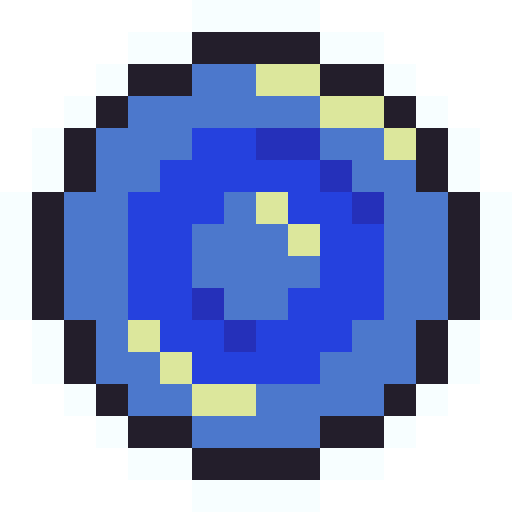
**STAKING APP DOCUMENTATION**

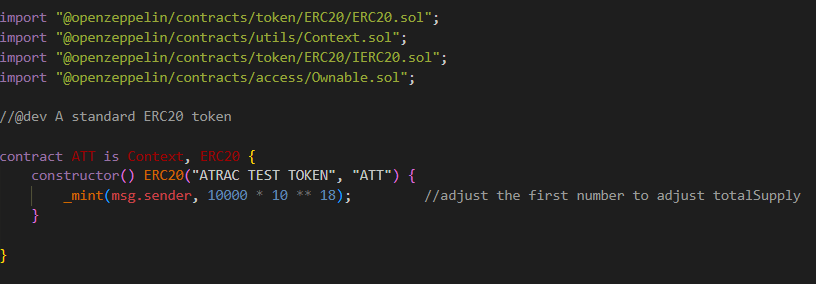
@dev Danilo Pavićević



# Solidity backend

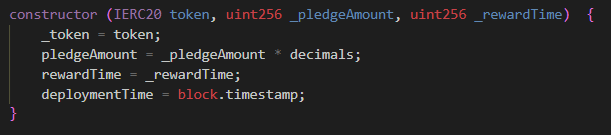
The backend of this project was written in Solidity using the 0.8 compiler version. All of the main logic of the application is on the smart contracts as that gives users assurance and safety as contracts are immutable and public. Backend is composed of two contracts, a Bank contract and a Token contract.

Token contract is the standard ERC20 token using the *openzeppelin* implementation.



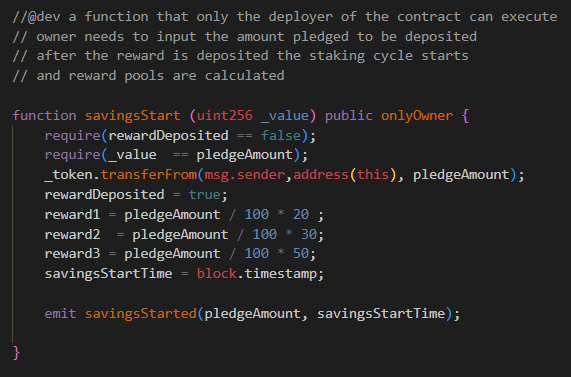
Adjusting the total supply can be done by simply changing the ‘10000’ argument in the *\_mint* function to the desired amount of tokens.

Bank contract handles all of the staking logic. Upon deployment the owner needs to pass the Token contract’s address to the Bank’s constructor. This way the Bank contract will be configured to use the Token that was passed as argument (In case of using Rinekby ATRAC, everything that is needed is its contract address). In the constructor owner must specify the *pledgeAmount*, which represents the amount of reward tokens he is going to deposit to start a staking cycle as well as the *rewardTime* which specifies how long is one reward period (*R*) going to last.



After deployment the owner of the contract can call the *‘savingsStart’* function to initiate a staking cycle. This differs from the task specification which says “The bank smart contract also contains an additional token reward pool of R XYZ tokens, deposited to the contract by the contract owner (bank owner) at contract deployment.”. To be able to deposit tokens at the contract creation, the owner would need to predict the Bank contract address, to give it allowance in the Token contract which would then allow the owner to deposit tokens upon calling the Bank constructor. This is a key safety feature of the ERC-20 token. While the address is deterministically calculated from the deployer’s address and the nonce, as I have not tried doing that in the past, I am not too familiar with the process and how that solution preforms. To take a safe route the Bank contract is set up in such a way that staking starts only after the owner deposits the *pledgeAmount* he specified in the constructor. This way also adds the advantage of reusability, as once the staking cycle ends the owner can start it again with the same parameters.

Upon owner depositing the *pledgeAmount* reward pools are calculated and *startTime* is set to current blockstamp.

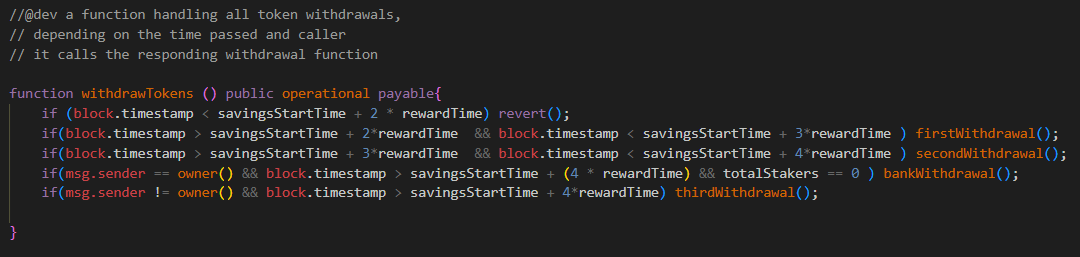


After the staking has started users have a period of *1R* to deposit their tokens. This ‘deposit period’ ends after the *1R* passes and deposits are no longer allowed. User data gets stored in a mapping and a struct.



In the time period from *1R* to *2R* the ‘lock’ period is active. During this time no user can deposit or withdraw tokens.

After *2R* the users are able to withdraw their tokens. The amount of reward they are going to recive depends upon how patient they are. When users decide it is time to withdraw, they call the *withdrawTokens* function which has a series of if checks to determine which withdrawal function is going to be called.



First, second and third withdrawal correspond to the reward time periods *R*, and determine the reward user is going to get. *BankWithdrawal* is a special function only available to the owner of the Bank contract that allows them to withdraw all tokens that remain in the contract and end the staking cycle.

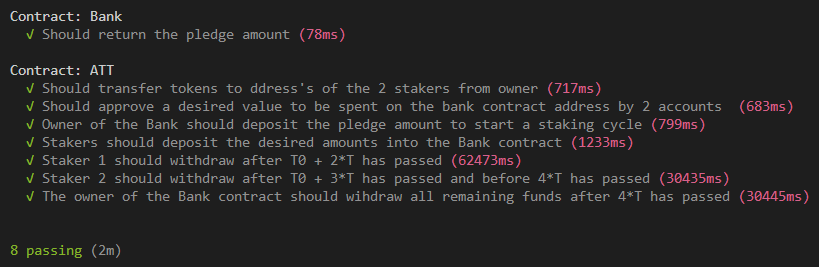
# Truffle tests

To test these Solidity contract functionalities a truffle test is used. The logic of the test is in the *test/ATT.js* . The composition of the test is as follows:

1. "Should return the pledge amount" - testing if the Bank is deployed correctly
2. "Should transfer tokens to addresses of the 2 stakers from owner" - testing if the Token is deployed correctly and sending tokens to the 2 stakers addresses so that they can deposit them
3. "Should approve a desired value to be spent on the bank contract address by 2 accounts " - testing giving approval to the Bank address to receive the tokens from the stakers
4. "Owner of the Bank should deposit the pledge amount to start a staking cycle" - testing the *startSavings* function by depositing the *pledgeAmount* from the owner to the Bank
5. "Stakers should deposit the desired amounts into the Bank contract" - testing users depositing their tokens in the period of *1R*
6. "Staker 1 should withdraw after T0 + 2\*T has passed" - testing the first staker withdrawing their reward after lockdown period
7. "Staker 2 should withdraw after T0 + 3\*T has passed and before 4\*T has passed" - testing the second staker withdrawing their reward in the second period
8. "The owner of the Bank contract should withdraw all remaining funds after 4\*T has passed" - testing the Bank owner withdrawing remaining tokens and ending the staking cycle

\*Note – tests should run with *rewardPeriod* set in *migarations* to “30”s.

The tests are run against a local blockchain in ganache using ***–truffle test***.



The contract passes all tests defined in the task specifications.

# React frontend

The frontend of this project was done in React. The main library needed to connect the React frontend to the ganache back-end is *web3*. After running the ganache blockchain, Metamask is required to interact with the front end and call the underlaying Solidity contracts in the backend.

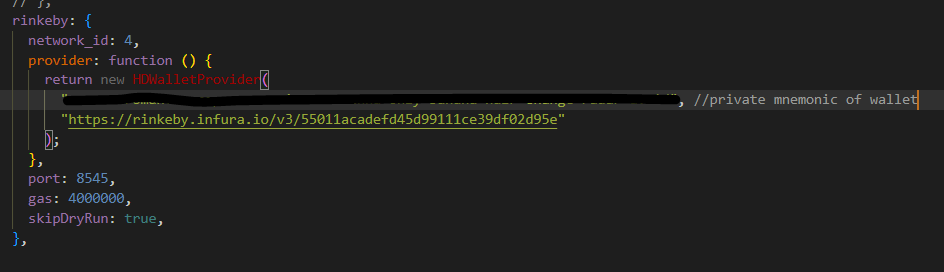
# Testing the full stack

To test the full stack app, next steps should be followed.

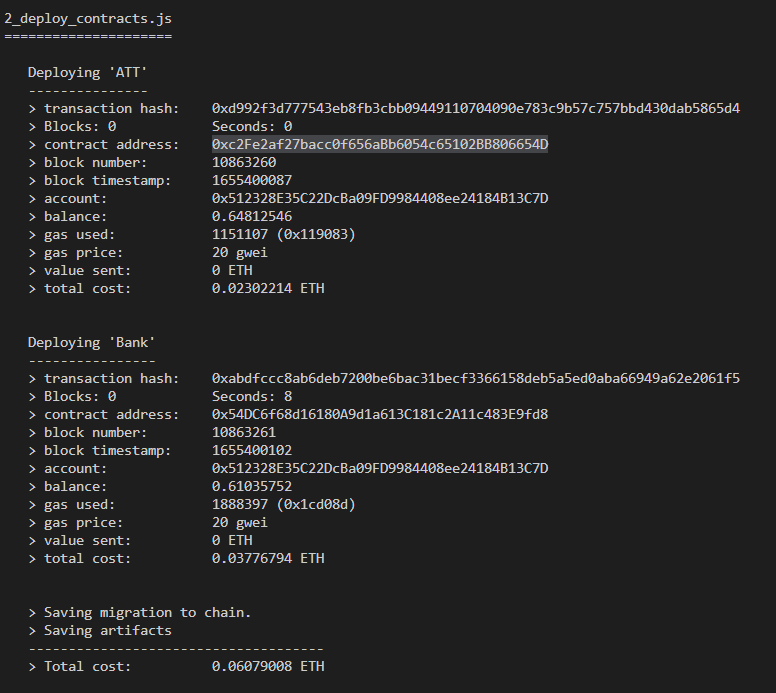
1. Unpack the ***/truffle & /solidity-app*** and run ***npm install*** inside both folders
2. Run Ganache local blockchain and set up a workspace
3. Go to the ***/truffle*** of the project source and run ***truffle migrate***. This will compile the contracts and deploy them on the Ganache chain.
4. Go to the ***/solidity-app*** in the project folder and open the ***.env*** file
5. In the ***.env*** file replace the written values with values corresponding to your Ganache chain/ or configure Ganache to set up these values to match the ones in the ***.env*** (the project was originally tested with variables already found in the ***.env***)
6. Along with the network variables, both the Token and Bank address should be entered as they have been deployed in ganache to ensure the proper functionality.
7. Also a symbolic link is needed so that *App.js* can access the contract abi, but React dosent allow any paths outside its src folder. So the create a symoblic link on Windows we run Powershell Admin and run the command {New-Item -ItemType SymbolicLink -Name contracts -Target C:\...\...\...\truffle\build\contracts} from the C:\...\...\solidit-app\src\
8. After running Ganache and setting up the ***.env***, run ***npm start*** in ***/solidity-app*** and the localhost page will open containing the app
9. Metamask will prop up, to ask for credentials
10. To test all the functionalities accounts should be imported from Ganache to Metamask using the ‘Import account’ and pasting its private key. First three accounts are required for the specified test case (as the first one is the owner of Token and Bank)
11. First account should transfer tokens to Account2 and Account3, 1000 and 4000 respectively.
12. Also the Account1, Account2 and Account 3 should give allowance to the bank address (which is displayed in the Token modal) corresponding with values they will deposit. Or they can give allowance to the max supply of token (10000000000000000000000).
13. Account1 then start savings by depositing the 1000 tokens of *pledgeAmount*.
14. Account2 and Account3 deposit their respective amounts.
15. Account2 withdraws tokens after *2R*
16. Account3 withdraws tokens after *3R*
17. Account1 withdraws remaining tokens and ends staking

# Rinkeby

In the ***/rinkeby*** folder you can find the same project, with modifications made so that it can work on the Rinkeby. To deploy to Rinkeby, correct details are needed as well as a Infura node. After creating the node, and chaning the details in the truffle-config to the following:

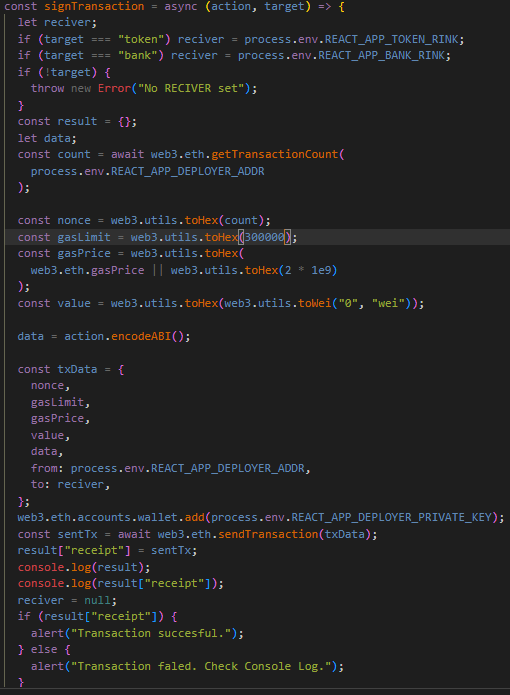


Next step is running *truffle migrate –rinkeby* after making sure enough rETH is available in the account. After compiling and running migrations, the contracts have been successfully deployed.



With the project successfully deployed it was time to make some changes to react front-end. Firstly in the .env file, we add the new environment variables to adapt our front-end. To the end of the file we can add the new Bank and Token addresses as well as the deployer/owner address and the deployer private key which we can get from metamask.

All of this info is needed since transactions need to be signed, instead of just sending them. For this purpose a new helper module is made “signTransactions.js”.



*SignTransaction* accepts two arguments: action to preform (which is the old method of calling contract on ganache with the *send()/call()* at the end deleted), and the target contract which should be either *‘token’* or *‘bank’.*

And with these changes in place we can run the *npm start* from the */solidity-app* directory and acces our fronend. Change the metamask network to Rinkeby and connect. After this we can interact with our contract as normal.

Currently both contracts are deployed on the Rinkeby and a staking period started on the bank contract.

