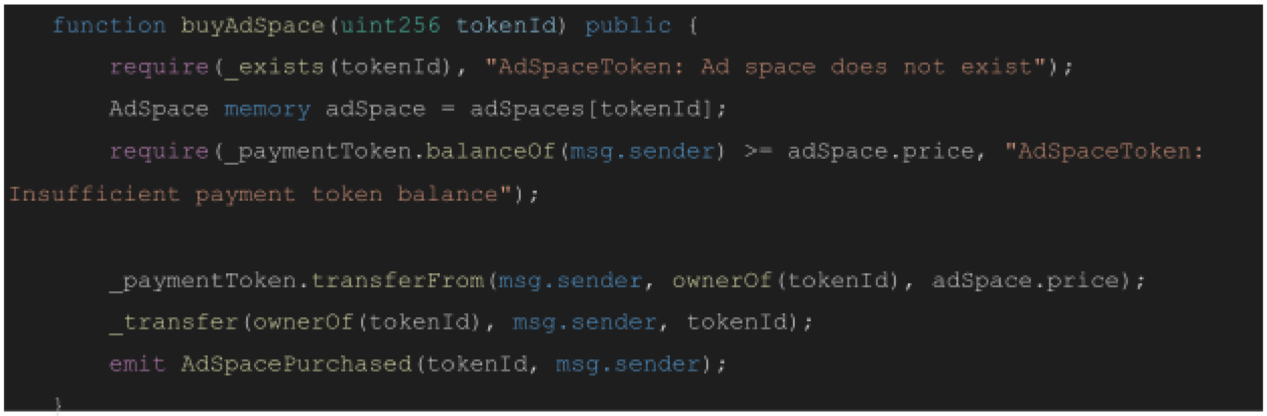
The problem with traditional digital advertising is the lack of transparency, trust, and effectiveness. Advertisers are often unsure about where their ads are displayed and whether their ad impressions are authentic. Ad fraud, click-bots and non-human traffic can cause advertisers to lose money and lower conversion rates. AdChain is a blockchain-based platform that solves these issues by creating a transparent and secure environment where advertisers can buy and sell digital advertising space represented as NFTs. By leveraging blockchain technology, AdChain provides a clear record of ad ownership and display, tracks user engagement, and eliminates the possibility of fraudulent activities. The platform offers a more reliable and efficient way for advertisers to connect with their audiences, something that is not easily achievable through traditional means.

Digital asset and token and the reasoning.



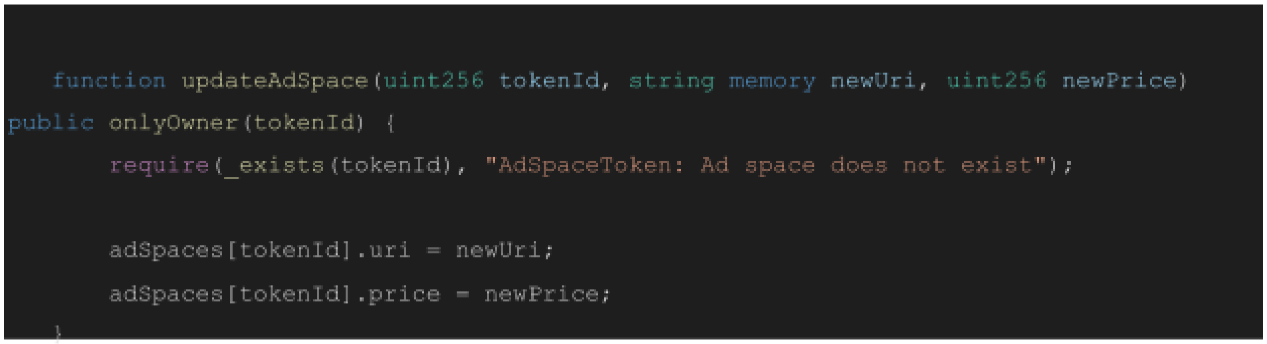
```
  function createAdSpace(string memory uri, uint256 price) public {
      uint256 tokenId = _tokenIdCounter.current();
      _mint(msg.sender, tokenId);
      adSpaces[tokenId] = AdSpace(tokenId, uri, price);
      _tokenIdCounter.increment();
  }
```

Creating an ad space with a URI and price



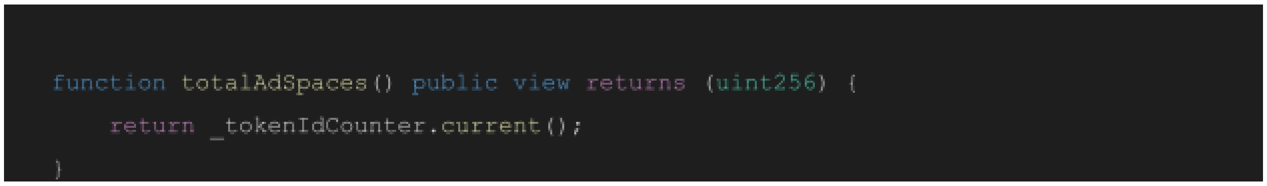
```
  function buyAdSpace(uint256 tokenId) public {
      require(_exists(tokenId), "AdSpaceToken: Ad space does not exist");
      AdSpace memory adSpace = adSpaces[tokenId];
      require(_paymentToken.balanceOf(msg.sender) >= adSpace.price, "AdSpaceToken:
Insufficient payment token balance");

      _paymentToken.transferFrom(msg.sender, ownerOf(tokenId), adSpace.price);
      _transfer(ownerOf(tokenId), msg.sender, tokenId);
      emit AdSpacePurchased(tokenId, msg.sender);
  }
```

Buying an ad space by transferring the required amount of tokens.



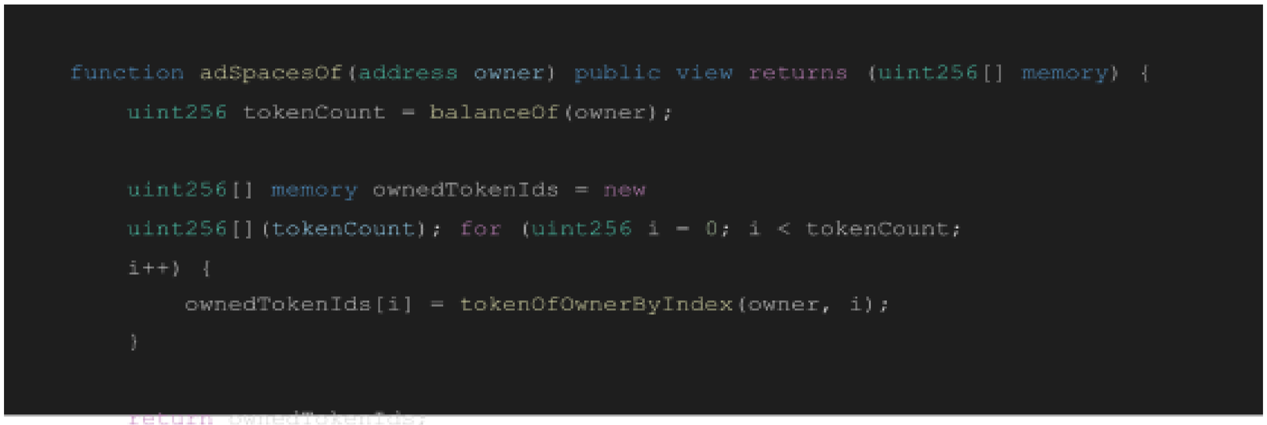
```

  function updateAdSpace(uint256 tokenId, string memory newUri, uint256 newPrice)
public onlyOwner(tokenId) {
      require(_exists(tokenId), "AdSpaceToken: Ad space does not exist");

      adSpaces[tokenId].uri = newUri;
      adSpaces[tokenId].price = newPrice;
  }
```

This function allows the current owner of an ad space to update its URI and price.



```

  function totalAdSpaces() public view returns (uint256) {
      return _tokenIdCounter.current();
  }
```

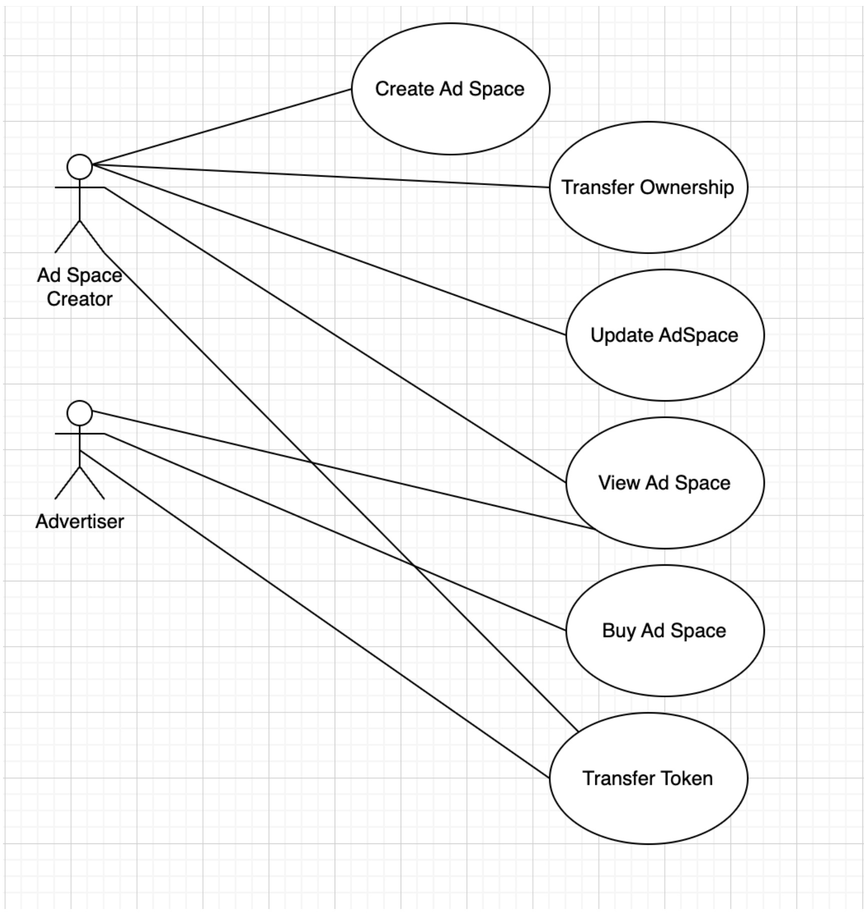
This function returns the total number of ad spaces created.



```
function adSpacesOf(address owner) public view returns (uint256[] memory) {
    uint256 tokenCount = balanceOf(owner);

    uint256[] memory ownedTokenIds = new
    uint256[](tokenCount); for (uint256 i = 0; i < tokenCount;
    i++) {
        ownedTokenIds[i] = tokenOfOwnerByIndex(owner, i);
    }

    return ownedTokenIds;
}
```

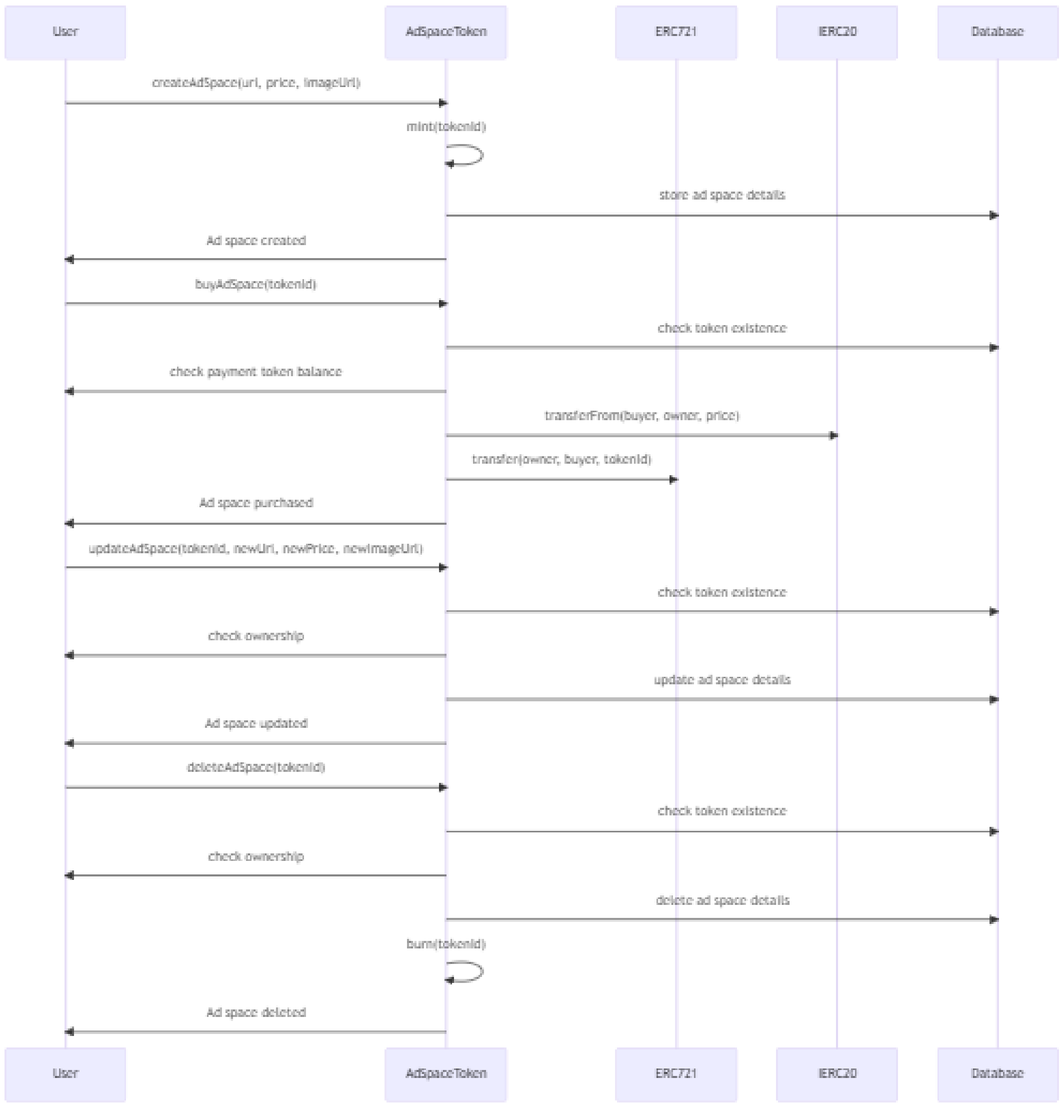
This function returns an array of token IDs owned by a advertiser address.

Use case diagram explaining the project idea



Other Diagrams:

Sequence Diagram:



The sequence diagram shows how a user interacts with the AdSpaceToken contract to create, buy, update, and delete ad spaces.

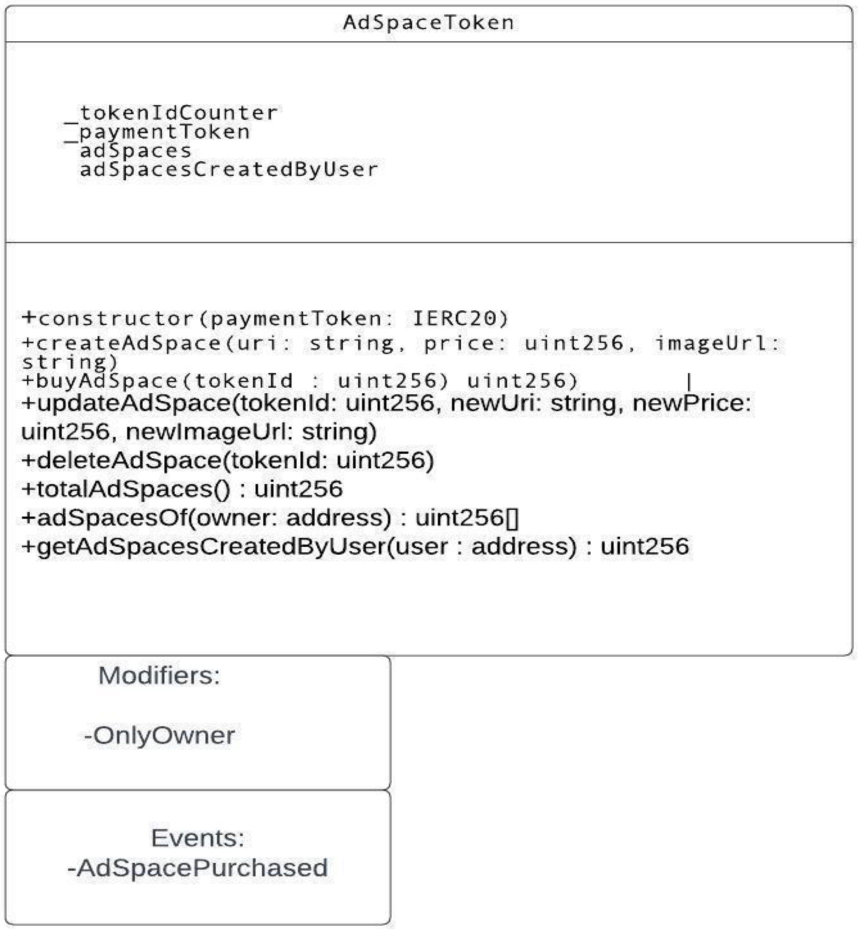
When a user creates a new ad space, they call the createAdSpace function with the URI, price, and image URL parameters. The AdSpaceToken contract mints a new token for the ad space and stores its details. Once the ad space is created, the contract notifies the user.

If a user wants to buy an ad space, they call the buyAdSpace function with the token ID. The contract first checks if the token exists and if the user has enough payment token balance to make the purchase. If the checks pass, the contract transfers the payment tokens from the buyer to the owner and then transfers the ad space token from the owner to the buyer. Once the ad space is purchased, the contract notifies the user.

To update an ad space, a user calls the updateAdSpace function with the token ID and the new URI, price, and image URL parameters. The contract first checks if the token exists and if the user is the owner of the token. If the checks pass, the contract updates the ad space details with the new parameters. Once the ad space is updated, the contract notifies the user.

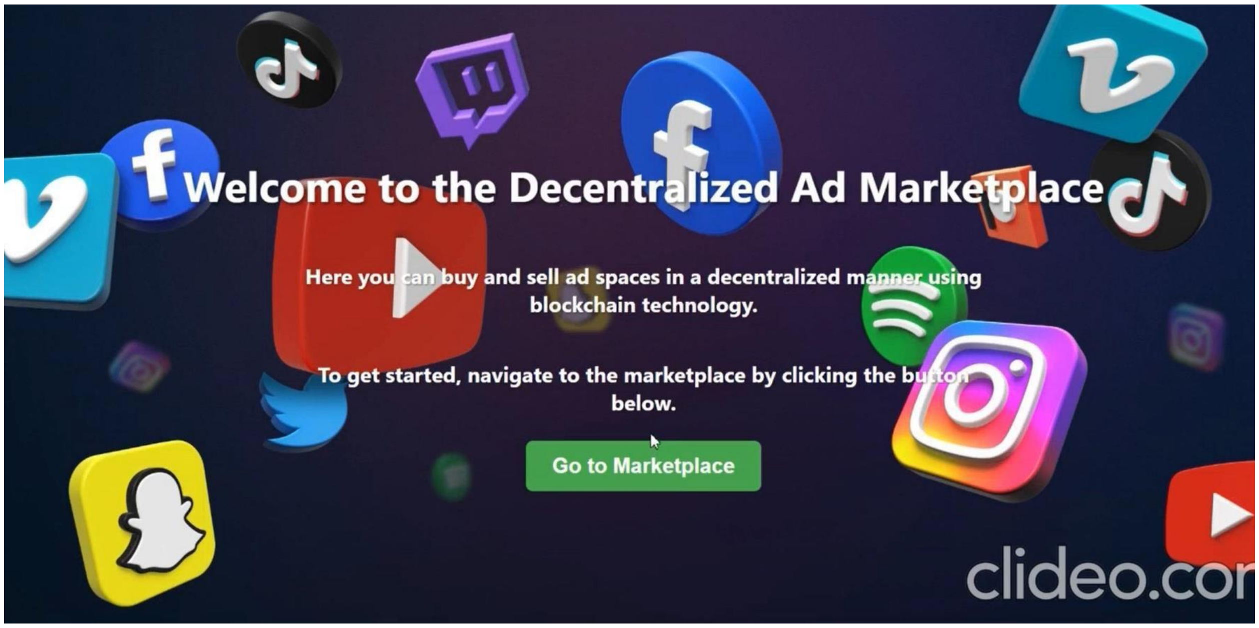
Finally, to delete an ad space, a user calls the deleteAdSpace function with the token ID. The contract first checks if the token exists and if the user is the owner of the token. If the checks pass, the contract deletes the ad space details and burns the ad space token. Once the ad space is deleted, the contract notifies the user.

Contract Diagram:


AdSpaceToken
_tokenIdCounter
_paymentToken
adSpaces
adSpacesCreatedByUser
+constructor(paymentToken: IERC20)
+createAdSpace(uri: string, price: uint256, imageUrl: string)
+buyAdSpace(tokenId : uint256) uint256)
+updateAdSpace(tokenId: uint256, newUri: string, newPrice: uint256, newImageUrl: string)
+deleteAdSpace(tokenId: uint256)
+totalAdSpaces() : uint256
+adSpacesOf(owner: address) : uint256[]
+getAdSpacesCreatedByUser(user : address) : uint256
Modifiers:
-OnlyOwner
Events:
-AdSpacePurchased

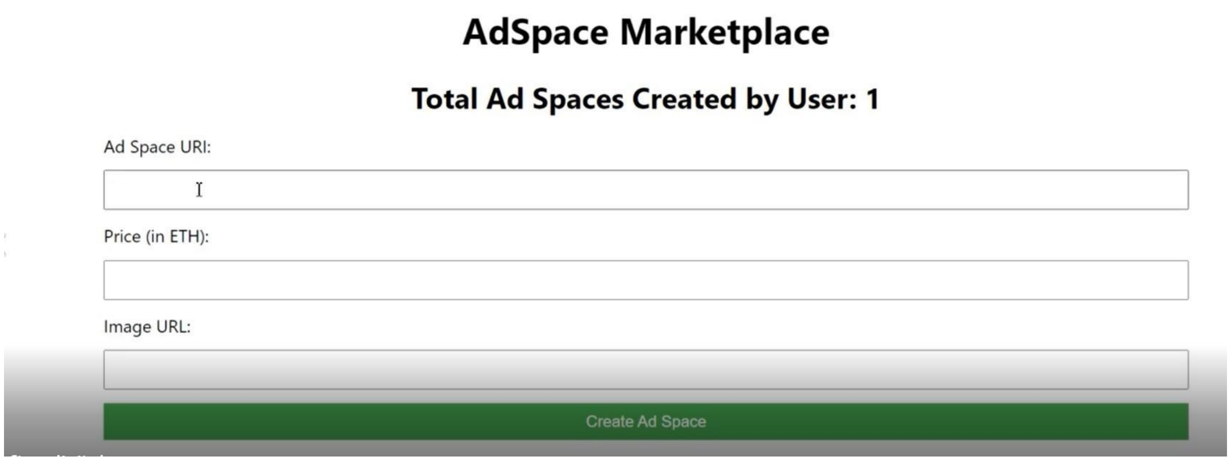
The class diagram represents the AdSpaceToken contract, which is the smart contract responsible for creating, buying, updating, and deleting ad spaces on the blockchain.

UI wireframe or concept figure.

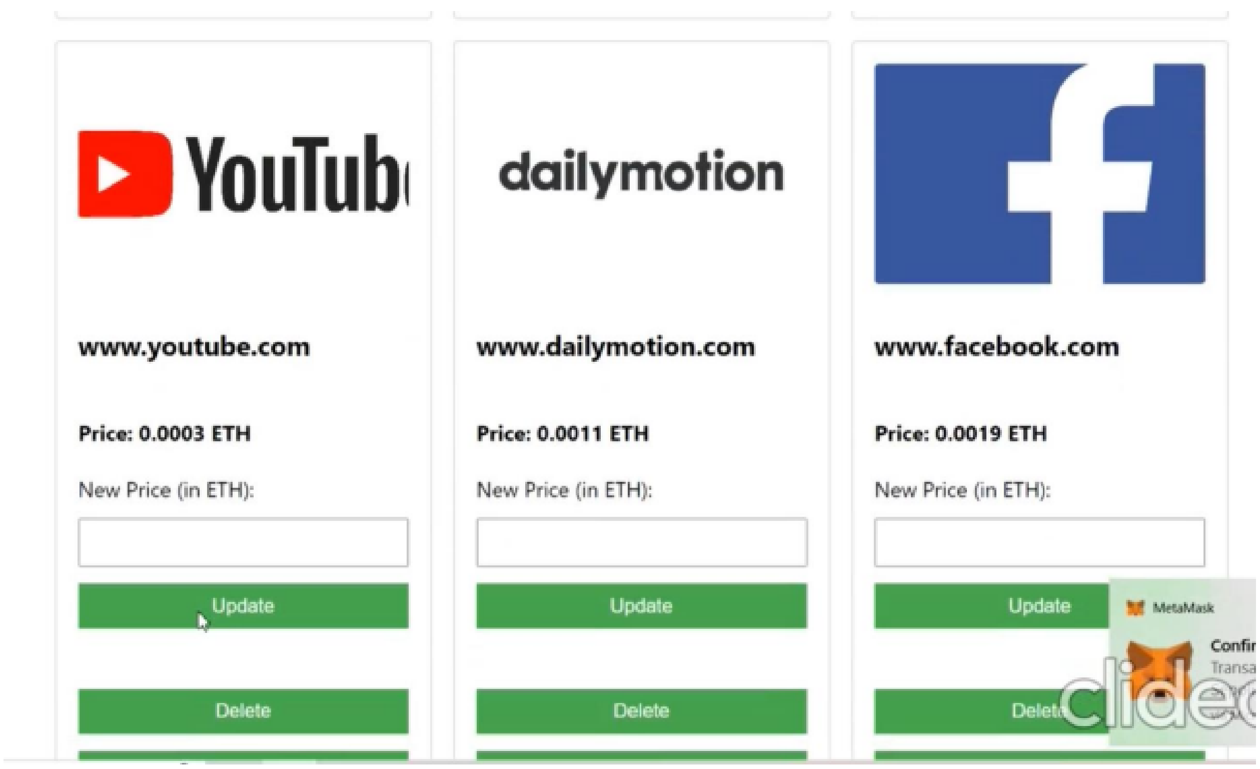

Welcome to the Decentralized Ad Marketplace

Here you can buy and sell ad spaces in a decentralized manner using blockchain technology.

To get started, navigate to the marketplace by clicking the button below.


AdSpace Marketplace

Total Ad Spaces Created by User: 1


www.youtube.com

Price: 0.0003 ETH

New Price (in ETH):

www.dailymotion.com

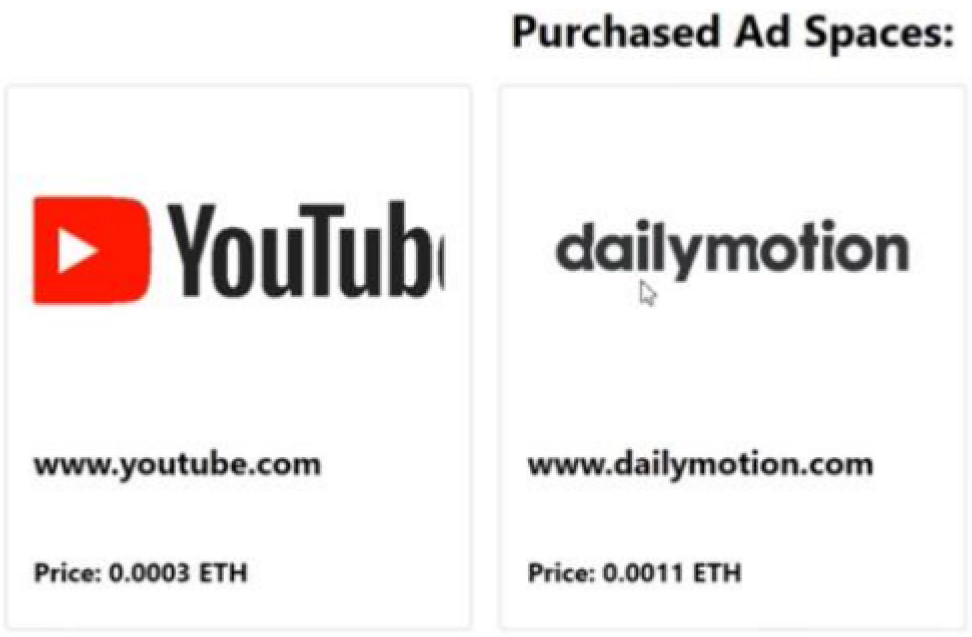
Price: 0.0011 ETH

New Price (in ETH):

www.facebook.com

Price: 0.0019 ETH

New Price (in ETH):


Purchased Ad Spaces:

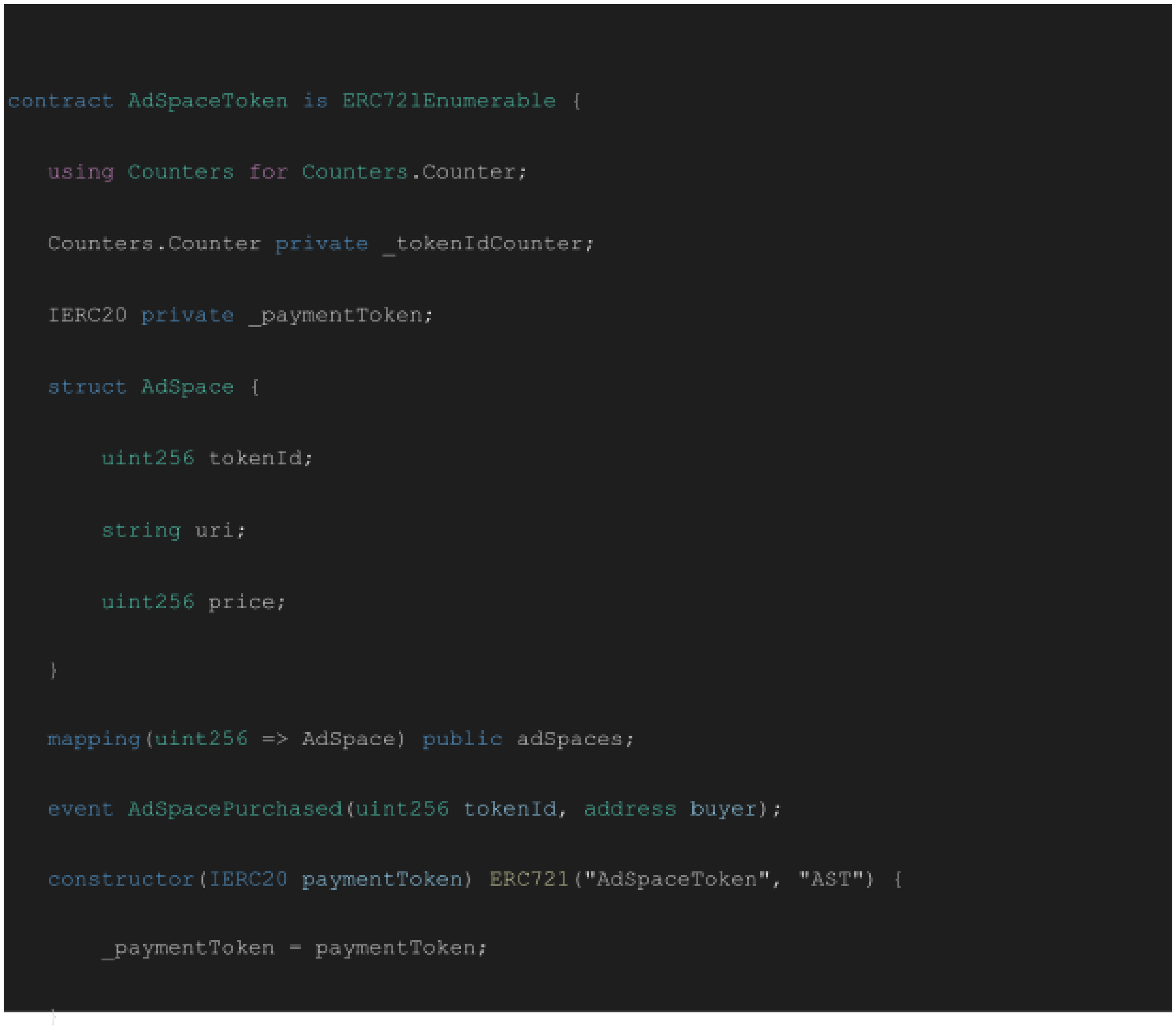
www.youtube.com

Price: 0.0003 ETH

www.dailymotion.com

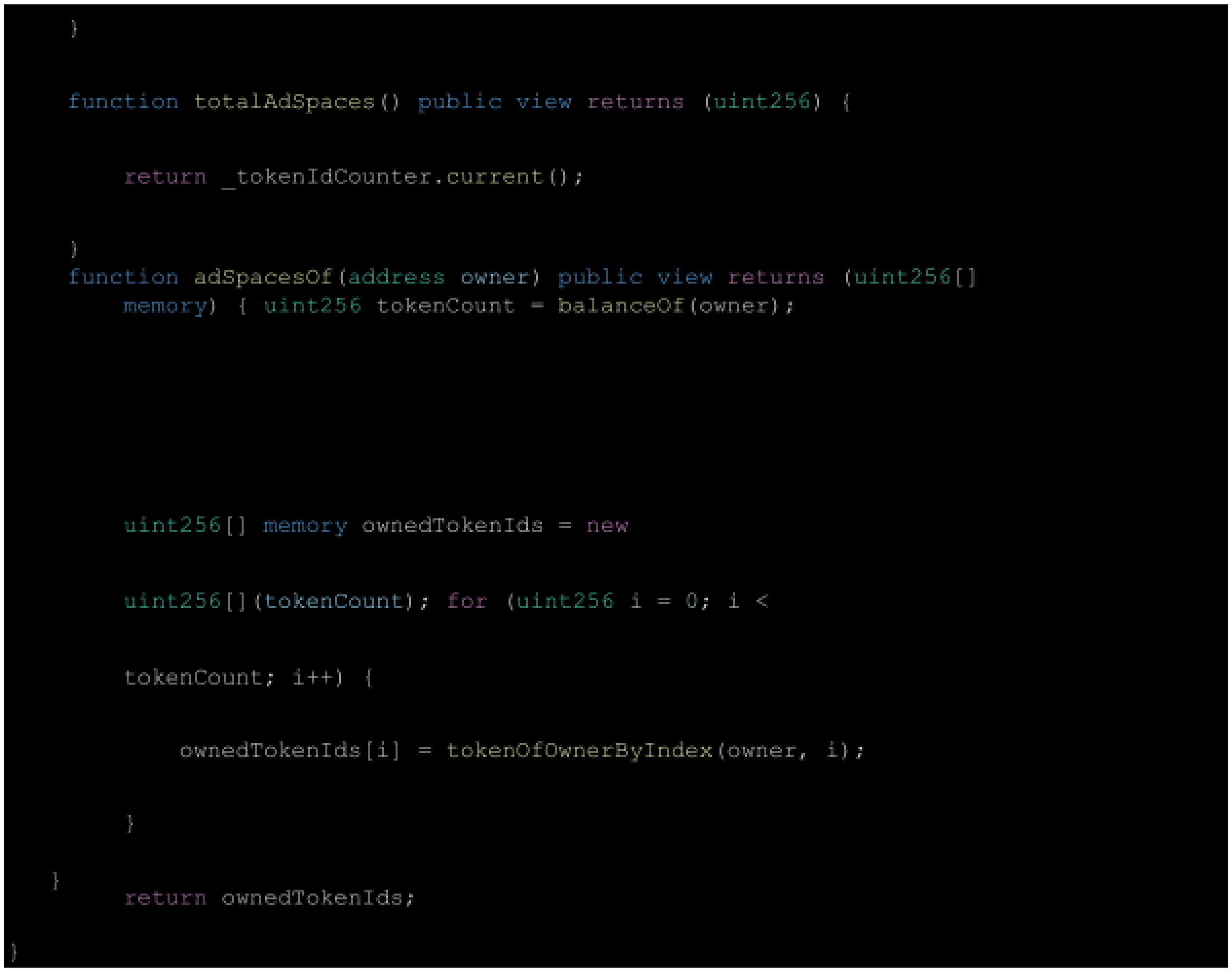
Price: 0.0011 ETH

Code implementation of the digital assets-token smart contracts.



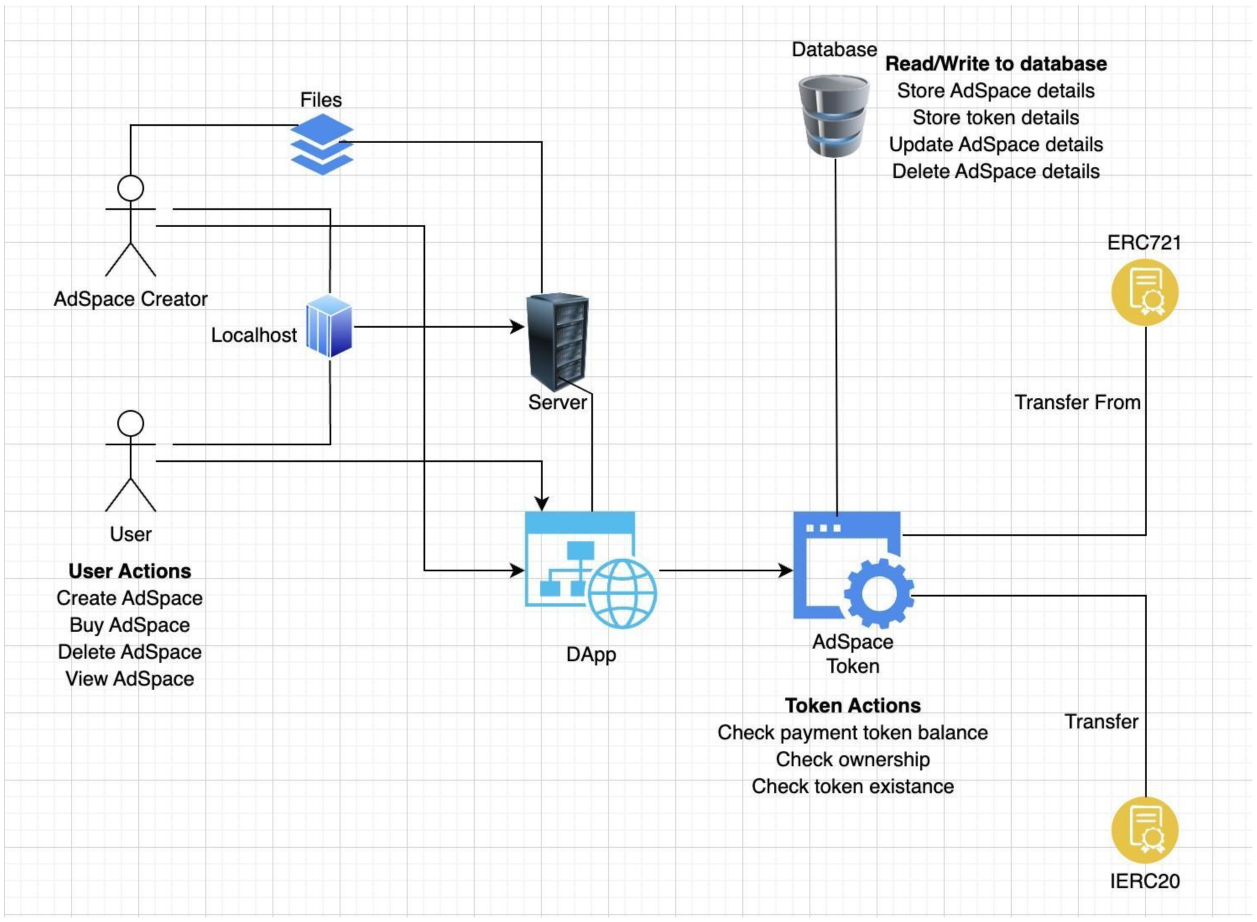
```
contract AdSpaceToken is ERC721Enumerable {

   using Counters for Counters.Counter;

   Counters.Counter private _tokenIdCounter;

   IERC20 private _paymentToken;

   struct AdSpace {

       uint256 tokenId;

       string uri;

       uint256 price;

   }

   mapping(uint256 => AdSpace) public adSpaces;

   event AdSpacePurchased(uint256 tokenId, address buyer);

   constructor(IERC20 paymentToken) ERC721("AdSpaceToken", "AST") {

       _paymentToken = paymentToken;

   }
```

```
    modifier onlyOwner(uint256 tokenId) {
        require(ownerOf(tokenId) == msg.sender, "AdSpaceToken: Caller is not the owner");
        _;
    }
    function createAdSpace(string memory uri, uint256 price) public {
        uint256 tokenId = _tokenIdCounter.current();
        _mint(msg.sender, tokenId);
        adSpaces[tokenId] = AdSpace(tokenId, uri, price);
        _tokenIdCounter.increment();
    }
    function buyAdSpace(uint256 tokenId) public {
        require(_exists(tokenId), "AdSpaceToken: Ad space does not exist");
        AdSpace memory adSpace = adSpaces[tokenId];
        require(_paymentToken.balanceOf(msg.sender) >= adSpace.price, "AdSpaceToken: Insufficient payment token balance");
        _paymentToken.transferFrom(msg.sender, ownerOf(tokenId), adSpace.price);
        _transfer(ownerOf(tokenId), msg.sender, tokenId);
        emit AdSpacePurchased(tokenId, msg.sender);
    }
    function updateAdSpace(uint256 tokenId, string memory newUri, uint256 newPrice) public onlyOwner(tokenId) {
        require(_exists(tokenId), "AdSpaceToken: Ad space does not exist");
        adSpaces[tokenId].uri = newUri;
        adSpaces[tokenId].price = newPrice;
```



```
    }

    function totalAdSpaces() public view returns (uint256) {

        return _tokenIdCounter.current();

    }
    function adSpacesOf(address owner) public view returns (uint256[]
        memory) { uint256 tokenCount = balanceOf(owner);





        uint256[] memory ownedTokenIds = new

        uint256[](tokenCount); for (uint256 i = 0; i <

        tokenCount; i++) {

            ownedTokenIds[i] = tokenOfOwnerByIndex(owner, i);

        }

   }
        return ownedTokenIds;

}
```

Architectural Diagram:



User Actions: Users engage with the app's user interface (UI) to perform various tasks, such as creating a new ad space, updating existing ad space details, or purchasing an available ad space. These actions are captured by the app through different input fields, buttons, and forms.

Connecting to the Blockchain: When a user performs an action, the app communicates with the Ethereum blockchain using the Web3.js library. This library helps the app interact with the blockchain by handling user accounts, encoding and decoding transactions, and managing smart contract interactions.

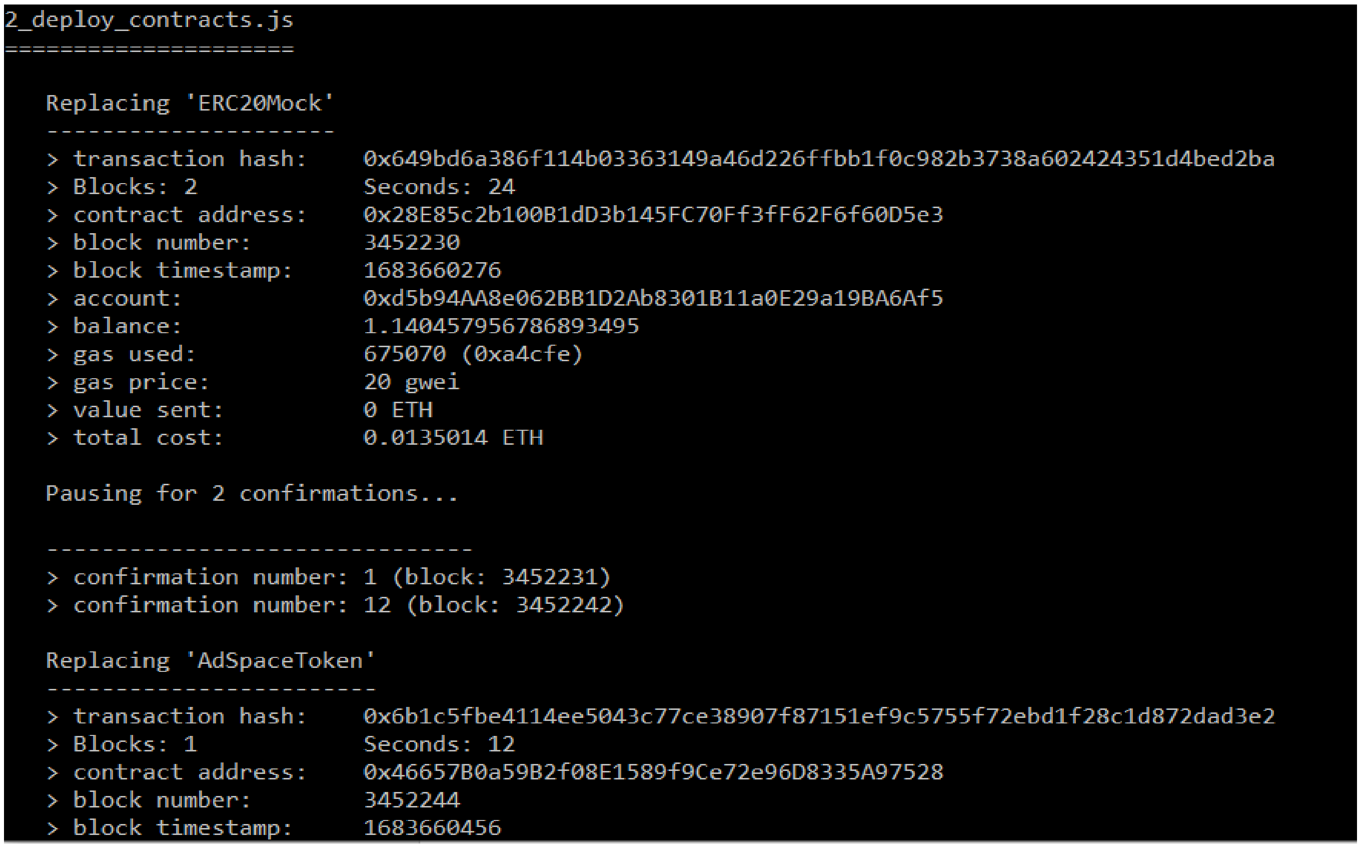
Processing User Actions: The Ethereum blockchain takes care of the user's action by updating the state of the smart contracts (AdSpaceToken and ERC20Mock) accordingly. For instance, when a user creates a new ad space, the AdSpaceToken contract generates a new ad space token with the provided details and updates its records. Similarly, when a user buys an ad space, the ERC20Mock contract manages the transfer of funds between the buyer and the seller.

Keeping the UI Updated: The app listens for events emitted by the smart contracts, such as the creation or purchase of an ad space, to stay informed about changes happening on the Ethereum blockchain. The app can also directly ask the smart contracts for information, like the list of available ad spaces or the total number of ad spaces a user has created. This information is used to refresh the UI and display the most up-to-date data to the user.

Phase 3

For Phase 3 we have deployed our smart contract on infura and hosted our website on cloud. The steps to host the smart contract on infura are as follows:

1) Create a 2_deploy_contracts.js in the migrations folder
2) Update your truffle config file to use the secret key and infura API key from the .env file
3) Compile the smart contract using truffle compile
4) Migrate using truffle migrate –network sepolia
5) Update the contract address in the app.js file and start your application



```
2_deploy_contracts.js
=====================

   Replacing 'ERC20Mock'
   ---------------------
   > transaction hash:    0x649bd6a386f114b03363149a46d226ffbb1f0c982b3738a602424351d4bed2ba
   > Blocks: 2            Seconds: 24
   > contract address:    0x28E85c2b100B1dD3b145FC70Ff3fF62F6f60D5e3
   > block number:        3452230
   > block timestamp:     1683660276
   > account:             0xd5b94AA8e062BB1D2Ab8301B11a0E29a19BA6Af5
   > balance:             1.140457956786893495
   > gas used:            675070 (0xa4cfe)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.0135014 ETH

   Pausing for 2 confirmations...

   -------------------------------
   > confirmation number: 1 (block: 3452231)
   > confirmation number: 12 (block: 3452242)

   Replacing 'AdSpaceToken'
   ------------------------
   > transaction hash:    0x6b1c5fbe4114ee5043c77ce38907f87151ef9c5755f72ebd1f28c1d872dad3e2
   > Blocks: 1            Seconds: 12
   > contract address:    0x46657B0a59B2f08E1589f9Ce72e96D8335A97528
   > block number:        3452244
   > block timestamp:     1683660456
```

The steps to host the website on Google Cloud Platform are as follows:

1) Create a 'app.yaml' file in the root directory of your app and then create a bucket in GCP
2) Upload the 'build' folder and 'app.yaml'
3) Make a new directory using the PowerShell in GCP.
4) Execute the following commands:
   `gsutil rsync -r gs://bucket_name ./directory_name`
   Then enter the directory you just created and execute -
   `gcloud app deploy`
5) The app will run and also display the URL it's running on.

The Website URL is : https://adchain-386221.de.r.appspot.com/

References:

- https://ub.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=b95b95e8-35b3-474e-9422-af9c0146e45b
- Deploy Solidity Smart Contracts with Ganache Personal Blockchain
- Deploy a smart contract to Ethereum using Truffle - A step-by-step guide.
- https://medium.com/coinmonks/5-minute-guide-to-deploying-smart-contracts-with-truffle-and-ropsten-b3e30d5ee1e
- Debug Smart Contracts with Truffle and Ganache (in 2022)
- How to deploy a static React site to Google Cloud Platform | by Denise Ortega | Google Cloud - Community | Medium
- Deploying a React app to Google Cloud. - YouTube

Approved by: Nilkumar Dhamech