XCS251 Assignment 4: An Ethereum payment app

Due Sunday, October 16 at 11:59pm PT.

Guidelines

1. If you have a question about this assignment, we encourage you to post your question on slack channel.
2. Familiarize yourself with the collaboration and honor code policy before starting work.

Submission Instructions

You should submit a PDF with your solutions online in Gradescope. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Download and install Tex Live or try Overleaf.
- Submit the compiled PDF.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on assignment in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the assignment the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Introduction

In this assignment, you'll use Solidity and web3.js to implement a complex decentralized application, or DApp, on Ethereum to track debit and credit . You will write both a smart contract and the user client that accesses it, learning about 'full-stack' development of a DApp. Details on the application, the instructions on getting started with the started code, and requirements of the design will be provided in the handout. To save you time, you should read the whole assignment handout - especially the notes section - before you start development. By the end of this assignment, you should have a strong understanding of Ethereum smart contracts.

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

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. Run `npm install -g ganache-cli` to install the Ganache CLI, which we will use to simulate a real Ethereum node on our local machines. Then, run `ganache-cli` to run the node. You should see something like Figure 1 on the screen.

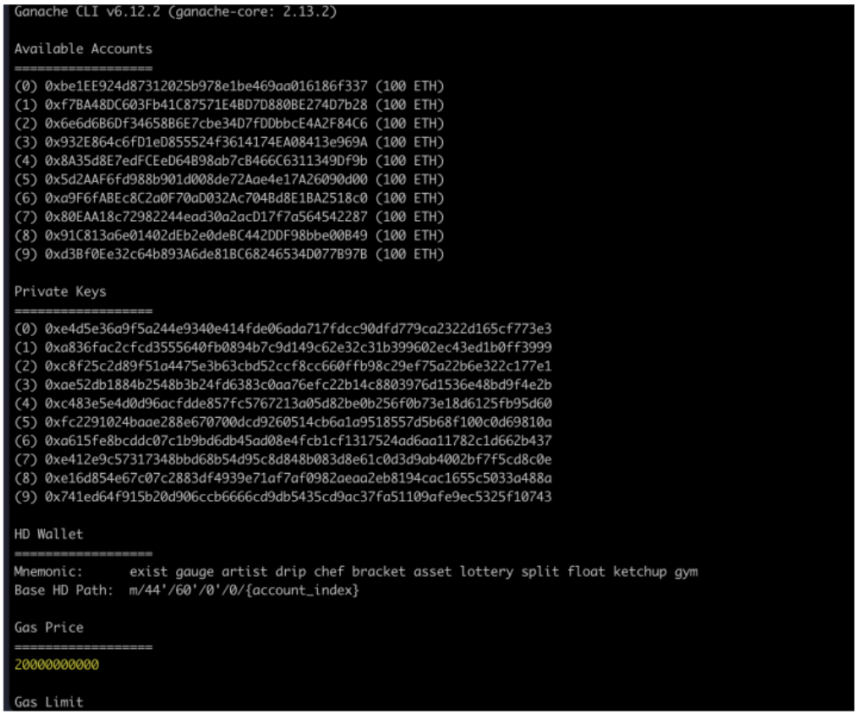

Figure 1: Screenshot of running ganache-cli.

   You can stop the node at anytime with Ctrl-C. If npm fails to install Ganache due to a lack of permissions, you can run npm again as root with `sudo npm install -g ganache-cli`.

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

4. Open https://remix.ethereum.org in your web browser. We recommend Firefox or Chrome. Remix supports the following browsers: Firefox, Chrome, Brave. Remix does not support the use on tablets or mobile phones. In the 'Deploy and Run Transactions' tab (on the left hand of the page), set the environment to 'External HTTP Provider'. This will launch a popup menu. On the menu, make sure

that the 'External HTTP Provider Endpoint' is set to `http://localhost:8545` or `http://127.0.0.1:8545` - this should be the default - then click 'Ok'. This is where you will develop your smart contract (which you will write in Solidity). **Confirm that the environment remains set as External HTTP Provider.** If you see a popup that begins "Not possible to connect to the External HTTP provider...", please try again using the latest version of Firefox. We have found that it works the most consistently.

5. Open the `index.html` file in your web browser (you should see a page titled 'Blockchain Splitwise'). If your browser defaults to searching for index.html on Google, you can open the file directly by clicking on it in your file system. It's very helpful to also open your browser's JavaScript console, so that you can see error messages (see Reference 1). If everything so far is working, you should see no errors in the console (you will probably see a warning; this is fine, see the the notes at the end for more details).

6. 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 modifying `script.js` to build the client, but looking at the other files may help. There are places marked with functions to modify - please do not modify any of the other code. Feel free to add helper functions.

7. Peruse the starter code, the web3.js API (see Reference 2), and the Solidity documentation (see Reference 7). web3.js is a collection of libraries that allow you to interact with a local or remote ethereum node using HTTP, IPC or WebSocket. 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?

8. Implement code for the requirements outlined below. When you have your contract, compile the contract (see https://remix-ide.readthedocs.io/en/latest/compile.html to understand Solidity Compiler), deploy it using remix, and then **update the contract hash and ABI** in `script.js`. Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem. Hence, we need to copy contract ABI in `script.js` so that javascript knows how to call solidity contract. The ABI can be copied to the clipboard from the 'Solidity Compiler' tab, and the contract hash can be copied from the 'Deployed Contracts' panel in the 'Deploy and Run Transactions" tab. Note that the contract hash is *not* the transaction hash of the transaction that created the contract. Demos for copying ABI and contract address are provided in 8.2 and 8.3.

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 Requirements

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 web3.js and can call functions in the smart contract. As described in 2, open index.html in the browser to see the interface to interact with the deployed contract, which utilizing web3.js. The functions in the client will be written in script.js. The functions in the contract will be written in Solidity, which you compile and test in remix.

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. In short, synchronous functions return after operation is finished, and may prevent our program from continuing to run. Asynchronous function, i.e., function with keyword async, starts the operation and returns right away, so our program can still be responsive to other events and be notified with the result of the operation, when it eventually completes (see Reference 3 and 4).

Async functions return a Promise by default. A promise is an object returned by an asynchronous function, which represents the current state of the operation. At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation (see Reference 5).

You will notice that inside an async function of the starter code, the await keyword before a call to a function that returns a promise is used. This makes the code wait at that point until the promise is settled, at which point the fulfilled value of the promise is treated as a return value, or the rejected value is thrown. This enables you to write code that uses asynchronous functions but looks like synchronous code (see Reference 4).

Our grading system will assume a Promise is returned from each client function. For more about async functions, Promises, and await, see Reference 3, 4, and 5.

1. `getUsers()`: Returns a Promise for a list of addresses. You can have this return either: 'everyone who has ever sent or received an IOU' OR 'everyone currently owing or being owed money'. You may find this useful as a helper for other functions.

   Hint: If you look at every block, you can see every transaction and find every address that has been involved in a transaction. We have partially implemented this for you in the JavaScript starter code.

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. **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.methods.functionname(arguments).call()` and `BlockchainSplitwise.methods.functionname(arguments).send()`.

`BlockchainSplitwise.methods.functionname(arguments).call()` will call a "constant" method and execute its smart contract method in the EVM without sending any transaction. Note this cannot alter the smart contract state.

`BlockchainSplitwise.methods.functionname(arguments).send()` will send a transaction to the smart contract and execute its method. Note this can alter the smart contract state.

Please see documentation *here* for how to use `send` and `call` to call your Solidity contract functions from the client with web3.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 $C \xrightarrow{5} A$ as shown in Figure 2.

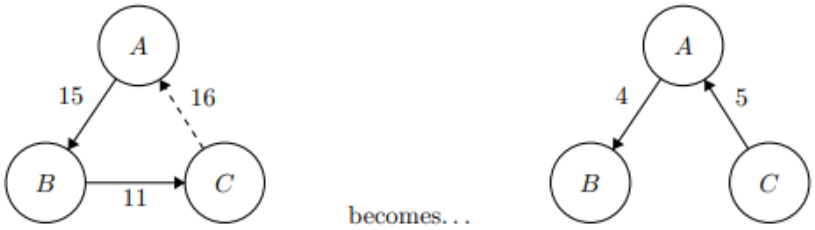

Figure 2:

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 $C \xrightarrow{0} A$ as shown in Figure 3.

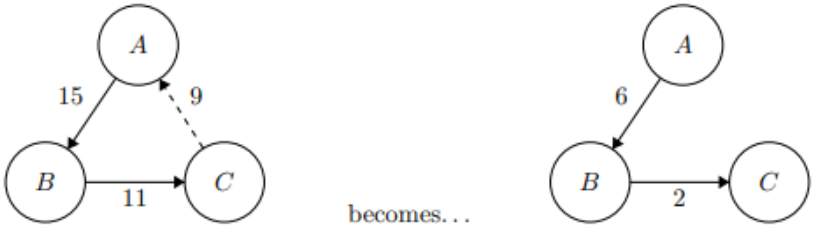

Figure 3:

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, `web3.eth.getAccounts()` does not contain every possible user of the system.

6 Submitting your code

We will be using Gradescope for submission. Your submission will be graded on whether it correctly answers queries and whether it incurs a reasonable amount of gas. **Before submitting, please make sure to copy and paste your contract's Solidity code from Remix into `mycontract.sol`.**

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 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 overcomplicate 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 in Remix. You can call functions in the lower left panel, and

switch accounts using the 'Account' selector in the top left. To copy the addresses to your clipboard, you can click the copy icon next to the selector. Once you're certain the contract works as intended, then you should start writing the client.

- 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. Errors about not connecting to `localhost:8545` are usually because you don't have `ganache-cli` running.
- Solidity has a very useful function `require` that will allow you to check preconditions
- Click on "listen on all transactions" in the Remix console (bottom middle of the screen) to see incoming requests to your deployed contracts.
- If you want to just debug your contract, you can click on the "Debug" button next to incoming requests in the Remix console and replay the function calls step by step.
- 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 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 make sure to compile and test your Solidity contracts with the **0.8.9** compiler version. Having `pragma solidity 0.8.9;` at the beginning of the solidity code will make sure that you do compile with a compiler version 0.8.9. Please double check in the Compilation tab on Remix.

- web3.js version 1.3 should be included in the starter code in the `web3.min.js` and `web3.min.js.map` files. Please don't change or remove these files. If you get any strange errors if may be because you are not running version 1.3 of web3.js. Versions 0.X of web3.js have entirely different syntax. To double check what version you're running, call `console.log(web3.version)`.

8 Demos

8.1 Demo 1

We provide a demo based on the interface provided in `index.html`. An example of the starting page is shown in Figure 4. Note that there are 10 wallet addresses and you can select the address in my account to see how much the address owes and use the "Add IOU" function.

Figure 4:

Suppose that you copy the address 2 to "Address of person you owe" and type 10 in the "Amount you owe them" as shown in Figure 5.

Figure 5:

Press Add IOU. You can see address 1 now owes $10 as shown in Figure 6.

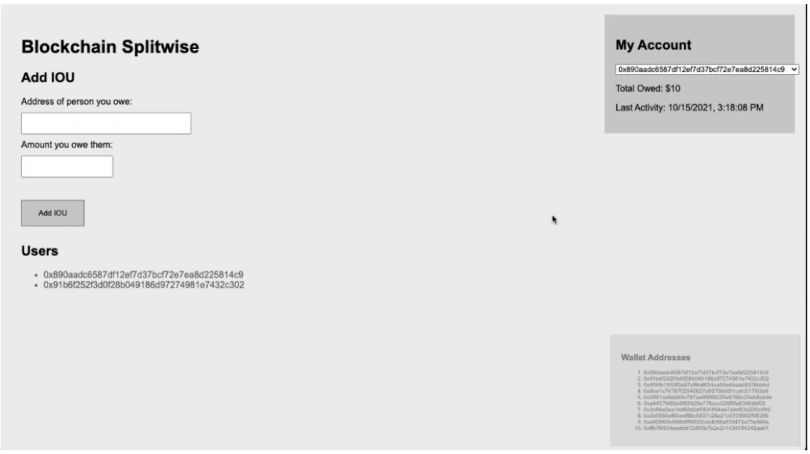

Figure 6:

Select address 2 in my account. Copy the address 3 to "Address of person you owe" and type 6 in the "Amount you owe them" as shown in Figure 7.

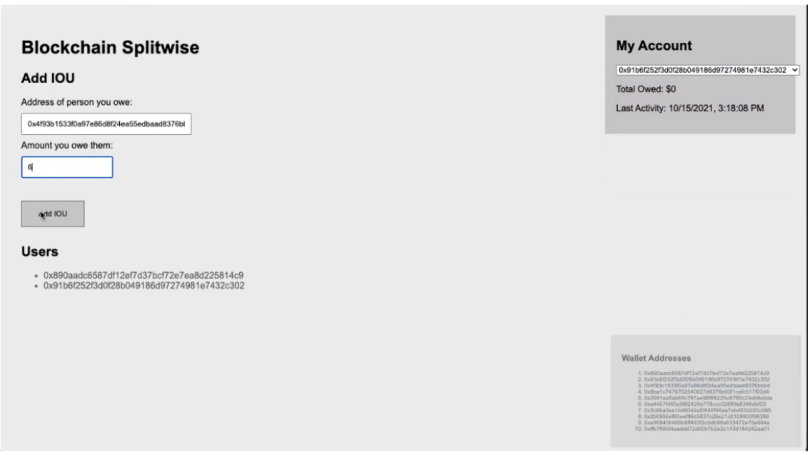

Figure 7:

Press Add IOU. You can see address 2 now owes $6 as shown in Figure 8.

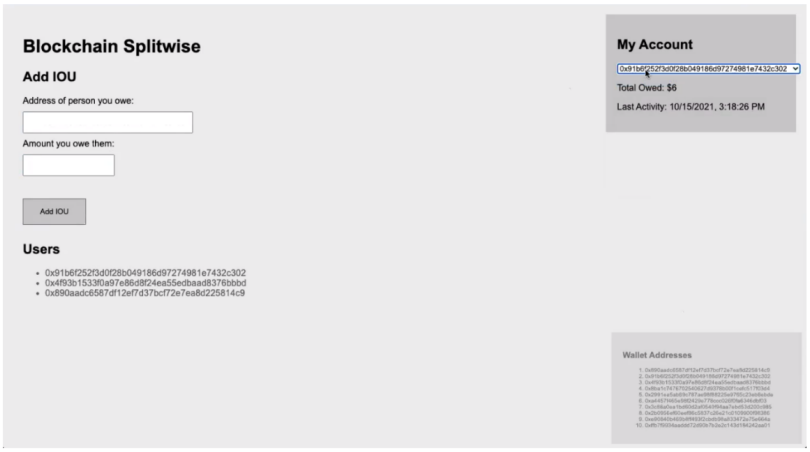

Figure 8:

Select address 3 in my account. Copy the address 1 to "Address of person you owe" and type 4 in the "Amount you owe them" as shown in Figure 9.

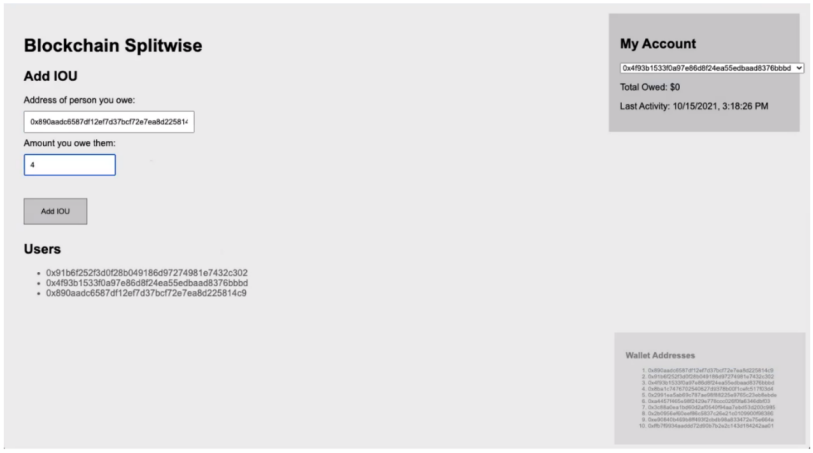

Figure 9:

Press Add IOU. You will now see address 1 owes only $6 because a cycle is resolved as shown in Figure 10.

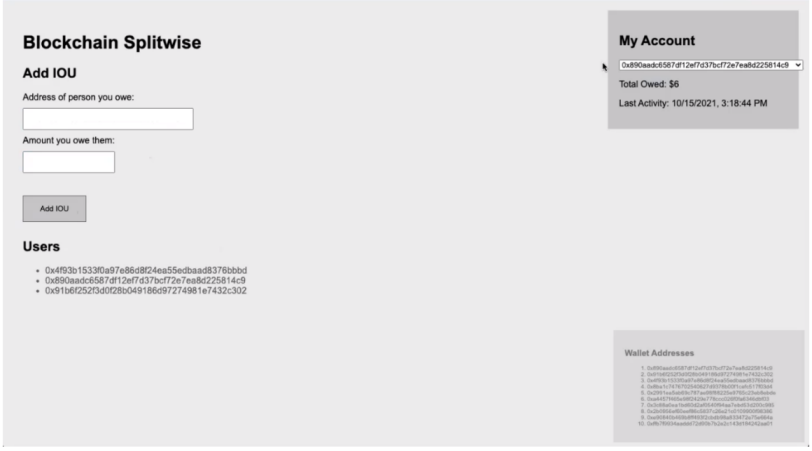

Figure 10:

8.2 Demo 2

After you compile the contract, you can copy the ABI by clicking the ABI tab as shown in Figure 11.

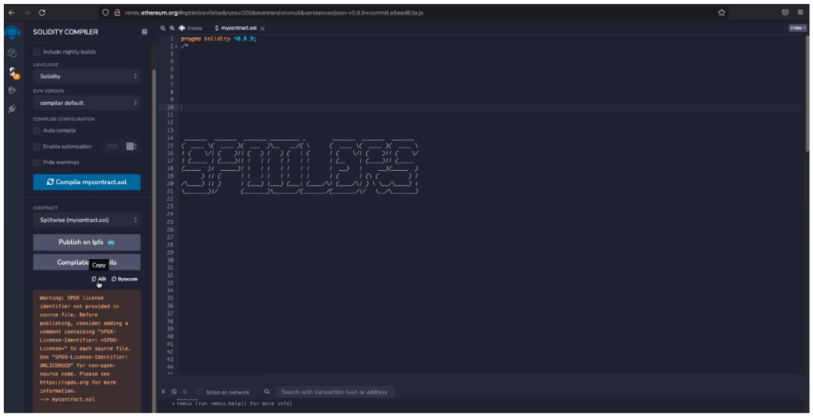

Figure 11:

Then you can put that in the `script.js` as shown in Figure 12.

```
// This is the ABI for your contract (get it from Remix, in the 'Compile' tab)
// ============================================================
var abi = [
    {
        "constant": true,
        "inputs": [
            {
                "name": "",
                "type": "address"
            },
            {
                "name": "",
                "type": "address"
            }
        ],
        "name": "debts",
        "outputs": [
            {
                "name": "",
                "type": "uint32"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [
            {
                "name": "debtor",
```

Figure 12:

8.3 Demo 3

After you deploy the contract, you can copy the contract address by clicking the tab beside the deployed contract as shown in Figure 13.

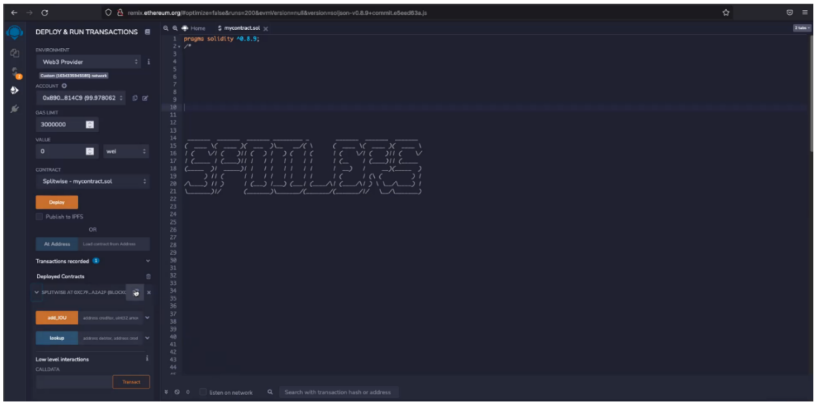

Figure 13:

Then you can put that in the `script.js` as shown in Figure 14.

```
abiDecoder.addABI(abi);
// call abiDecoder.decodeMethod to use this - see 'getAllFunctionCalls' for

var contractAddress = '0xd9145CCE52D386f254917e481eB44e9943F39138'; // FIXME
var BlockchainSplitwise = new web3.eth.Contract(abi, contractAddress);

// ================================================================================
//                                Functions To Implement
// ================================================================================
```

Figure 14:

9 FAQ and tips

1. The `add_IOU` function prototype we have to implement on the contract has uint32 as the type. This implies that the amount owed always has to be an integer.

2. The `script.js` file in starter code uses web3.eth.Contract, an object to interact with smart contracts on the ethereum blockchain. You may want to read a few examples in the web3js documentation to understand how to use the object. https://web3js.readthedocs.io/en/v1.2.11/web3-eth-contract.html

3. In the past, students asked if there is a hard limit on the gas cost per transaction for all the functions we wrote in the contract. For this project, we will only deduct points if you're doing something obviously inefficient. In practice, lots of popular contracts get optimized down to the per instruction level, but we do not expect you to do anything like this. Just to make sure your solution pushes as much work as possible off chain while remaining secure.

4. If you run the sanity check in `script.js` multiple times in a row, you will have it fail after the initial attempt, even with a correct solution. It's stateful, meaning we set some IOU and that persists after the test is over. You should start from scratch with a fresh chain rerunning and deploying the contract if you want to run sanity check.

5. In the client function `add_IOU`, you can get debtor address by using web3.eth.defaultAccount, which gets set to `$(this).val()` when an account is selected via the UI dropdown.

6. If you want to estimate the gas cost, you can use `estimateGas` (see Reference 2).

7. In the send() method in web3, you can specify a "gas" parameter to change the gas limit (see Reference 2).

8. To test your code, the script will not be able to do anything meaningful until the contract has been implemented. Once you've gotten your contract in a state where you think things are working, one possible way to proceed and test is to implement the javascript functions in the order that is listed on the handout.

9. Our sanity check code uses "===" to check the result. Hence, make sure that you get the type and value right to pass the sanity check.

10. When a call to a contract fails via require, all changes made to the state of the contract are automatically undone.

11. Sample code from remix has one line about SPDX-License-Identifier. Don't worry about the SPDX license, as we are not going to be officially publishing your contract.

12. Here are a few general security considerations for writing code with solidity.

    - Are we checking math calculations for overflows and underflows? In Solidity 0.8, the compiler will automatically take care of checking for overflows and underflows. See https://ethereum-blockchain-developer.com/010-solidity-basics/03-integer-overflow-underflow/.
    - What assertions should be made about function inputs, return values, and contract state?
    - Who is allowed to call each function?
    - Are we making any assumptions about the functionality of external contracts that are being called?

10 References

1. You can read about how to open the JavaScript console of your browser *here*.

2. The web3.js API is *here*.

3. A guide to asynchronous in Javascript *here*.

4. A guide to async/wait in Javascript *here*.

5. A guide to Promises in Javascript *here* and *here*.

6. The Remix documentation is *here*.

7. The Solidity v0.8.9 documentation is *here*.

8. To have code on your local computer automatically sync to Remix, check out `remixd` *here*.