Dùng solidity, ethers.js tạo DApp và smart contract

In this assignment, you'll use Solidity and ethers.js to implement a complex decentralized application, or DApp, on Ethereum. You will write both a smart contract and the user client that accesses it, learning about 'full-stack' development of a DApp. To save you time, please read the whole assignment – especially the notes section – before you start development.

1 Blockchain Splitwise

We want to create a decentralized system to track debit and credit - a blockchain version of Splitwise. If you haven't heard of the app, it's a simple way to keep track of who owes who money within a group of people (maybe after splitting lunch, groceries, or bills). To illustrate the application, consider the following scenario:

Alice, Bob and Carol are all friends who like to go out to eat together. Bob paid for lunch last time he and Alice went out to eat, so Alice owes Bob $10. Similarly, Carol paid when she and Bob went out to eat, and so Bob owes Carol $10.

Now, imagine Carol runs short on cash, and borrows $10 from Alice. Notice that at this point, instead of each person paying back their 'loan' at some point, they could just all agree that nobody owes anyone. In other words, whenever there is a cycle of debt, we can just remove it from our bookkeeping, making everything simpler and reducing the number of times cash needs to change hands.

We will build a decentralized way to track who owes what to who, so that no trusted third party has to be relied upon. It will be efficient: it won't cost an exorbitant amount of gas to store this data. No value will get transferred 'on the blockchain' using this app; the only ether involved will be for gas.

Because it's on the blockchain, when Carol picks up the check for her and Bob's meal, she can ask Bob to submit an IOU (which he can do using our DApp), and she can verify that he indeed has. The public on-chain storage will serve as a single source of truth for who owes who. Later, when the cycle illustrated above gets resolved, Carol will see that Bob no longer owes her money.

As part of this, we will also build a user interface that computes useful information for the user and allows non-programmers to use the DApp.

2 Getting Started

2.1 Setup

1. Install the prerequisite software: you'll need to download and install Node.js from `https://nodejs.org/en/`. Choose the LTS version (the one on the left).

2. Download and extract the starter code from the course website.

3. `cd` into the starter code directory.
4. Run `npm install --save-dev hardhat` to install the Ethereum development environment Hardhat, which you will use to simulate an Ethereum node on your local machine.
5. Run `npm install --save-dev @nomiclabs/hardhat-ethers ethers` to install a Hardhat plugin that your script `scripts/deploy.js` to deploy a Hardhat node will use.

2.2 Compile, Deploy and Test

1. Open the starter code directory in your favorite IDE or text editor (something like Sublime Text, Atom, or Visual Studio Code works nicely). You'll be only modifying `contracts/mycontract.sol` to define your Solidity contract and `web_app/script.js` to build the client. Looking at the other files may help understand how the Hardhat node and web client works. There are places marked with functions to modify and you can add helper functions to `web_app/script.js`. Please do not modify any other code or install additional node packages.
2. Peruse the starter code, the ethers.js, and the Solidity documentation [8]. Think carefully about the overall design of your system before you write code. What data should be stored on chain? What computation will be done by the contract vs. on the client?
3. After you finish implementation, run `npx hardhat node` to start the local node. If the node is started correctly, you should see in terminal: *Started HTTP and WebSocket JSON-RPC server at https://localhost:8545*, followed by 20 accounts with corresponding private keys.
4. Open another terminal tab or window. `cd` into the starter code directory.
   Run `npx hardhat run --network localhost scripts/deploy.js` to compile and deploy your contract. Upon success, expect to see this message in your terminal: *Finished writing contract address: ....* Save this address for use in the next step.
5. **Update the contract address and ABI** in `web_app/script.js`. The ABI can be copied from an auto-generated file `artifacts/contracts/mycontract.sol/Splitwise.json`. To correctly update the ABI, please copy the whole list after the 'abi' field, starting from the square bracket. The address is saved in the previous step. Make sure your contract address is a string.
6. Open the `web_app/index.html` file in your browser. You can either play with your system through the frontend, or run our sanity check! More information in **section 7.2 Practical Development and Debugging**.

Note on OSs: All of the above steps should work on Unix-based systems and Windows. The commands we ask you to execute will work in a standard Unix terminal and the Windows Command Prompt.

3 Components

The project has two major components: a smart contract, written in Solidity and running on the blockchain, and a client running locally in a web browser, that observes the blockchain using the ethers.js library and can call functions in the smart contract. For more information on how ethers.js works and setting up this assignment, watch the section 3 recording on panopto.

3.1 Functions in the client

Please note, all the client functions we ask you to implement are given as async functions in the starter code. This means they return a Promise by default. Our grading system will assume a Promise is returned from each client function. For more about async functions, Promises, and await, see the reference at the bottom of this handout.

1. `getUsers()`: Returns a Promise for a list of addresses. The list would be the addresses of 'everyone who has ever sent or received an IOU'. You may find this useful as a helper for other functions.
2. `getTotalOwed(user)`: Returns a Promise for the total amount that the given `user` owes.
3. `getLastActive(user)`: Returns a Promise for a UNIX timestamp (seconds since Jan 1, 1970) of the last recorded activity of this user (either sending an IOU or being listed as 'creditor' on an IOU). Returns `null` if no activity can be found.
4. `add_IOU(creditor, amount)`: Submits an IOU to the contract, with the passed `creditor` and `amount`. **See note about resolving loops below**.

3.2 Functions in the contract

1. `lookup(address debtor, address creditor) public view returns (uint32 ret)`: Returns the amount that the `debtor` owes the `creditor`.
2. `add_IOU(address creditor, uint32 amount, ...)`: Informs the contract that `msg.sender` now owes `amount` more dollars to `creditor`. It is additive: if you already owed money, this will add to that. The amount **must** be positive. You can make this function take any number of additional arguments as long as the first two arguments are `creditor` and `amount`. **See note about resolving loops below**.

You are welcome to write more helpers for either the client or contract. The client can call contract functions with `BlockchainSplitwise.functionName(arguments)` and `BlockchainSplitwise.connect(anotherSigner).functionName(arguments)`. Please see documentation *here* for how to interact with your Solidity contract functions from the client with ethers.js. Make sure you know what the difference is between those two methods. Remember that the client functions will be written in JavaScript, and the contract functions will be written in Solidity.

4 Resolving Loops of Debt

It's helpful to think of the IOUs as a graph of debt. That is, say that each user is a node, and each weighted directed edge from A to B with weight X represents the fact 'A owes $X to B'. We will write this as $A \xrightarrow{X} B$. We want our app to 'resolve' any cycles in this graph by subtracting the minimum of all the weights in the cycle from every step in the cycle (thereby making at least one step in the cycle have weight '0').

For example, if $A \xrightarrow{15} B$ and $B \xrightarrow{11} C$, when C goes to add $C \xrightarrow{16} A$, the actual balances will be updated to reflect that $A \xrightarrow{4} B$, $B \xrightarrow{0} C$, and $C \xrightarrow{5} A$.

Similarly, if C goes to add $C \xrightarrow{9} A$, the actual balances will be updated to reflect that $A \xrightarrow{6} B$, $B \xrightarrow{2} C$, and $C \xrightarrow{0} A$.

The requirement is that if any potential cycles are formed when you are about to add an IOU using the client (`add_IOU`), you must 'resolve' at least one of them. You **do not** need to worry about complex cases involving multiple loops, or optimizing which path to take (something like max flow) in those cases. You can assume that as a precondition to both contract functions (`add_IOU` and `lookup`), there are no cycles in the graph. Finally, you can also assume that any cycle found will be somewhat small (say, less than 10).

We provide you with a breadth-first search algorithm in the code - to use it, pass in a start and end node, and a function to get the 'neighbors' of any given node. You are free to not use this implementation as well.

It's up to you to implement this resolution securely. It should not be possible for a malicious client to somehow 'wipe away' their debt once it is posted.

We can now illustrate exactly how you can pay back an IOU in this system. Say Alice borrowed $10 from Bob; now, she wants to pay Bob back in cash. When Alice gives Bob $10 in cash, Bob will add an IOU for $10 with the creditor as Alice. This will create a cycle: specifically, $A \xrightarrow{10} B$ and $B \xrightarrow{10} A$. By the cycle resolution requirements above, this will end with $A \xrightarrow{0} B$ and $B \xrightarrow{0} A$.

5 Overall Requirements

You are welcome to write your contract in any way you like, as long as it has the specified `lookup` and `add_IOU` functions. Your goal is to write a contract that minimize the amount of storage and computation used by both contract functions. This will minimize gas costs.

You can assume that the transaction volume is small enough that it's feasible to search the whole blockchain on the client, but you should not assume that the only users are the ones in your wallet - in other words, `provider.listAccounts()` does not contain every possible user of the system.

6 Submitting your code

We will be using Gradescope for submission. Please submit your **mycontract.sol** and **script.js** file **ONLY**! Your submission will be graded on whether it correctly answers queries, whether it incurs a reasonable amount of gas and contract security.

7 Notes

We will be posting an up-to-date listing of all clarifications and advice on Ed. Please follow that post to get the latest information as we work through any issues with the assignment.

7.1 System Architecture

- You should decide on what data structure(s) will be stored on the blockchain first. Think carefully about what you information you need to provide to the client. You don't need to use any particularly fancy data structures. Your decision may make the implementation more difficult, so you should be okay with going back and changing your architecture.
- We have not mentioned what to do in the case of a cycle formed between just two people. We recommend designing your system so that this is not a special case - when the debtor has 'paid back' the creditor, the creditor simply attempts to add an IOU in the opposite direction, triggering cycle resolution and ending with both owing 0 to each other. We also recommend that you avoid any concept of 'negative' debt, as this can over-complicate things.
- Remember when optimizing for gas cost that functions run on the client are free - they incur no cost.
- We suggest that after designing your system, you start by writing and thoroughly debugging the contract.
- You don't need a massive amount of code to complete the assignment. Our solution is about 40 lines of Solidity and about 70 new lines of JavaScript (not including the ABI).

7.2 Practical Development & Debugging

- To debug client-side code, make liberal use of `console.log`. You should see the results of the calls and the line number they originated from in your browser's JavaScript console.
- Warnings about synchronous XMLHttpRequest are fine to ignore.
- Solidity has a very useful function `require` that will allow you to check preconditions
- The autograder will assume the client functions return Promises for some other value. For more information about promises and asynchronous code, please see the references at the bottom of this handout. We have also provided a sanity check test function that should enable you to understand how we will test your code. You can call this function from the browser console or uncomment the call to it in `script.js`. It is recommended that you redeploy your contract before you run the sanity check function to reset the system to its

initial state. You are encouraged to write other tests, but please make sure your code passes the provided sanity check before submitting.

- The ABI decoder will decode function inputs in all lower case. To accommodate this, the starter code attempts to return all values as their lower case version using the function `toLowerCase()`. Remember to keep your upper and lower case values straight. Each function should work regardless of the case of the alphabetic characters passed in as part of a hexadecimal value.
- Please note that we are using Solidity version 0.8.17 so do not rely on documentation for the older versions since they may behave differently.

8 References

- You can read about how to open the JavaScript console of your browser *here*.
- A guide to async/await in Javascript *here*.
- A guide to Promises in Javascript *here*.
- A helpful tutorial for understanding Hardhat is *here*.
- The ethers.js documentation is *here*.
- The Solidity v0.8.17 documentation is *here*.