CMPE 483
PROJECT 1
GROUP REPORT

GROUP MEMBERS:
BURAK FERİT AKTAN - 2019400183
İBRAHİM FURKAN ÖZDEMİR - 2018400123
MUSTAFA ATAY - 2020400333

Faucet

Faucets is a function to distribute initial coin supply freely. There are some rules for taking a faucet.

- Any address can take at most one free coin.
- After initial supply many coins are distributed via faucet function, the function will not give any free coins.

These two rules are checked using “require” function of Solidity with form $require(condition,\ error\ message)$.

To prevent a user taking faucet multiple times, we implemented a mapping (named $tookfaucet$) from address to bool, storing addresses which have taken faucet. We required $tookfaucet[adress]$ to be false to take a faucet.

We had a variable called $tokens\_to\_mint$ storing have many tokens can ve give using this faucet. This variable starts with the initial supply value and decrements till 0. After this variable reaches 0, we no longer give coins.

$\_mint(address,\ amount)$ function of ERC20 is used to supply and send faucet coins.

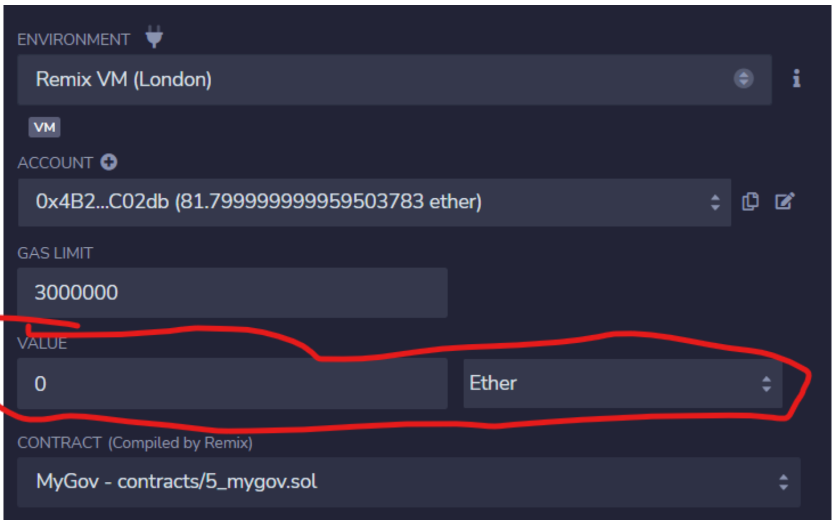
Donate Ether

$donateEther$ function allows people to donate to the system. In order to donate, there is no requirement to be a member (everyone can donate).

To implement this functionality we used $send$ function of Solidity in the given format:

$payable(address(this)).send(amount);$

When this line successfully executes, the system (address of the contract) receives the value sent with the function call.



ENVIRONMENT
Remix VM (London)
VM
ACCOUNT
0x4B2...C02db (81.799999999959503783 ether)
GAS LIMIT
3000000
VALUE
0 Ether
CONTRACT (Compiled by Remix)
MyGov - contracts/5_mygov.sol

The send function may return false in case it encounters a problem, in this case we return an error message (`"Failed to donate ethers”`) using $require$ function.

Note that $send$ function costs 2300 gas units.

To enable our contract to receive ethers, we made the constructor $payable$, and added $receive$ and $fallback$ functions to the contract.

Donating MyGov Tokens using DonateMyGov function
The DonateMyGov function is public and takes only one parameter: an unsigned integer that represents the number of tokens to be donated.

The function starts by calls the transfer function of ERC20 to transfer a certain amount of tokens from the caller's account to the contract's address. Then, the function checks the status (success or fail) using arequire statement. If the transfer was unsuccessful, the require statement throws an error message "Failed to donate tokens" and reverts.

Remember the definition of “user od MyGov”. An address is accepted as a user if and only if it has at least one MyGov token balance. If a user has 2 MyGov tokens and
It also has an if statement that checks the balance of the user who called the function. If their balance is 0, it decrements the userCount variable. This probably is related to how many users have tokens and can vote.

The overall purpose of this function is to allow the users to donate a specific amount of tokens to the contract address, which would be tracked in the donatedTokens variable, also it updates the user count if it founds out the user has no tokens.

Survey Functions

Submit Survey
"submitSurvey" is defined as public and payable. The function takes in 4 parameters:
1. A string memory variable called "ipfshash" that represents a hash for the IPFS file.
2. An unsigned integer variable called "surveydeadline" represents the deadline for taking the survey.
3. An unsigned integer variable called "numchoices" represents the number of choices in the survey.
4. An unsigned integer variable called "atmostchoice" represents the maximum number of choices a user can make in the survey.

Cost of submitting a survey: 2 MyGov Tokens and 0.04 Ethers

The function first charges the user 2 MyGov tokens using donateMyGov token function (no need to implement the same logic here again, calling donateMyGov here is causing cleaner code)
Then, there is a require statement that checks whether the value sent with the function call (i.e., msg.value) is equal to 4*10^16 wei (which is equivalent to 0.04 ether). If this condition is not met, the require statement throws an error message "you should pay 0.04 ethers".
Then the system checks whether survey deadline is passed or not by using a require statement. It checks whether the current block timestamp is greater than the "surveydeadline" parameter. If this condition is not met, the require statement throws an error message "deadline should be after current time".
Then, a new survey memory object is created and it's properties are set by the input arguments. surveys array is pushed the survey memory object.
Finally, the function returns the current length of the surveys array after adding the newly created survey to the surveys array (first created survey has id= 0, second created survey has id=1…). which should be the id of the survey which is submitted.
To sum up, the function submits a new survey, with IPFS hash, deadline, number of choices and maximum allowed choice, also with the 0.04 ether and 2 MyGov tokens as a fee. And it returns the survey id.

TakeSurvey
The function takes in 2 parameters: unsigned integer "surveyid" that representing the ID of the survey to be taken, array of unsigned integers "choices" representing the choices selected by the user in the survey.
The function checks:
* whether the msg.sender is a member of the DAO or not, using the balanceOf function. If the user doesn't have any tokens(balance is 0), the require statement throws an error message "you aren't a member".
* whether the survey id passed in the input exists or not (logically, a user can’t vote a non-existing survey). If the survey with that id doesn't exist, the require statement throws an error message "survey with given id doesn't exist"
* whether the number of choices passed in the input is less than or equal to the maximum number of choices allowed for that survey. If the number of selected choices is greater than the maximum allowed choices, the require statement throws an error message "you selected so many choices"
For each choice, the function checks whether the choice is a valid choice for the survey by checking whether the choice is less than the number of choices in the survey, if not, the require statement throws an error message "choices should be valid numbers". At the end,

function increments the choices in the survey and increments the number of taken for that survey.
To sum up, the function allows users to take a survey by providing their survey ID and selected choices.

View Functions related to Surveys

getSurveyResults: takes an input "surveyid" (id of the survey) and returns two outputs number of user who took the survey and survey results.
The function starts by checking whether the surveyid exists, if not it throws an error message "survey with given id doesn't exist" . If the surveyid exists, returns the number of taken survey and the choices of that survey.
getSurveyInfo: takes an input "surveyid"(id of the survey) and returns four outputs "ipfshash", "surveydeadline", "numchoices" and "atmostchoice" .It starts by checking whether the surveyid exists, if not it throws an error message "survey with given id doesn't exist" . If the surveyid exists, returns the IPFS hash of the survey, survey's deadline, number of choices and atmost choice of that survey.
getSurveyOwner: takes an input "surveyid"(id of the survey) and returns address of the owner of the survey: "surveyowner" . It starts by checking whether the surveyid exists, if not throws an error message "survey with given id doesn't exist" . If surveyid exists, returns the owner of the survey.
getNoOfSurveys: takes no input, returns the total number of surveys submitted to the system (i.e., length of the surveys array)

Project Functions

Submit Project Proposal

<u>Cost of submitting a project proposal: 1 MyGov Token and 0.01 Ethers</u>
“submitProjectProposal” function allows users to submit a project proposal to the DAO. It takes four input parameters:
1) string "ipfshash" representing the IPFS hash of the project data.
2) uint "votedeadline" representing the deadline for voting on the project.
3) array of uints "paymentamounts" representing the payment amounts for the project.
4) array of uints "payschedule" representing the payment schedule for the project.

The function first checks if the proposal's vote deadline has passed using the "now" variable which represent the current block timestamp, and throws an error message "deadline is already passed" if the vote deadline has passed.
Remember, submitting a project proposalhas a cost, therefore the function calls donateMyGovToken to charge the user and requires the user to pay 0.01 Ether(if msg.value is different than 0.01 ethers, the function reverts and gives an error message "you should pay 0.01 ethers")
The function assigns a unique ID to the project by incrementing the global "projectId" variable and storing the ID in the "projectid" variable. (first proposed project has id= 0, second proposed project has id=1…)
The function then then stores the project data in the global projects array, including the IPFS hash, vote deadline, payment amounts, payment schedule, and the project's owner address. Finally, it returns the project's ID to the user.
delegateVoteTo
"delegateVoteTo function allows MyGov members to delegate their vote to other members. It takes two input parameters: address "memberaddr" representing the address of the member who the vote is being delegated to, and uint "projectid" representing the id of the project the vote is being cast on.
The function checks:
* whether the senders trying delegate themselves using an require statement and throws an error message "you can't delegate yourself".
* whether the msg.sender is a member or not by checking their token balance using the balanceOf function. If user does not have any tokens, the require statement throws an error message "in order to delegate vote, you should be a member".
* whether the msg.sender already voted or not, if yes throws an error message "you already voted"
There shouldn’t be any loop in delegation chains. The system assigns the delegated to address to the delegating_to variable of the voter struct and then checks for loop by following the delegating_to pointers till it reaches to the end of the chain or comes to the initial voter (reaching to the initial voter mean a loop exists). If a loop is detected the function throws an error message "loop detected".
The function increments the power of the person the vote was delegated to, If the delegated person already voted, it increments the yesCount of the proposal if the delegate's vote is yes. If the delegate has not voted yet it increases the power of the delegate
VoteForProjectProposal
"voteForProjectProposal" function allows users to vote on a project proposal. It takes two input parameters: uint "projectid" representing the id of the project to be voted on and bool

"choice" representing the user's vote choice.
The function starts by checking the voting deadline for the project has passed or not by comparing the votedeadline to now, if it has passed throws an error message "voting deadline is passed".
Note that to vote for a project proposal, the user must be a member, therefore the system checks msg.sender’s token balance using the balanceOf function. If user does not have any tokens (i.e., not a member), the require statement throws an error message "in order to vote, you should be a member".
We don’t want a user to vote more than once, therefore the system checks if msg.sender has already voted or not by checking the did_vote status of the voter (defined in the Voter struct), If the user is already voted throws an error message "you already have voted, you can't vote again". Note that users delegated their vote, couldn’t vote since the system checks users as voted when they delegate their votes.
Before continue let’s define “power of a voter”: As default, each voter has a power of one, however, the power changes with vote delegations
The function then checks if the voter has any power or not, if not it assigns 1 to the power of the voter (why: Solidity initializes variables with 0, however the default voting power is 1)
It then assigns the vote choice of the user who called the function to the voter's choice variable, sets the voter's did_vote variable to true, and increments the project's yesCount by the power of the voter if the vote is "yes".

voteForProjectPayment

"voteForProjectPayment" function allows MyGov members to vote on payments of a project. It takes two input parameters: uint "projectid" representing the id of the project being voted on and bool "choice" representing the user's vote choice.
The function first finds the current payment deadline.
Then it checks:
* whether the payment deadline is passed or not, if it has passed throws an error message "voting deadline is passed".
* whether the msg.sender is a member or not. If user does not have any tokens, the require statement throws an error message "in order to vote, you should be a member".
* whether the msg.sender has already voted or not by checking the did_vote status of the voter, If user already voted throws an error message "you already have voted, you can't vote again".
It then assigns the vote choice of the user who called the function to the voter's choice variable, sets the voter's did_vote variable to true, and increments the project's yesCounts_payment[] of the current ongoing payment if the user voted "yes".

reserveProjectGrants
reserveProjectGrant function allows the owner of a project proposal to reserve the funding by the deadline of the proposal. The function takes one input parameter: uint "projectid" representing the id of the project which the funding is being reserved for.
The function checks:
* whether the project as already been reserved or not using the 'reserved' variable of the project. If the project has already been reserved, it throws an error message "can not reserve more than once."
* whether the project proposal has received enough votes by comparing the yesCount of the project with the userCount and a minimum required ratio of 10%, if not it throws an error message "not enough votes."
* whether the balance of the contract minus the reservedWei is greater than or equal to total_payment. If there is not sufficient ether in MyGov contract when trying to reserve, it throws an error message "not enough ethers."
Finally, it increments the reservedWei with the total payment amount, sets the 'reserved' variable of the project to true and increments the number of funded projects.
withdrawProjectPayment
This function is used to withdraw funds for a specific project after a successful vote on its payment schedule.
The function checks:
* whether yes counts are greater than or equal to 1/100 of the users, if not gives an error message**:** "not enough vote for withdrawing (payschedule)"
* whether the reserves are enough yo make the payment or not, if not gives an error message: “not enough reserve”
Then the function sends the payment amount using transfer function of Solidity

View Functions related to Projects
getProjectOwner(uint projectid): returns the owner of the project of the given id
getProjectInfo(uint projectid): returns the details of the project proposal with the given id, like the IPFS hash, voting deadline, payment amount, payment schedule, ongoing payment index, the yes vote count for each payment schedule
getProjectNextPayment(uint projectid): returns the timestamp of the next payment schedule for the project of given id, assumes that the project is funded.
getNoOfFundedProjects(): returns the number of funded projects
getIsProjectFunded(uint projectid): returns a boolean indicating whether the given project is funded or not
getEtherReceivedByProject(uint projectid): returns the total ether received by the given

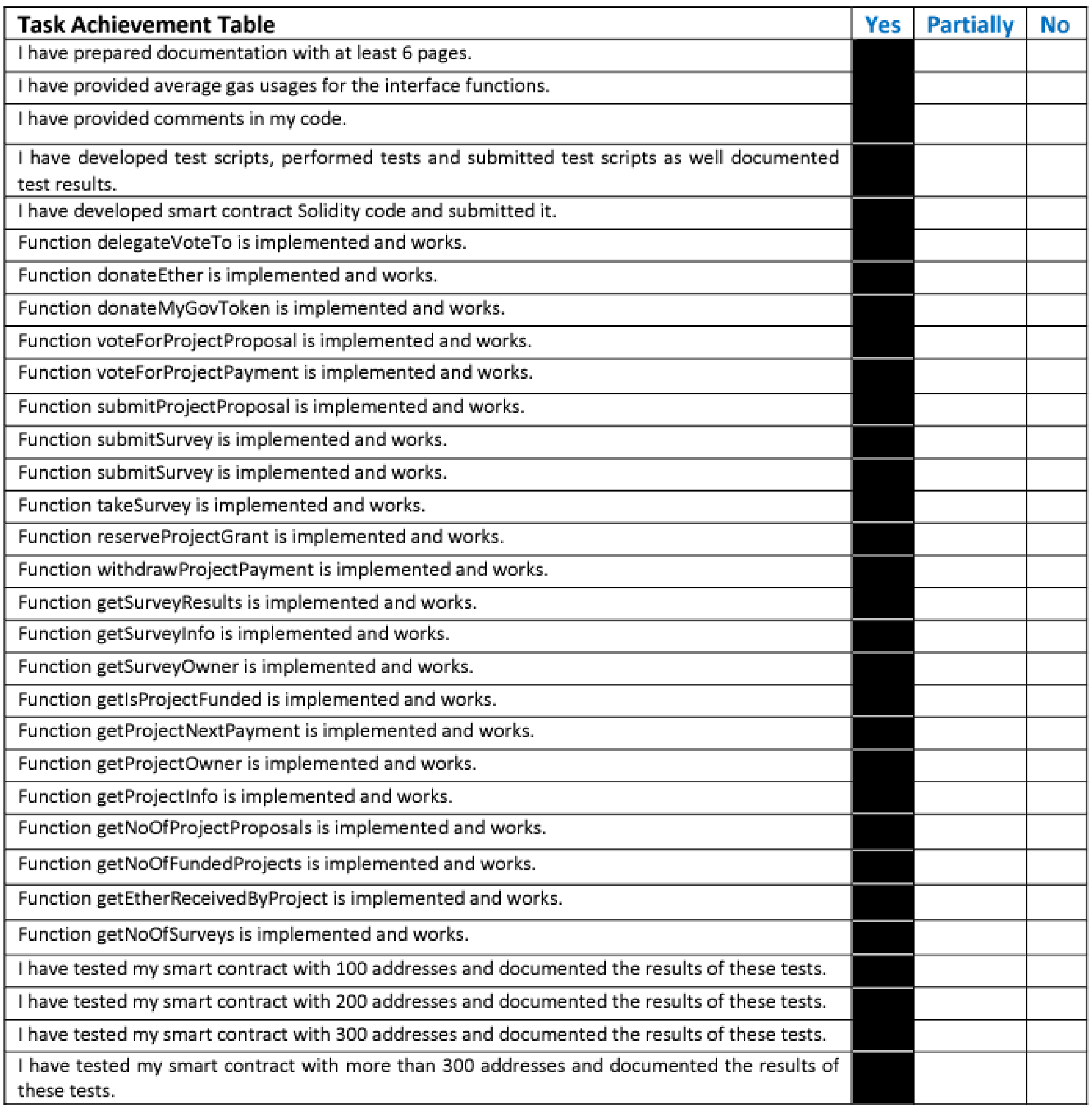
project

getNoOfProjectProposals(): returns the total number of project proposals

Average Gas Usages

Function Name	Average Gas Usage
constructor	**5554026 gas**
delegateVoteTo	**68856 gas**
donateEther	**30418 Gas**
donateMyGovToken	**85120 Gas**
voteForProjectProposal	**93218 gas**
voteForProjectPayment	**82073 gas**
submitProjectProposal	**250146 gas**
submitSurvey	**212718 gas**
takeSurvey	**58486 gas**
reserveProjectGrant	**57221 gas**
withdrawProjectPayment	**109125 gas**
getSurveyResults	**0**
getSurveyInfo	**0**
getSurveyOwner	**0**
getIsProjectFunded	**0**
getProjectNextPayment	**0**
getProjectOwner	**0**
getProjectInfo	**0**
getNoOfProjectProposals	**0**
getNoOfFundedProjects	**0**
getEtherReceivedByProject	**0**
getNoOfSurveys	**0**
faucet	**96377 gas**

TASK ACHIEVEMENT TABLE



Task Achievement Table	Yes	Partially	No
I have prepared documentation with at least 6 pages.	■		
I have provided average gas usages for the interface functions.	■		
I have provided comments in my code.	■		
I have developed test scripts, performed tests and submitted test scripts as well documented test results.	■		
I have developed smart contract Solidity code and submitted it.	■		
Function delegateVoteTo is implemented and works.	■		
Function donateEther is implemented and works.	■		
Function donateMyGovToken is implemented and works.	■		
Function voteForProjectProposal is implemented and works.	■		
Function voteForProjectPayment is implemented and works.	■		
Function submitProjectProposal is implemented and works.	■		
Function submitSurvey is implemented and works.	■		
Function submitSurvey is implemented and works.	■		
Function takeSurvey is implemented and works.	■		
Function reserveProjectGrant is implemented and works.	■		
Function withdrawProjectPayment is implemented and works.	■		
Function getSurveyResults is implemented and works.	■		
Function getSurveyInfo is implemented and works.	■		
Function getSurveyOwner is implemented and works.	■		
Function getIsProjectFunded is implemented and works.	■		
Function getProjectNextPayment is implemented and works.	■		
Function getProjectOwner is implemented and works.	■		
Function getProjectInfo is implemented and works.	■		
Function getNoOfProjectProposals is implemented and works.	■		
Function getNoOfFundedProjects is implemented and works.	■		
Function getEtherReceivedByProject is implemented and works.	■		
Function getNoOfSurveys is implemented and works.	■		
I have tested my smart contract with 100 addresses and documented the results of these tests.	■		
I have tested my smart contract with 200 addresses and documented the results of these tests.	■		
I have tested my smart contract with 300 addresses and documented the results of these tests.	■		
I have tested my smart contract with more than 300 addresses and documented the results of these tests.	■		

TESTS

Our test starts with making sure initially all users' balance equals to 0

After that %90 of users calls faucet function and we check if %90 of them have 1 mygov and rest of them still has no token.

Then first %10 of users tries to call faucet again but it reverts since they already called faucet

Then we again check to see if %90 of them has 1 token and rest of them has no token.

Then some users tries to take a survey that doesn’t exist and we make sure it reverts with appropriate failure message.

Then user21 and user22 sends token to user20 which then submits a survey.

Then we getSurveyOwner of recently created survey and make sure owner equals to user20

Then we getSurveyInfo of same survey and check if they equal to given data by testing

Then we check for total no of surveys to equal to 1

Then again some users try to take survey and this time survey exists, 4 of them votes option1, 1 of them votes option0

Then we check total no of votes, votes for option1, votes for option0.

Then 5 users send mygov to user7

User7 submits a projectProposal

We make sure owner of proposal equals to user7, no of proposals equals to 1 and getProjectInfo for project0 and make sure data equals to data given by us.

At this point project is not funded yet is no voting has been done. We check to see getProjectNextPayment raises an error and reverts.

Then %90 of users tries to vote yes for proposal, some of these users have no token left at this point so they cannot vote and reverts due to error but other users vote yes for proposal.

After that user0 tries to reserveProjectGrant for project0 but since he is not owner he cannot.

User7 tries to reserveProjectGrant for project0 and successfully reserves.

Then %50 of users try to vote for nextProjectPayment and again some of them are not members due to not having any mygov so they fail and revert but other users vote yes for next payment.

Then we withdraw project payment and get next project payment.

Results: All the tests worked successfully for 100, 200, 300, 350 users.