COMS 4995-001 (Science of Blockchains): Homework #1

Due by 11:59 PM on Wednesday, February 5th, 2025

Instructions:

(1) Solutions are to be completed and submitted in pairs.

(2) We are using Gradescope for homework submissions. See the course home page for instructions, the late day policy, and the School of Engineering honor code.

(3) Please type your solutions if possible and we encourage you to use the LaTeX template provided on the Courseworks page.

(4) Write convincingly but not excessively. (We reserve the right to deduct points for egregiously bad or excessive writing.)

(5) Except where otherwise noted, you may refer to your lecture notes and the specific supplementary readings listed on the course Web page *only*.

(6) You are not permitted to look up solutions to these problems on the Web. You should cite any outside sources that you used. All words should be your own. Submissions that violate these guidelines will (at best) be given zero credit, and may be treated as honor code violations.

(7) You can discuss the problems verbally at a high level with other pairs. And of course, you are encouraged to contact the course staff (via the discussion forum or office hours) for additional help.

(8) If you discuss solution approaches with anyone outside of your pair, you must list their names on the front page of your write-up.

Throughout this homework assignment, unless otherwise specified, we consider a network of n validators, denoted by $\{v_1, \ldots, v_n\}$.

Problem 1

(10 points) This problem concerns protocol A from Lecture 3. Protocol A was our first attempt at a consensus protocol guaranteeing consistency and liveness in a synchronous network, in the presence of crash faults. Let us now consider a slightly revised fault model, referred to as *benign crash* faults. In this fault model, we treat message sending in a more atomic fashion. More precisely, a validator that crashes at timestep t, crashes exactly before sending *any* messages it intended to send at timestep t, or exactly after sending *all* messages it intended to send at timestep t. In other words, a validator either succeeds in sending all of its messages at a timestep or none of its messages. Prove that protocol A satisfies consistency and liveness in a synchronous network in the presence of an arbitrary number of benign crash faults.

Problem 2

(10 points) This problem concerns protocols A and B from Lecture 3. Setting aside benign crash faults and turning our attention back to regular crash faults (where a validator may crash after sending some but not all of the messages it intended to send at that timestep), suppose validators can order the messages they send at a given timestep t (e.g. first message to validator 17, then to validator 18, etc.) such that, in the

event that a validator v crashes at timestep t, it is guaranteed that the set of messages sent by v at timestep t is a *prefix* of that order. More formally, if validator v intended to send messages $m_1, \ldots, m_k$ (in that order) at some timestep, then the set of messages that actually get sent by v is guaranteed to be of the form $\{m_1, \ldots, m_j\}$ for some $j \in \{1, 2, \ldots, k\}$.

With this ability in mind, consider augmenting protocol A with the following changes:

1. Each validator sets its message delivery order to be the order of the subsequent leaders: the first message goes to the leader of the next view, the second message to the leader of the view after that, and so on.

2. As in protocol B from Lecture 3, the leader sends its entire chain instead of just the most recently constructed block. (If a validator hears a proposal from the current leader, it updates its local chain to that proposal; otherwise, it leaves its local chain as-is.)

Prove that with these modifications, protocol A satisfies consistency and liveness in a synchronous network in the presence of an arbitrary number of crash faults.

Problem 3

(10 points) This problem concerns protocol B from lecture 3. In this problem, we consider a new fault model, called the *omission fault* model. A validator that has suffered an omission fault behaves exactly like a non-faulty validator, except that at any timestep it may fail to send one or more of the messages it intended to send and/or may fail to receive one or more of the messages that were supposed to be delivered to it. (Intuitively, the validator's local Internet connection may drop outgoing or incoming packets without warning.)

- Explain why crash faults are a special case of omission faults.

- Suppose I promise you that at most one validator will suffer a crash fault and at most one other validator will suffer an omission fault. Does Protocol B still guarantee consistency in a synchronous network? Justify your answer either with a proof of consistency or with an execution of the protocol in which consistency is violated. (Remember that consistency needs to be satisfied only for the local chains of non-faulty validators.)

Problem 4 (Optional)

(5 extra-credit points) In protocol B from Lecture 3, a view may conclude with different non-faulty validators knowing about different sets of blocks (which is why everyone sent entire chains to each other rather than just the latest block). Suppose we really wanted each view to preserve the property that, at the end of the view, every non-faulty validator has exactly the same local chain (and were willing to make views longer in order to achieve this). With this in mind, suppose we modify Protocol A by expanding the view length to $k \cdot \Delta$ (for a parameter k, discussed below) and changing the pseudocode within a view to the following (in effect, with all validators repeatedly comparing notes on what they've heard so far in the view):

- At time $k\Delta \cdot v$, the leader ℓ of the view assembles a block B (e.g., all not-yet-included transactions that it knows about, ordered by time of arrival) and sends B to all validators.

- For $j = 1, 2, \ldots, k-1$:

  - At time $k\Delta \cdot v + j\Delta$: If validator i has received a block B (from the leader or secondhand from another validator) by this time during the view, send B to all validators.

- At time $k\Delta \cdot v + k \cdot \Delta$, if validator i received a new block B at some point in this view, update $C_i := (C_i, B)$. [I.e., append B to its local chain.]

Assume that the network is synchronous, and that you are promised that there will be at most f crash faults (you are told the value of $f \in \{1, 2, \ldots, n-1\}$ up front). What is the *smallest* value of k for which the subprotocol above guarantees agreement among the non-faculty validators at the end of a view (with either all the non-faulty validators appending the same block B to their local chains, or all non-faulty validators appending nothing to their local chains)? For full credit, provide a complete answer for all values $1 \leq f \leq n-1$ (i.e., a proof that your suggested value for k is sufficient for agreement at the end of a view, and that any smaller value of k is not).

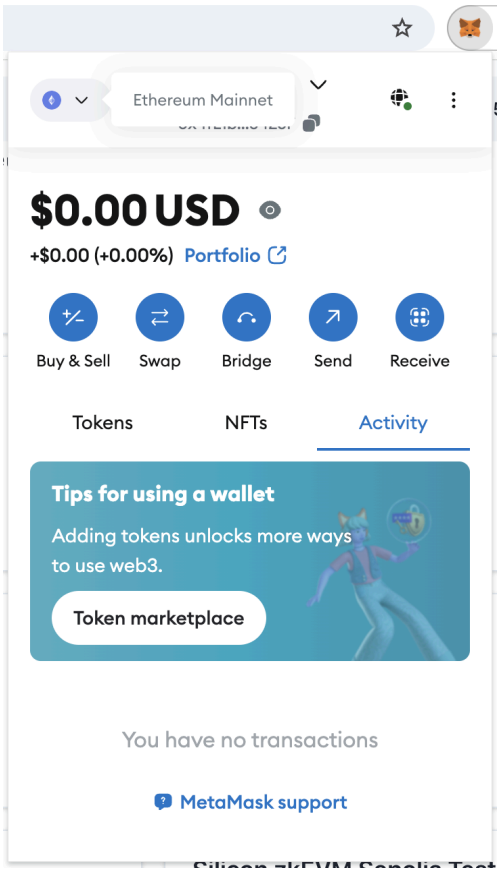
Ethereum Sepolia Demo (20 points)

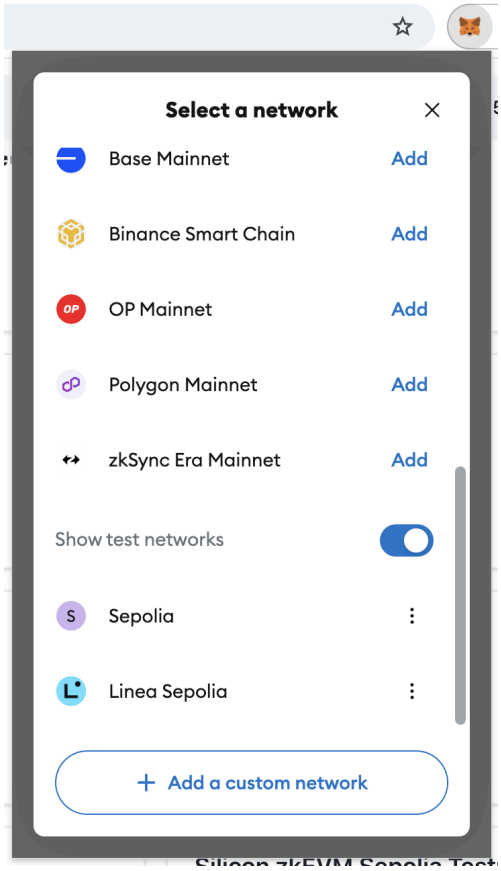
For the final task of this assignment, we're going to deploy a smart contract to the Ethereum Sepolia Testnet. Minimal/no coding is going to be required, and everything can be done from your browser. The main objective of this demo is to interact with a blockchain and develop a sense for what the practical end-to-end flow looks like.

We'll primarily be following this short tutorial: https://www.youtube.com/watch?v=I_OZd0HN7ro

Step 1: Download MetaMask. The first thing we're going to need to do is download a wallet client that will allow us to interface with the chain. We recommend downloading MetaMask, which you can use as a Chrome extension. You can download it here: https://metamask.io/download/

Step 2: Link MetaMask to Sepolia. Now that we have MetaMask installed, we need to make it interface with the Sepolia Testnet. You can do this by going to https://chainid.network/ and searching for "sepolia" in the search bar. Click the filter icon to the right of the search bar and make sure that "Show Testnets" is switched on. You will see many results, but only one of them will say just "Sepolia" and nothing else (it should have Chain ID 11155111). Click on "Add chain" for that result, and approve the MetaMask popup that comes up. After that, click on the MetaMask icon, and switch from Ethereum Mainnet to Sepolia by clicking the MetaMask icon on the Chrome toolbar, and then selecting the Ethereum icon on the upper left of the MetaMask menu (as in the left image). Then, scroll down to the bottom and select the "Sepolia" option (shown as a purple S in the right image below). This step is also shown in the YouTube tutorial.

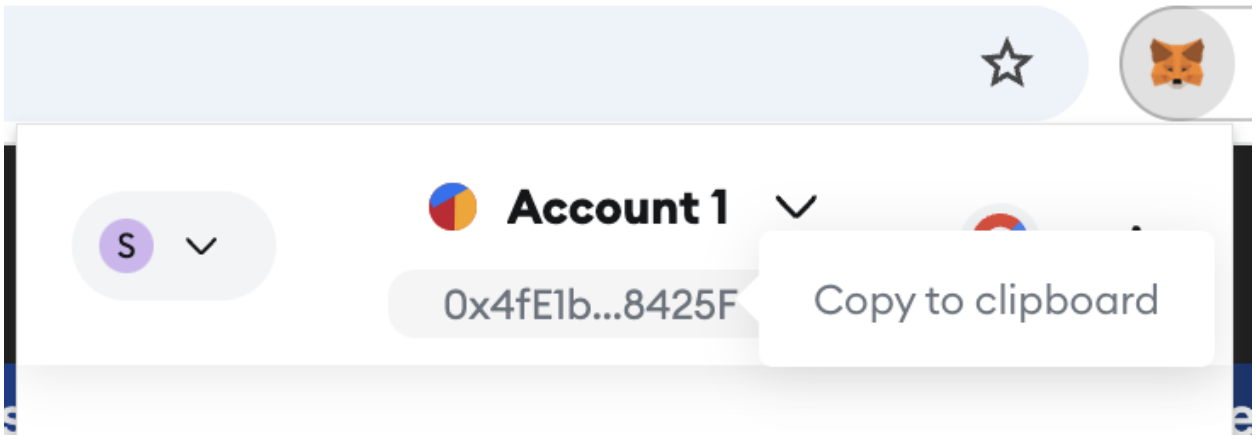




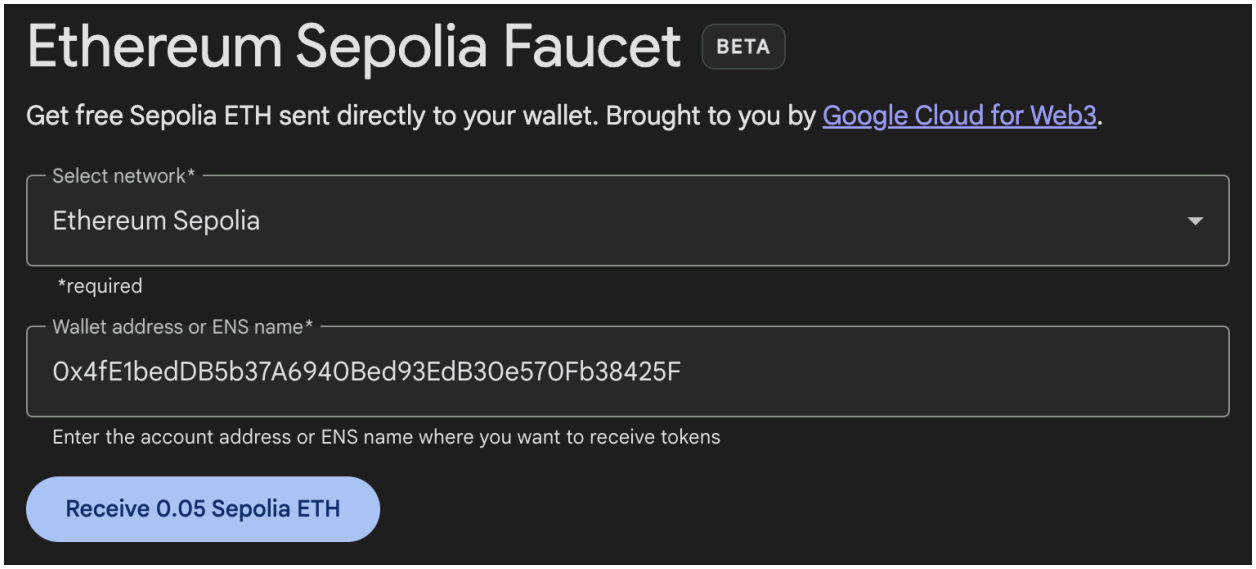
Step 3: Receive Sepolia Tokens. In order to submit a transaction (including the smart contract that we'll deploy today) to the chain, you're going to need tokens. This is because in order to manage congestion on the chain, a transaction fee is charged to users who submit transactions. On main networks, these tokens can hold real value. Today, we'll only be interacting with the Sepolia test network. While there is still technically still scarcity in test network tokens, you can acquire them for free through a *faucet* that distributes small denominations of the token.

The faucet that we'll use can be found here:
https://cloud.google.com/application/web3/faucet/ethereum/sepolia

After going to the website, first click the MetaMask icon and copy your wallet address:

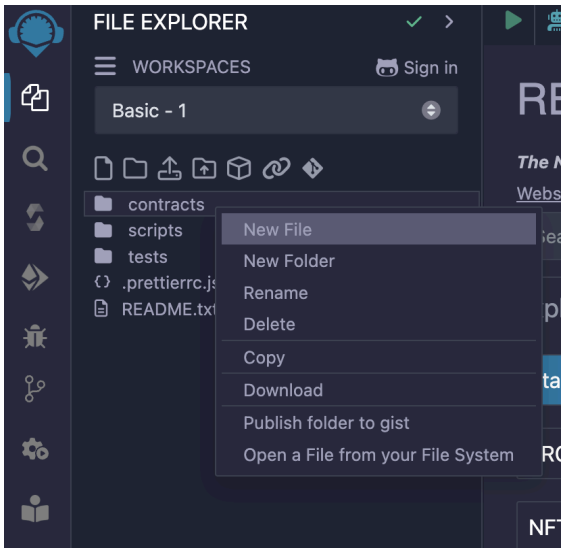


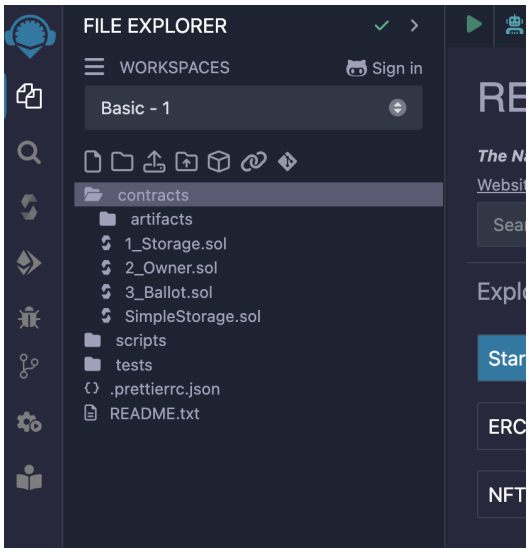
Then, paste it into the "Wallet address or ENS name*" box and click "Receive 0.05 Sepolia ETH



After this step, you should see 0.05 SepoliaETH in your wallet.

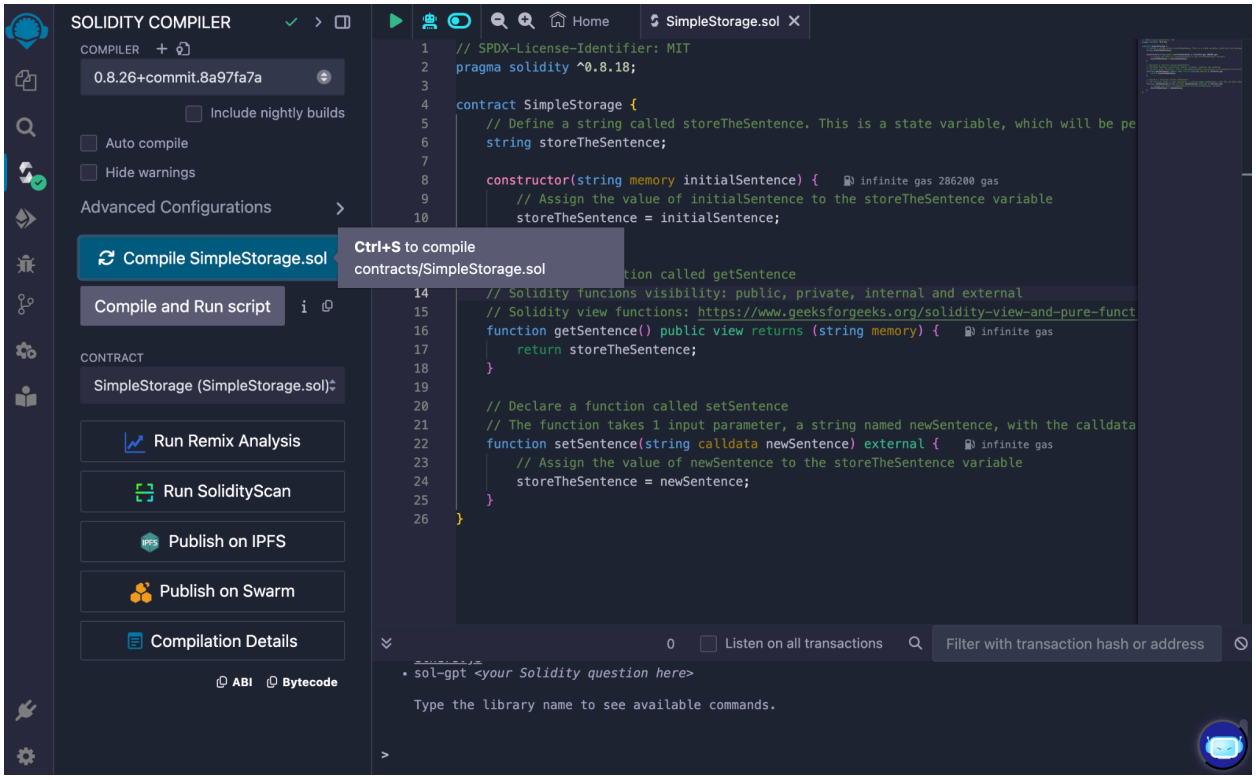
Step 4: Write a contract on Remix. Remix is a web IDE that allows you to write smart contracts in a programming language called Solidity. You can access Remix at https://remix.ethereum.org/. In the file explorer on the left, right click on the “contracts” folder and make a new file (as shown in the left image). Name the file “SimpleStorage.sol”. The explorer should look like the image on the right when you’re done.



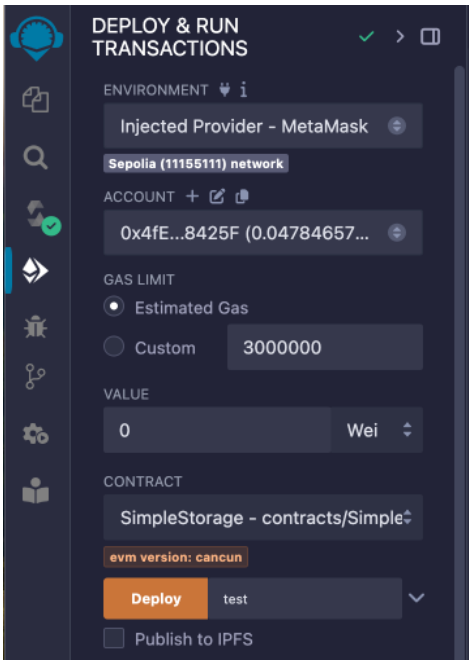


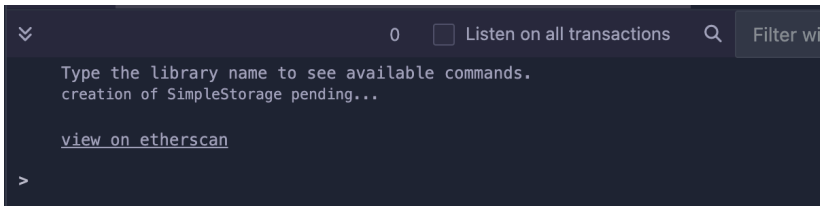
Next, open SimpleStorage.sol and paste in this code:
https://github.com/charmingdata/all-smart-contracts/blob/main/SimpleStorage.sol

Step 5: Deploy a contract on Remix. To deploy the contract, we have to first compile it. You can do that by either clicking Ctrl/Cmd-S, or by going to the compilation menu and clicking “Compile SimpleStorage.sol” as shown in the image below

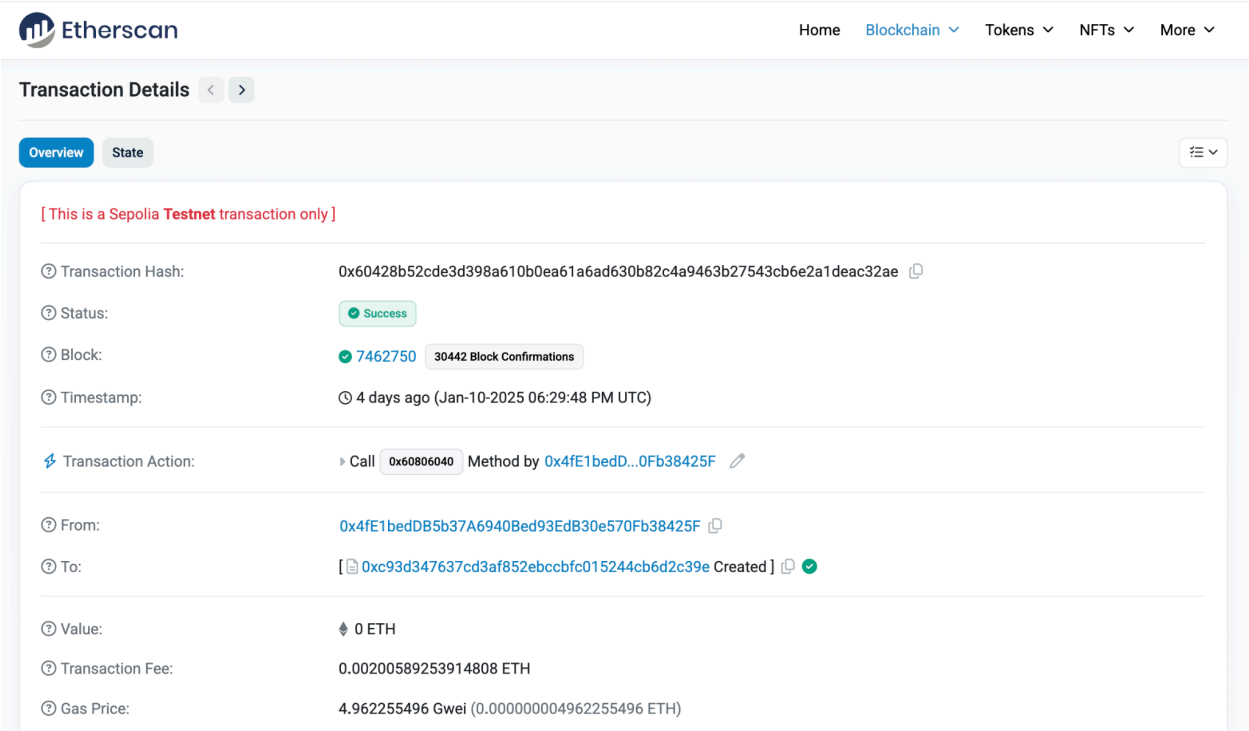


To deploy the contract, go to the deploy menu (indicated by the Ethereum sign and right arrow). Under Environment, select “Injected Provider - MetaMask”. Finally, type in a sentence (e.g. “test”) in the box next to the orange “Deploy” button. After clicking “Deploy” and confirming the MetaMask transaction fee, the contract will be live! You can view it on Etherscan by clicking the link in the Remix logs (as shown in the right image). The YouTube tutorial also shows this step.





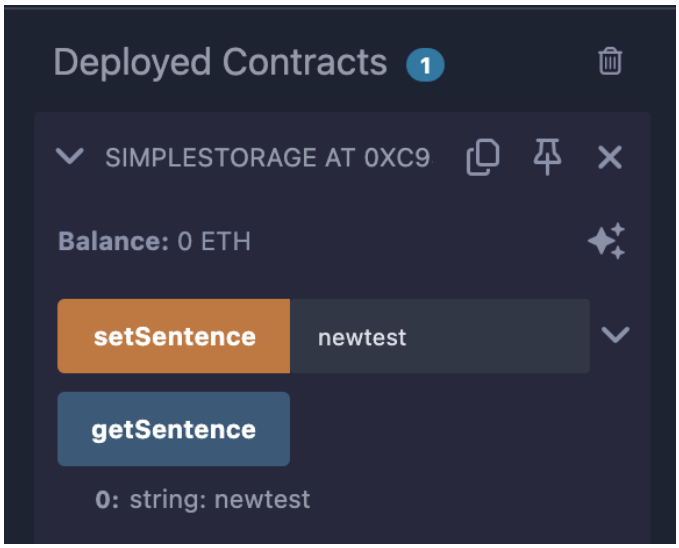
On Etherscan, you should see a page that looks like this



Congrats, you've deployed your first smart contract!

Step 6: Interacting with the contract

Now that the contract has been deployed, we can interact with it. In Remix, open the dropdown for SIMPLESTORAGE under "Deployed Contracts". When you click "getSentence", you should see the string you entered in before. If you type in a new string next to "setSentence" and hit the button, you'll be prompted by MetaMask to pay an additional transaction fee, and you should see another transaction show up on Etherscan. This is because when you're calling getSentence, you're only reading the network state. When you're calling setSentence, you're modifying it, which requires submitting a new transaction.



Step 7: Submit to Gradescope

Submit the URLs of the Etherscans for the transactions you submitted in steps 5 and 6 to Gradescope.