SmartChallenge

$SmartChallenge - Report$

The **SmartChallenge** report provides a comprehensive analysis of the SmartChallenge decentralized application (dApp) and its underlying smart contract.

The **dApp** offers a platform for users to solve coding, math, or security challenges and earn rewards, while the smart contract governs the state transitions, manages data, and interacts with the Ethereum network for multiple functionalities.

These include user login using **MetaMask**, **challenge** browsing and submission, score tracking and **leaderboard** display.

The **smart contract** also empowers the contract owner to add new challenges, update player information, and withdraw funds.

In addition, the report delves into the technology used for the development of **SmartChallenge**, **data management strategy**, and details about the smart contract interaction with dApp.

Smart Contract: 0x795aCbf5D411d8a8271b38108CD628F0525FE73a

Technology used to develop SmartChallenge

- `Solidity`: Used for writing the smart contract on the blockchain
- `Next.js`: Next.js simplifies the development of complex user interfaces by offering features like efficient client-side routing, server-side rendering, and hot code reloading.
- `Ether.js`: For interacting with the Sepolia Testnet blockchain
- `Infura`: Provider that provides access to Ethereum nodes via API endpoints. We used it to connect our applications to the Sepolia network without running our own Ethereum node

- `Pinata`: Service to have easy-to-use tools for interacting with the InterPlanetary File System (IPFS). As Infura IPFS service was down😕.

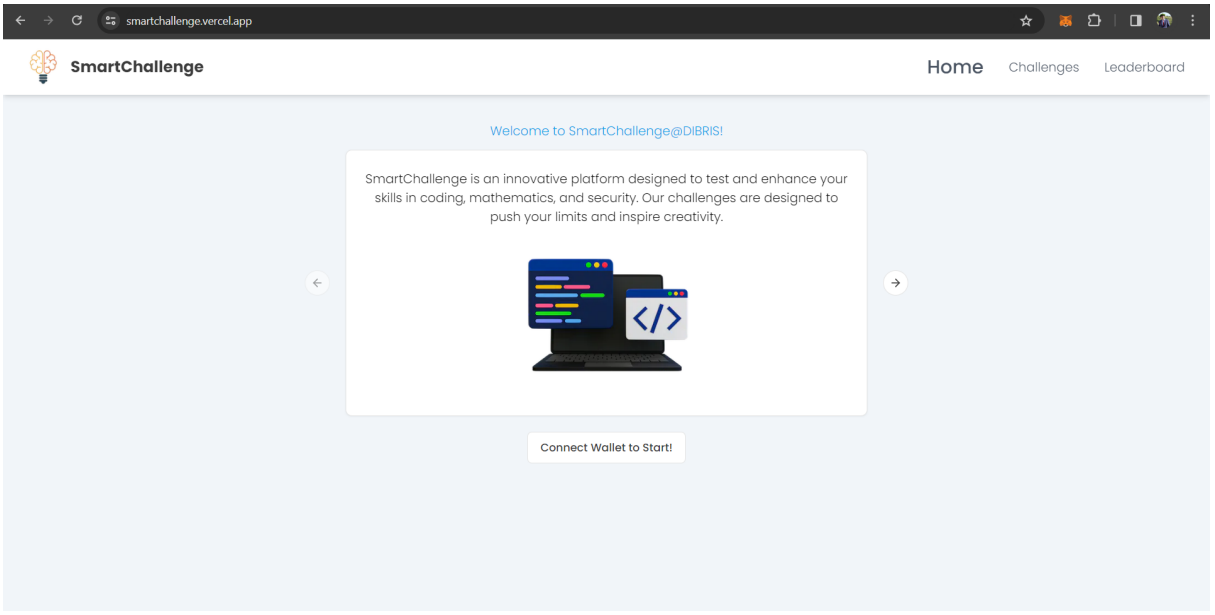
SmartChallenge Decentralized Web Application

The web application, accessible at https://smartchallenge.vercel.app, serves as a platform for users to solve coding, math, or security challenges and earn rewards.

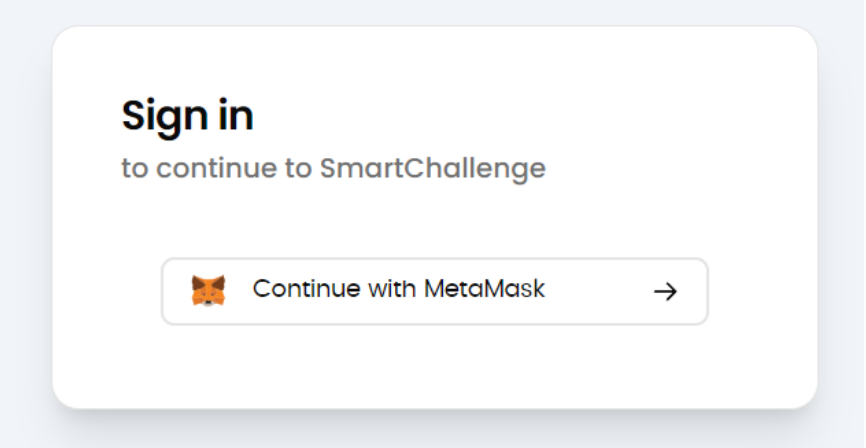
Key features of the web application include:

User login using MetaMask

- *Before login*



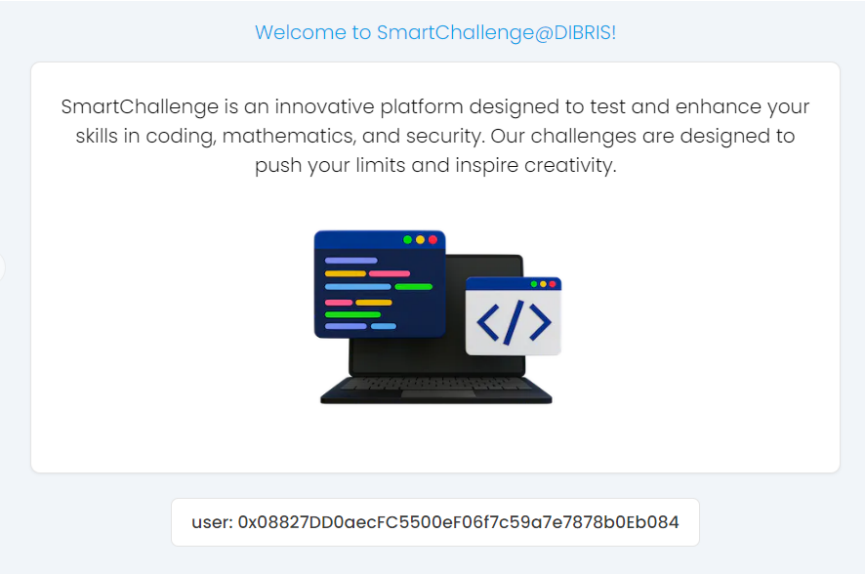
When users click on "Connect Wallet to Start!" or "Challenges" without being logged in, they are redirected to the login page, where they can connect to MetaMask.

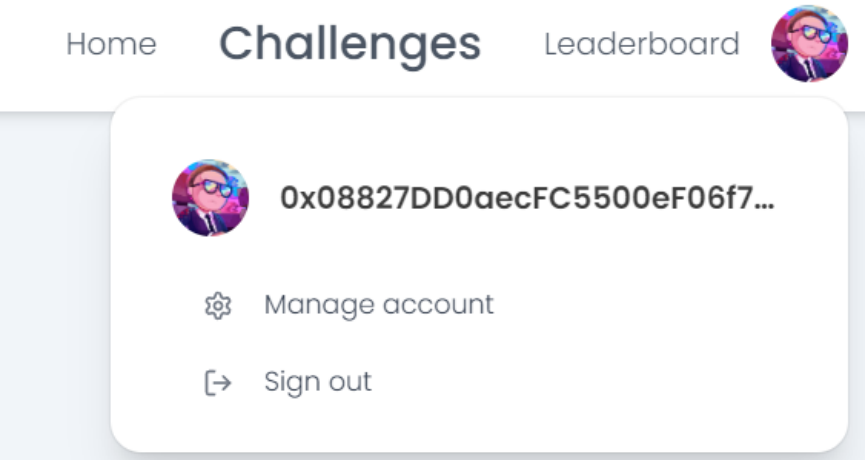



```
const connectMetaMask = async (e: React.MouseEvent) => {
    e.preventDefault();
    let user: any;
    let provider;

    // Check if MetaMask is installed
    if (window.ethereum) {
      provider = new ethers.BrowserProvider(window.ethereum);
      user = await provider.getSigner();
    } else {
      // If MetaMask is not installed, use the default provid
      toast({
        title: "MetaMask not installed. Using read-only defau
        description: "Install MetaMask to continue",
        duration: 2000,
      });
      user = ethers.getDefaultProvider("mainnet");
    }
    const addr = await user.getAddress();
    setUserAddress(addr);
    await set_cookie(addr);
  };
```

- *After login*

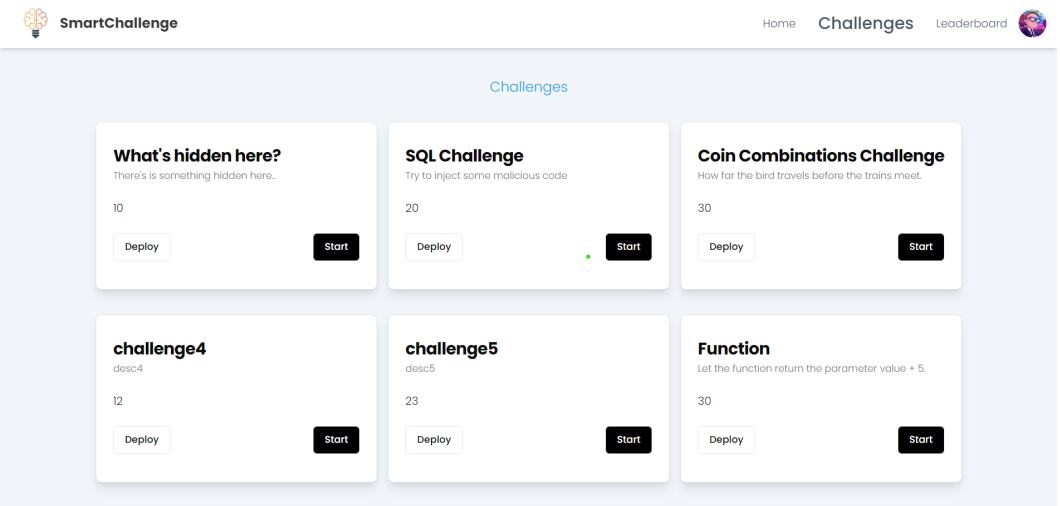


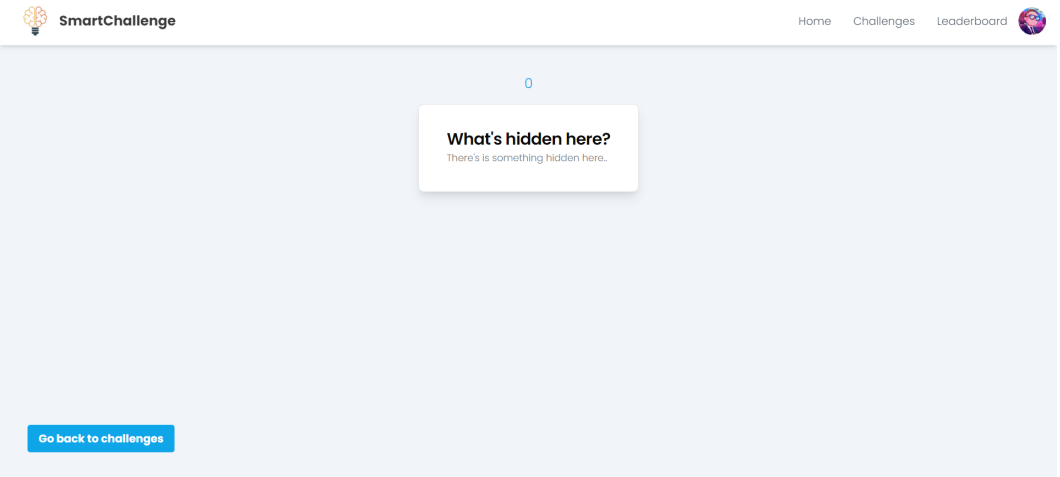


We can now have access to the display of the user tab and the user address also in the home page. In this way users can easily find the information about the EOA connected to the smartContract.

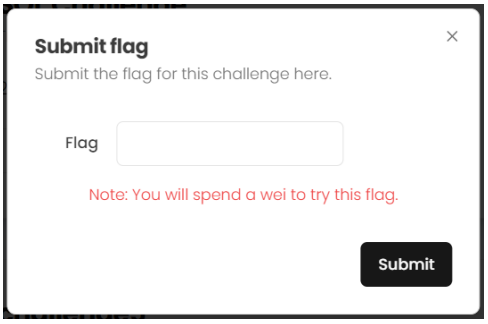
Challenge browsing and submission

- *If you're a "simple user" you can discover and play a challenge and submit the flag for a particular challenge.*

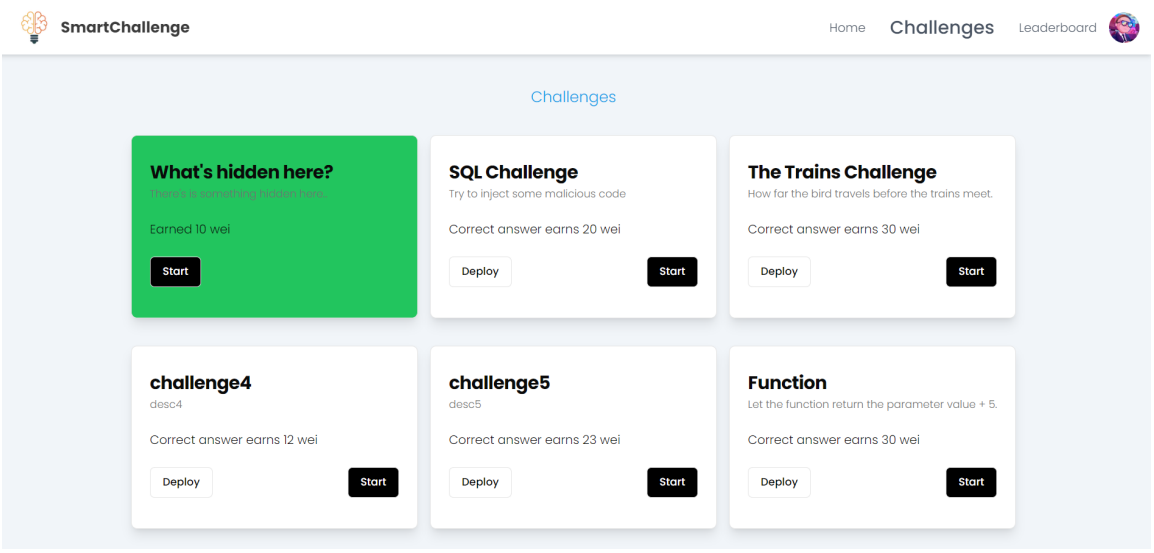




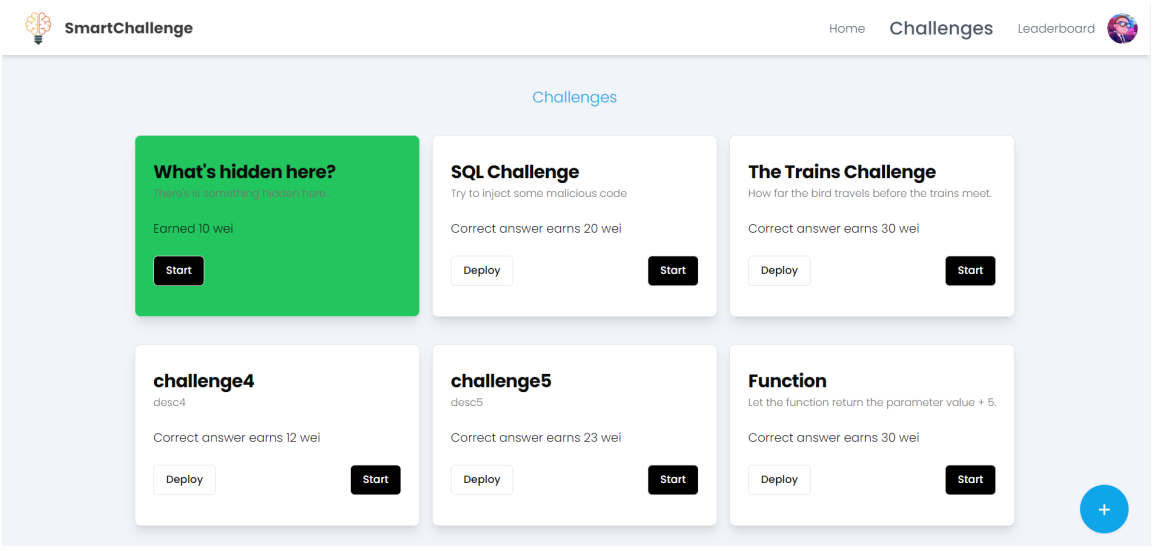
- *When the user submits a flag, **1 wei** will be deducted and transferred to the smart contract. This amount will be withdrawable by the owner.*



- *If the player submits the correct answer, they will be **rewarded** with the **score and wei** specified on the card. As specified in the smart contract the played will not be able to submit again for the challenges already solved.*



- *If you're the owner of the contract you can also add new challenges into the platform.*



Score tracking and Leaderboard

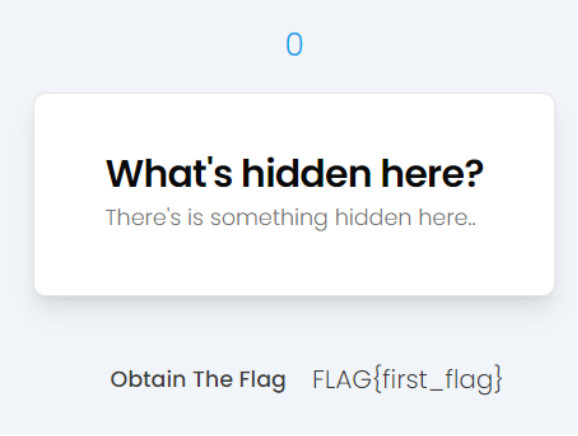
Upon successful completion and submission of the right flag, users' achievements are proudly displayed on the Leaderboard. All state transitions are governed by the smart contract.

Leaderboard

Pos	Player	Score
1	0x0085f20CC2e3A705386D07D8ced8e0E8a65A3330	**83**
2	0x08827DD0aecFC5500eF06f7c59a7e7878b0Eb084	**53**
3	0xeF7dd23FE8e8fF730031C342EF9D030e8b2Dd634	**10**

Data Management Strategy

- Storing data on-chain is expensive, so we store only the data we strictly need on the contract. Also, it's public so information like flag and solution cannot be saved on it.
- To ensure that the flag for each mission remains hidden, we store it in a hashed format in the contract.
- We store information about our missions and the profile image of the user on IPFS, including the name, description and category.
- Store the flag and the solution in clear text in our authenticated API we use as a backend retrieval for those hidden informations. We must keep this information secret to provide the flag to the user in clear text when they complete the challenge.



Smart Contract

The underlying smart contract, named `SmartChallenge`, is written in Solidity. It defines two main structures and several functions.

Interacting with the smart contract in SmartChallenge dApp

Retrieve the contract using ethers

This code snippet creates an instance of Ethereum's provider, Infura, using the Ethereum network and Infura API key specified in the environment variables. The Infura provider is then used to create an instance of the smart contract located at the address specified by `CONTRACT_ADDRESS`, using the ABI specified by `abi`. This instance of the smart contract can be used to interact with the contract's functions.

- *in server components*

```
const provider = new ethers.InfuraProvider(
  process.env.ETHEREUM_NETWORK,
  process.env.INFURA_API_KEY
);

const contract = new ethers.Contract(CONTRACT_ADDRESS, a
bi, provider);
```

- *in client components*

```
let provider = new ethers.BrowserProvider(window.ethereu
m);
let contract = new ethers.Contract(CONTRACT_ADDRESS, ab
i, provider);
```

Connect with the IPFS gateway to retrieve information stored in IPFS

Use one of the following gateways to access the IPFS:

```
https://turquoise-neighbouring-nightingale-673.mypinata.cloud
https://gateway.pinata.cloud/ipfs/
```

Structures

- `Challenge`: Stores information about a challenge, including its ID, flag hash, reward, and hash.
- `Player`: Stores information about a player, including their registration status, score, solved challenges mapping, and hash.

```
struct Challenge {
    uint challengeId;
    string flagHash;
    uint8 reward;
    string hash;
}

struct Player {
    bool isRegistered;
    uint256 score;
    mapping(uint => bool) solvedChallenges;
    string hash;
}
```

Mappings and Variables

- `challenges`: This mapping links a challenge ID to a `Challenge` structure.
- `players`: This mapping links an address to a `Player` structure.
- `playerAddresses`: This is an array of addresses of all players.
- `owner`: This is the address of the contract owner.
- `costFlagSend`: This is the cost to submit a flag, currently set to 1 wei.

- `challengeCounter`: This is a counter for the challenges added to the contract.

```
mapping(uint => Challenge) public challenges;
mapping(address => Player) public players;
address[] public playerAddresses;
address payable owner;
uint8 public constant costFlagSend = 1; // wei
uint32 public challengeCounter;
```

Events

- `ChallengeSubmitted`: This event is emitted when a flag is submitted.
  - We utilize this event in our decentralized application (dApp) to receive updates from the contract about the validity of the solution submitted by the user.
    - Upon user submission:
      - The user pays for the gas and an additional 1 wei to deter spamming.
      - If the flag stored in the smart contract matches the submitted flag, the user receives a reward.
    - *This is what we used to listen the event from SmartChallenge:*

```
SignedContract.on("ChallengeSubmitted", (value: st
ring) => {
    toast({
        title: "Challenge submitted",
        description: value,
        duration: 2000,
    });
});
```

- `ChallengeAdded`: This event is emitted when a new challenge is added.
  - We don't use this event because we simply refresh the page to view the added challenge, provided the owner's transaction for the challenge creation doesn't fail.

Functions

- `getOwner`: Returns the owner of the contract.

```
function getOwner() public view returns (address) {
        return owner;
}
```

- `payUser`: Internal function to transfer a certain amount of wei to a recipient when the submission of a challenge is done correctly:

```
function payUser(address payable recipient, uint8 amount)
    require(address(this).balance >= amount, "Insufficient
    recipient.transfer(amount);
}
```

- `submitFlag`: This function allows users to submit a flag for a challenge. If the flag is correct, the user earns the reward and their score increases. This is the function that triggers the event monitored by the dApp, as previously explained:

```
function submitFlag(uint _challengeId, string memory _fl
ag) public payable {
    require(challenges[_challengeId].challengeId == _cha
llengeId,
                        "Challenge does not exist");
    require(bytes(_flag).length > 0,
                        "Flag cannot be empty");
    require(!players[msg.sender].solvedChallenges[_chall
engeId],
                        "Challenge already solved by thi
s player");
    require (msg.value >= costFlagSend,
                         "Not enough wei to submit");

    if(keccak256(abi.encodePacked(challenges[_challengeI
d].flagHash)) ==
              keccak256(abi.encodePacked(_flag))) {
        emit ChallengeSubmitted("Correct answer!");
```

```
        players[msg.sender].solvedChallenges[_challengeId] = true;
        players[msg.sender].score += challenges[_challengeId].reward;
        payUser(payable (msg.sender), challenges[_challengeId].reward);

        if (!players[msg.sender].isRegistered) {
            players[msg.sender].isRegistered = true;
            playerAddresses.push(msg.sender);
        }
    } else {
        emit ChallengeSubmitted("Incorrect Flag!");
    }
}
```

- `addChallenge`: Allows the owner to add new challenges to the contract.

```
function addChallenge(string memory _flag, uint8 _reward, string memory _hash)
public onlyOwner {
    require(bytes(_flag).length > 0, "Flag cannot be empty");
    challenges[challengeCounter]=Challenge(challengeCounter, _flag, _reward, _hash);
    challengeCounter++;
    emit ChallengeAdded(challengeCounter, _flag, _reward, _hash);
}
```

- `updatePlayer`: This function allows players to update their hash. It's used to change the profile image of the user, which is stored on IPFS. Essentially, this method is used to update the hash associated with the user's profile page to a more recent one.
    - *in the smartContract*

```
function updatePlayer(string memory _hash) public {
    players[msg.sender].hash = _hash;
```

```
}
```

- *in the dApp SmartChallenge*

```
export async function ModifyProfile(selectedFile: an
y, toast: any) {
  let provider = new ethers.BrowserProvider(window.et
hereum);
  let user = await provider?.getSigner();
  const SignedContract = new ethers.Contract(CONTRACT
_ADDRESS, abi, user);

  const data = new FormData();
  data.append("file", selectedFile);
  data.append("pinataMetadata", JSON.stringify({ nam
e: "pinnie.json" }));
  data.append("pinataOptions", JSON.stringify({ cidVe
rsion: 1 }));

  const options = {
    method: "POST",
    headers: {
      accept: "application/json",
      authorization:
        "Bearer <authorization_bearer>",
    },
    body: data,
  };

  try {
    const response = await (
      await fetch("https://api.pinata.cloud/pinning/p
inFileToIPFS", options)
    ).json();
    console.log(response.IpfsHash);
    try {
      let add = await SignedContract.updatePlayer(res
ponse.IpfsHash);
```

```
      toast({
        title: "Player modified",
        description: "The player has been modified su
ccessfully",
      });
    } catch (error: any) {
      toast({
        title: "Error",
        description: error.message,
        duration: 2000,
      });
    }
  } catch (error) {
    console.error(error);
  }
}
```

- `getPlayer`: Allows players to retrieve their hash representing the image stored on IPFS.

```
function getPlayer() public view returns (string memory)
{
    Player storage player = players[msg.sender];
    return (player.hash);
}
```

- `getChallenges`: Returns all challenges available in the contract. Used in the challenges page to display all the challenges and in the challenge/[id] page to get the challenge with a specific id.
  - *in the smart contract:*

```
function getChallenges() public view returns (Challen
ge[] memory) {
    Challenge[] memory allChallenges = new Challenge
[](challengeCounter);

    for (uint i = 0; i < challengeCounter; i++) {
        allChallenges[i] = challenges[i];
```

```
    }

    return allChallenges;
}
```

- *retrieve the content in the dApp(in the challenges page):*

```
async function getChallenge() {
  const challenges = await contract.getChallenges();
  const pinata
            = "<https://turquoise-neighbouring-nighti
ngale-673.mypinata.cloud/ipfs/>";

  try {
    const challengeObject = await Promise.all(
      challenges.map(async (challenge: any[]) => {
        const cid = pinata + challenge[3];
        const res = await (await fetch(cid, { cache:
"no-store" })).json();

        return {
          key: Number(challenge[0]),
          reward: Number(challenge[2]),
          name: res.name,
          description: res.description,
          category: res.category,
        };
      })
    );

    return challengeObject as any[];

  } catch (error) {
    console.log(error);
  }
}
```

- `getScore`: Returns the score of a player.

```
function getScore(address _player) public view returns
(uint) {
    return players[_player].score;
}
```

- `getPlayers`: Returns a list of all player addresses.

  Here is the formatted version of the selected text:

  ```
  function getPlayers() public view returns (address[] mem
  ory) {
      return playerAddresses;
  }
  ```

- `isChallengeSolved`: Checks if a player has solved a specific challenge.
  - *in the smartContract:*

    ```
    function isChallengeSolved(address playerAddress, uin
    t challengeIndex)
    public view returns (bool) {
          return players[playerAddress].solvedChallenges
    [challengeIndex];
    }
    ```

  - *in the dApp used to set a specific challenge in green representing the status → Solved.*

    ```
    ...
    {
      challenges?.map(async (challenge) => {
        let solved = await contract.getSolvedChallenge(us
    erAddress, challenge.key);
        return (
          <Challenge
            key={challenge.key}
            isSolved={solved}
            challenge={challenge}
          />
    ```

```
    );
  })
}
...
```

- `getScores`: Returns a list of all player addresses and their corresponding scores. Used to get the score to be displayed correctly in the leaderboard
  - *in smart contract:*

```
function getScores() public view returns (address[] m
emory, uint[] memory) {
    uint[] memory playerScores = new uint[](playerAdd
resses.length);
    for (uint i = 0; i < playerAddresses.length; i++)
{
        playerScores[i] = players[playerAddresses
[i]].score;
    }
    return (playerAddresses, playerScores);
}
```

  - *in dApp:*

```
const Leaderboard = () => {
  ...
  useEffect(() => {
    const fetchScores = async () => {
            ...
      let [addresses, scores] = await contract.getSco
res();

      // Create an array of objects, each object cont
ains an address and a score
      let players = addresses.map((address: string, i
ndex: number) => ({
        address,
        score: scores[index].toString(),
      }));
```

```
        ...
    };
...
}
```

- `withdraw`: Allows the owner to withdraw funds from the contract.

```
// Function to withdraw money from this contract
function withdraw() public payable onlyOwner {
    // get the amount of Ether stored in this contract
    uint256 balance = address(this).balance;
    require (balance > 0, "The balance is zero, nothing
to transfer");

    owner.transfer(balance);
}
```

A Concluding Note

The `SmartChallenge` smart contract, submitted for verification on Etherscan on February 18, 2024, empowers the contract owner to add new challenges, update player information, and withdraw funds.
This report is an illuminating guide for understanding and further enhancing the SmartChallenge smart contract.