Project Overview: ETHPool

ETHPool is a decentralized application (dApp) on the Ethereum blockchain, utilizing a smart contract to facilitate a pooled investment and reward distribution system. This platform enables participants to deposit Ether (ETH) into a collective pool and earn rewards over time, which are sourced from funds periodically added by the ETHPool team.

Initial Project Setup

Creating a New Project in Visual Studio Code (VSC) and cloning the Repository into the Project Folder:

- Open Visual Studio Code.
- Create a new project folder by selecting `File` > `New Folder`, and choose a location for your project.
- Open the newly created project folder in VSC by selecting `File` > `Open Folder`.
- Open the terminal in VSC by selecting `Terminal` > `New Terminal`.
- Navigate to the root of your project folder if not already there.
- Clone the repository directly into your project folder using the Git command:

```
git clone https://github.com/exactly/solidity-challenge .
```

Note: The period at the end of the command clones the repository into the current directory without creating an additional nested folder.

In the VSC terminal, ensure you are in the project root directory.

Install Hardhat by running: `npm install --save-dev hardhat@2.19.3`

Install all needed dependencies:

```
npm install dotenv@16.4.5 ethers@6.11.1
find-config@1.0.0 --save-dev
```

Create a Hardhat-Project: `npx hardhat`

Here we choose the following settings:

```
√ What do you want to do? · Create a JavaScript project

√ Hardhat project root: · (Desired root path of the project)

√ Do you want to add a .gitignore? (Y/n) · y

√ Do you want to install this sample project's dependencies with npm
(@nomicfoundation/hardhat-toolbox)? (Y/n) · y
```

After setting up our project in Visual Studio Code and incorporating the README from the solidity-challenge repository, we're ready to focus on the core task at hand. The README has provided us with a clear set of requirements and objectives for our Smart Contract, which will guide the development of ETHPool. These requirements, outlined in the documentation below, form the blueprint for our next steps in building the platform.

Here are the requirements that need to be met by the smart contract:

ETHPool offers a service where individuals can deposit ETH and receive weekly rewards. Participants must have the ability to withdraw their deposits and their share of rewards at any time. New rewards are manually deposited into the pool by the ETHPool team each week through a contract function.

- Only the team can deposit rewards.
- Deposited rewards are added to the user pool, not to individual users.
- Users must be able to withdraw their deposits as well as their share of rewards, taking into account the timing of their deposits.

Key Features and Technical Specifications:

Smart Contract Environment: Developed in Solidity ^0.8.24, the ETHPool smart contract is optimized for security, efficiency, and compatibility with the Ethereum Virtual Machine (EVM). This Solidity version ensures the contract leverages the latest Ethereum features while prioritizing security and performance.

User Deposits and Withdrawals: Users can deposit ETH into the pool at any time, and the contract records each user's deposits and rewards. Withdrawals are flexible, allowing users to withdraw their deposits along with their share of the rewards, ensuring the security of users' total balances. While the `onlyTeam` modifier restricts the ability to deposit rewards into the pool to authorized team members.

Reward Distribution Mechanism: Rewards are distributed based on each user's proportion of the total pool deposits. Importantly, when the team deposits rewards, the contract adjusts the share of the pool attributed to each user, effectively increasing their stake in the pool's rewards.

Functions

Function Name	**Description**
deposit()	Allows users to deposit ETH into the pool. Checks for a minimum deposit amount and updates the user's and pool's total deposits.
withdraw(uint _amount)	Enables users to withdraw a specified amount from their balance. Calculates the user's share of deposits and rewards, ensuring the withdrawal amount does not exceed their total balance.
rewardDeposit()	Exclusive to the team, this function allows for the deposit of rewards into the pool. Distributes the rewards based on the current proportion of each user's stake in the pool.
getUserDeposit(address _user)	Returns the current deposit amount of a specific user.
getUserReward(address _user)	Returns the current reward amount of a specific user.

Events

Event Name	Description
Deposit(address indexed user, uint amount)	Emitted when a user makes a deposit into the pool.
RewardDeposit(uint timestamp, uint amount)	Emitted when the team deposits rewards into the pool, including the timestamp for tracking.
Withdraw(address indexed user, uint amount)	Emitted when a user withdraws funds from the pool.

ETHPool Test Suite Documentation

The ETHPool test suite, written using Hardhat and Chai, is designed to validate the functionality and security of the ETHPool smart contract. This documentation provides a detailed overview of each test case within the suite, explaining the purpose, methodology, and expected outcomes. Before each test, the environment is set up with three signers (representing the team and two users) and deploys the ETHPool contract.

This ensures a fresh instance of the contract for each test case.

Test Case	Description	Execution	Expectations
User Deposit Functionality	Validates that users can deposit ETH into the pool and that these deposits are accurately recorded in the contract's state.	User1 deposits 1 ETH.	The deposit is correctly logged, and the contract's total deposits reflect this addition.
Exclusive Reward Deposit by the Team	Ensures that only the team can deposit rewards into the pool, enforcing the contract's security and intended functionality.	User1 attempts to deposit rewards, expecting the action to fail.	The contract reverts the transaction, maintaining exclusive control for the team.
Reward Deposit Preconditions	Confirms that rewards cannot be added to the pool unless there are existing user deposits, preventing potential errors or misuse.	The team attempts to deposit rewards into an empty pool.	The contract reverts the transaction, indicating that rewards require prior user deposits.
User Withdrawal Functionality	Tests the ability of users to withdraw their deposits and rewards, a critical	User2 deposits 1 ETH, receives rewards, and then withdraws 2 ETH.	The withdrawal is processed correctly, and the contract's

	feature for user trust and contract utility.		balances are updated accordingly.
Proportional Reward Distribution	Verifies that rewards are distributed among users based on their share of the total deposits, ensuring fairness and adherence to the contract's rules.	User1 and User2 deposit varying amounts, and the team distributes rewards.	Each user receives a share of the rewards proportional to their contribution to the pool.
Withdrawal Limits	Ensures users cannot withdraw more than their total balance (deposits plus rewards), a necessary check for contract integrity.	User1 attempts to withdraw more than their balance.	The contract reverts the transaction, enforcing the withdrawal limits.
Accurate Deposit Reporting	Confirms the contract accurately reports a user's deposit amount, essential for transparency and user confidence.	User1's deposit amount is verified after a deposit.	The reported deposit amount matches the actual deposit.
Accurate Reward Reporting	Ensures the contract accurately reports a user's reward amount, critical for clear and trustworthy user interactions.	User1's reward amount is verified after rewards are distributed.	The reported reward amount accurately reflects the rewards earned.
Dynamic Reward Distribution with New Deposits	Tests the contract's ability to dynamically adjust reward distribution based on new deposits and rewards, a complex scenario reflecting real-world usage.	User1 and User2 make initial deposits, followed by a team reward deposit. User2 then makes an additional deposit before a second team reward deposit.	The final balances of User1 and User2 reflect the dynamic changes in the pool, demonstrating the contract's ability to handle evolving conditions and maintain fair reward distribution.

Compiling and Testing of ETHPool.sol

Before commencing with testing, the ETHPool smart contract must be authored to meet project specifications. The creation of this smart contract involves crafting Solidity code tailored to the requirements outlined for the ETHPool platform. Due to the unique nature of the project, directly utilizing the code from the solidity-challenge repository was not feasible. As such, the contract had to be authored from scratch to ensure alignment with project goals and specifications.

Compiling the Smart Contract

Once the ETHPool.sol smart contract is authored, the next step is to compile it to ensure that it is syntactically correct and adheres to the Solidity language standards. Compiling the smart contract involves translating the Solidity code into bytecode, which can be executed on the Ethereum Virtual Machine (EVM).

```
npx hardhat compile
```

This command initiates the compilation process and verifies that the smart contract is compiled without any errors. Any errors encountered during compilation are addressed and resolved to ensure the contract's correctness.

Running the Test Suite

With the compiled smart contract in hand, the next phase involves testing its functionality and security using the ETHPool test suite. This suite, meticulously crafted using Hardhat and Chai, comprises a series of test cases designed to validate various aspects of the smart contract's behavior.

```
npx hardhat test
```

Executing this command triggers the test suite, which meticulously scrutinizes the smart contract's response to different scenarios and interactions. Each test case evaluates specific functionalities, ensuring that they operate as intended and meet project requirements.

Results: **ETHPool**

✓ should allow users to deposit (71ms)

✓ should allow only the team to add rewards (115ms)

✓ should not to add rewards when no deposits

✓ should allow users to withdraw (107ms)

✓ should distribute rewards based on shares (90ms)

✓ should not allow users to withdraw exceeds balance

✓ should return correct user deposit amount

✓ should return correct user reward amount (57ms)

✓ should distribute rewards based on shares after multiple deposits and reward distributions (145ms)

9 passing (3s)

The Role of the `.env` File in Project Security and Configuration

The .env (environment variables) file plays a crucial role in the security and configuration of projects, especially when preparing for GitHub uploads or similar version control scenarios. It serves as a secure storage location for sensitive information that should not be hard-coded directly into the project's source code or exposed publicly. This practice is paramount in protecting API keys, secret tokens, database credentials, and other confidential data from unauthorized access or exposure.

Variables Stored in the `.env` File for the ETHPool Project:

For the ETHPool project, the .env file typically contains the following variables, which are critical for the project's deployment and operation on the Ethereum blockchain:

1. **SEPOLIA_RPC_URL**: This variable stores the Remote Procedure Call (RPC) URL for the Sepolia testnet. It's used by the project to connect to the Ethereum blockchain without running a full node, facilitating interactions with the blockchain.
2. **PRIVATE_KEY**: The private key of the deployer's Ethereum account. It's used to sign transactions, including the deployment of the smart contract, ensuring that the deployer has the necessary permissions and funds to deploy the contract.
3. **ETHERSCAN_API_KEY**: An API key for Etherscan, a blockchain explorer for Ethereum. This key is used to verify the smart contract's source code on Etherscan, making it publicly visible and verifiable by anyone.

Incorporating the `.env` File into the Deployment and Verification Process

Before proceeding with the deployment and verification of the ETHPool smart contract, it's essential to ensure that the .env file is correctly set up with the above variables. This setup not only secures sensitive information but also streamlines the deployment process by automating the use of these variables in scripts and Hardhat tasks.

Important Note: When uploading projects to GitHub or sharing them publicly, the .env file should be added to the .gitignore file to prevent accidental exposure of sensitive information. This practice keeps the project's configuration flexible and secure, allowing developers to share their code without risking the security of their accounts or the integrity of their applications.

Deployment Process

With the ETHPool smart contract rigorously tested and proven to meet our stringent requirements, we transition to the pivotal phase of deployment. This step will transition our smart contract from a local development environment to the Ethereum blockchain, making it accessible to users worldwide.

Deploying the Smart Contract:

The deployment is executed through a script typically located in the `scripts` directory, named something akin to `deploy.js`. This script is the conduit through which the ETHPool smart contract is brought to life on the Ethereum blockchain.

To deploy, navigate to the terminal within Visual Studio Code and execute:

```
npx hardhat run scripts/deploy.js --network sepolia
```

This command engages Hardhat's deployment mechanism, targeting the `sepolia` network as specified. Upon successful deployment, the terminal will display the contract's address on the blockchain, signifying the contract's readiness for interaction.

Verification Process

The final step in our smart contract's journey is verification, a process that enhances transparency and trust by making the contract's source code publicly available on Etherscan.

Why Verification Matters:

Verification allows users and developers to review the contract's code, ensuring it operates as intended and without hidden malicious intent. It also enables interaction with the contract directly through Etherscan's interface.

Verifying the Smart Contract on Etherscan:

To verify the contract, you'll need the deployed contract's address and the `ETHERSCAN_API_KEY` set within your `.env` file.

Execute the verification command:

```
npx hardhat verify --network sepolia YOUR_CONTRACT_ADDRESS
```

Replace `YOUR_CONTRACT_ADDRESS` with the actual address of your deployed contract. This command submits your contract's source code to Etherscan, where it undergoes a verification process.

Upon successful verification, Etherscan will link the verified source code to the contract's address page, making it accessible for public viewing and interaction.

Smart Contract Balance Query Script Documentation

Following the successful deployment and verification of the ETHPool smart contract, a crucial aspect of managing and monitoring the dApp involves regularly checking the contract's balance. This process is vital for ensuring the smooth operation of the reward distribution system and maintaining transparency with the platform's users. To facilitate this, a dedicated script has been developed to query the contract's balance on the Ethereum blockchain.

Script Overview: The balance query script is a Node.js application that leverages the Etherscan API to fetch the current balance of the ETHPool smart contract. This script is designed to be both efficient and user-friendly, providing real-time insights into the contract's holdings.

Functionality:

- The script defines a function `getContractBalance(contractAddress)` that takes the smart contract's address as an argument.
- It constructs a request URL using the contract address and the Etherscan API key stored in the `.env` file.
- The script then sends a GET request to the Etherscan API to retrieve the contract's balance in Wei.
- Upon receiving a response, it converts the balance from Wei to Ether for readability and logs it to the console.

Usage:

- To use the script, ensure that the `.env` file is correctly set up with your `ETHERSCAN_API_KEY`.
- Replace the placeholder `contractAddress` with the actual address of your deployed ETHPool smart contract.
- Execute the script by running: `node path/to/your/script.js`

The script will output the current balance of the ETHPool smart contract in Ether, providing a clear and immediate understanding of the contract's financial status.