Governance Token

CMPE 483 Blockchain Programming

Homework 1 Report

Fall 2022

Altay Acar - 2018400084

Leyla Yayladere - 2018400216

Umut Deniz Şener - 2018400225

Introduction

This project provides a custom implementation of a governance token, named MyGov, which inherits OpenZeppelin's ERC20 fungible token standard. Decentralized autonomous organizations (DAOs) reside in the heart of the governance tokens. A decentralized autonomous organization (DAO) is a unique type of a collectively managed organization that has no central governing body and whose members share a common goal to act in the best interest of the entity. DAO works as participants gain voting power or a say in the decision making process in exchange for investing their own money. Thus, a token holder has the right to vote for the future of a project, while people without the governance token not.

Definition

MyGov token defines a DAO structure, in which there are two types of users: members and non-members. Members and non-members differentiate by holding at least one *MyGov* token. So, when a user has at least one *MyGov* token in their cryptocurrency portfolio, then they become a member of this DAO. Every user can donate money (in Ether or in *MyGov* token) to the DAO. But only members can take other actions regarding the interfaces defined in the smart contract.

MyGov tokens are distributed via faucet interface. Each member can receive *MyGov* token from the faucet interface only once, and become a member. Since *MyGov* token inherits ERC20 standards, users also can receive *MyGov* tokens from a cryptocurrency exchange, if that exchange offers the token, or transfer *MyGov* tokens between accounts, or even sell *MyGov* tokens to an exchange, if that exchange accepts the token.

Holding *MyGov* tokens makes users members of DAO and offers a variety of interfaces in the scope of the DAO: they can submit project proposals, raise funding to their projects, vote for any project proposal submitted in the contract, or create and take surveys across the contract. Submitting a project proposal or a survey comes with a cost, and that fee is added to the contract's balance, which will be awarded to the project proposals that successfully granted project payment.

Projects and surveys obey a strict deadline specified by the creators of them. Members cannot vote for a project proposal or take a survey after their respective deadlines are passed. Thus, for the project proposals only after the deadline is passed, payment protocols are initiated. When a project proposal gets a yes vote from at least ten percent of the users by its voting deadline, the owner of the proposal can initiate payment receive protocols. To receive payment according to the payment schedule, the owner must reserve payment from the contract's account and needs to have at least one percent of the members to say yes for receiving the payment.

With all of the interfaces and features defined above, our implementation of the governance token *MyGov* provides a reliable DAO system, where users can become members and have a say on the projects to ensure the projects' robust development.

Requirements

1. All standard ERC20 functions should be provided by *MyGov*. Additionally, interface functions given in Table 1 should be implemented and provided.
2. *MyGov* will provide three transactional activities:
    (i) Donation to *MyGov*
    (ii) Survey service and
    (iii) Project Proposal, Voting and Funding.
3. Anyone who owns at least 1 *MyGov* token is a *MyGov* member.
4. Members can carry out activities (i) (ii), (iii).
5. Anyone can carry out activity (i), i.e. you do not need to be a member to carry out (i).
6. Anyone can call get information functions. View functions are open to everyone, i.e. you do not need to be a member to get information.
7. Submitting Project Proposal costs 5 *MyGov* tokens and 0.1 Ether.
8. Submitting Survey costs 2 *MyGov* tokens and 0.04 Ether.
9. *MyGov* token supply is 10 million (fixed).
10. Donations can be accepted in ethers and *MyGov* tokens only. Ethers can be granted to winning Project proposals. *MyGov* tokens are given away via faucet function.
11. *MyGov* tokens are distributed via a faucet function. Faucet gives 1 token to an address. If the address obtained a token before, it cannot get a token from the faucet any longer.
12. Each member can give 1 vote for each proposal. Members can delegate vote. Members who voted or delegated vote cannot reduce their *MyGov* balance to zero until the voting deadlines.
13. In order to get a Project proposal funded, at least 1/10 of the members must vote yes and there should be sufficient ether amount in the *MyGov* contract. Project proposers must call the reserveProjectGrant function in order to reserve the funding by the proposal deadline. If the project proposer does not reserve by the deadline, funding is lost. Also, if there is not sufficient ether in *MyGov* contract when trying to reserve, funding is lost. In order to receive payments according to the schedule, at least 1/100 of the members should vote yes. If at least 1/100 do not vote yes, then the current as well as the remaining payments are terminated, i.e. the project is no longer funded.

Table 1
constructor(uint tokensupply)
function delegateVoteTo(address memberaddr, uint projectid) public
function donateEther() public
function donateMyGovToken(uint amount) public
function voteForProjectProposal(uint projectid, bool choice) public
function submitProjectProposal(string ipfshash, uint votedeadline,uint [] paymentamounts, uint [] payschedule) public returns (uint projectid)
function submitSurvey(string ipfshash, uint surveydeadline, uint numchoices, uint atmostchoice) public returns (uint surveyid)
function takeSurvey(uint surveyid, uint [] choices) public
function reserveProjectGrant(uint projectid) public

function withdrawProjectPayment(uint projectid) public
function getSurveyResults(uint surveyid) public view returns(uint numtaken, uint [] results)
function getSurveyInfo(uint surveyid) public view returns(string ipfshash, uint surveydeadline, uint numchoices, uint atmostchoice)
function getSurveyOwner(uint surveyid) public view returns(address surveyowner)
function getIsProjectFunded(uint projectid) public view returns(bool funded)
function getProjectNextPayment(uint projectid) public view returns(int next)
function getProjectOwner(uint projectid) public view returns(address projectowner)
function getProjectInfo(uint activityid) public view returns(string ipfshash, uint votedeadline, uint [] paymentamounts, uint [] payschedule)
function getNoOfProjectProposals() public view returns(uint numproposals)
function getNoOfFundedProjects () public view returns(uint numfunded)
function getEtherReceivedByProject (uint projectid) public view returns(uint amount)
function getNoOfSurveys() public view returns(uint numsurveys)

Interfaces

struct User

User struct to keep user related specific data with the help of struct variables and mappings. The data we need for each user in order to implement functions in the contract consist of *use_faucet*: to give only 1 MyGov to each user, *vote_weight*: to keep delegation information, *voted_projects*: to keep voting information for project proposals, *voted_payments*: to keep voting information for project payments, *vote_choices*: to keep choice information of voting for project proposals for the purpose of delegating vote to already voted user, and *taken_surveys*: to restrict user from taking a survey only once.

struct Project

Project struct to keep project related specific data with the help of struct variables.The data we need for each project in order to implement functions in the contract consist of owner address of the project, project id, yes vote count, funding status of the project, start date of the project, project deadline, end date of the project, ipfshash string, payment amounts and pay schedules, payment votes, reserved payments, payment termination status, next payment to reserve, next payment to withdraw and received ether amount.

struct Survey

Survey struct to keep survey related specific data with the help of struct variables. The data we need for each survey in order to implement functions in the contract consist of owner address of the survey, survey id, number of options in the survey, the maximum number of options that can be chosen by survey participant, start date of the survey,

survey deadline, end date of the survey, results of the survey for each option, and ipfshash string.

function faucet() public payable returns(bool)

Faucet function transfers one MyGov token from the contract to the user that calls the function. To receive MyGov token from the faucet function, there should be enough MyGov tokens in the contract's token supply, which is initially 10 million. Every user can use faucet function only once and as soon as they use faucet function and gain one MyGov token, they become a member and gain vote weight for the project proposals available in the contract.

function customFaucet(uint amount) public payable

This function is written only for testing purposes. It should be called only by test functions. To be able to submit a project proposal or a survey, a user needs more than 5 or 2 MyGov tokens respectively. However, it's only possible to give 1 MyGov to the user by faucet functions only for once. Users can exchange MyGov tokens in other platforms and thus have more than 1 MyGov token, whereas we can't mimic this exchange in our test functions. Thus, we give MyGov tokens in the amount of the parameter by calling this function in our test functions.

function delegateVoteTo(address memberaddr, uint projectid) public

By calling this function a user distributes their vote weight for a specific project proposal to another user, namely delegatee. To delegate vote, both the sender and receiver users must be members, sender has to have a voting right (vote weight) for that project proposal and also must not have voted for the project proposal. As logically, a user cannot delegate vote to themselves. When all preconditions are met, the delegated vote affects the delegatee's vote weight, if they have not voted yet. On the other hand, if the delegatee has already voted, the delegated vote is counted as the same vote for that project. For example, when a vote is delegated to a member, who has already voted yes for a project, the delegated vote is also counted as yes.

function donateEther() public payable

A user donates Ether to the contract by calling this function. Since this function is public and payable, every user can donate Ether to the contract by specifying the Ether amount in the value field of the transaction, as long as the donated amount is greater than zero.

function donateMyGovToken(uint amount) public payable

A user donates MyGov token to the contract by calling this function. Since this function is public and payable, every user can donate MyGov tokens to the contract by specifying the amount, as long as they have enough MyGov tokens in their balance to donate. However, since a member who has voted or delegated vote for a project proposal cannot reduce their MyGov balance to zero until the voting deadlines, if the donation would reduce a user who voted or delegated vote MyGov balance to zero, then they cannot donate MyGov tokens to the contract. Also, when a user successfully donates MyGov tokens to the contract, and this transaction reduces their MyGov balance to 0, then they become non-member.

function voteForProjectProposal(uint projectid, bool choice) public

A user votes for an available project proposal in the contract by using this function. To vote for a project proposal, the user must be a member and also not voted for that project proposal already. They also cannot vote for a non-existing project proposal or a project proposal, whose deadline is already passed. The voting logic of our implementation works as described: Every user is counted as voted for no by default. So, when a user has not voted yes for a project proposal, then it is counted as no. But, their vote weight remains the same, since they can decide to vote for a project proposal. Thus, in the implementation only yes votes are counted. When a user is voted for a project, either a yes vote or a no vote, their choice is saved and the project is marked as voted for them. Since the condition of 1/10 members must be voted yes for a project proposal to be funded is strongly dependent on the yes vote count, everytime when a user votes for a project, the yes count rate is calculated and it satisfies the 1/10 condition, then the project is marked as funded, and voting deadline condition will be checked later on.

function voteForProjectPayment(uint projectid, bool choice) public

A member votes for a project payment by calling this function. They cannot vote for a non-existing project's payment or for a project. The payment system of the project proposals work as follows: When a project has gathered a yes vote from at least ten percent of the members by the project deadline, then it is counted as funded. To receive payments from the payment schedule specified by the project proposal, the owner of the proposal must reserve the specified payment amount by the schedule of that payment and also one percent of the members must vote yes for the next scheduled payment. If there is not enough Ether in the contract for the reservation or the owner does not reserve it by the schedule or the payment cannot gather enough yes votes to meet the 1/100 condition, then the project payment is terminated and although there are still payments scheduled, the project cannot receive any payment. So, a member can also not vote for a terminated project's payment. The same voting logic applies to the votes for the payment as well: Only yes votes are counted and the default vote is no. When all the conditions specified above are met, the member votes for a project payment and the payment is counted as voted for that member. Their vote is also stored in the yes vote count in the project's payment votes.

function submitProjectProposal(string memory ipfshash, uint votedeadline, uint[] memory paymentamounts, uint[] memory payschedule) public payable returns (uint projectid)

A member can submit a project proposal by calling this function. To submit a project proposal, the member needs to specify parameters of the function correctly according to the implementation guidelines: Since a project proposal will be uploaded to IPFS, the returned hash value must be passed to the project as ipfshash parameter. Also, the user needs to specify a vote deadline in days. So, when the votedeadline is passed as 1, then the vote deadline would be 1 day after the proposal's successful submission. paymentamounts is an array of payment amounts that the project proposal intends to receive after the voting deadline and it needs to be the same length as the payschedule array, since there needs to be exactly one schedule for each payment. The payment schedule specification works the same as the voting deadline: it is given in days. But the schedule needs to be given in ascending order, since for example the first payment cannot receive payment before the second payment. Thus, the payments specified in

the paymentamounts are also ordered, the first amount in the array is also for the first payment. As logically, a specified payment in the paymentamounts cannot be zero. Submitting a project proposal also comes with a cost by itself: By submitting a project proposal, a member pays 5 MyGov tokens and 0.1 Ether to the contract. So, there should be enough MyGov tokens and Ether in their balance to submit a project proposal. For a member it is also possible to reduce their MyGov balance to zero by submitting a project proposal. So, if they are already voted or delegated vote for any project proposals, they need to have at least 6 MyGov tokens in their balance to submit the proposal so that they remain member after submission. When a project proposal is successfully submitted, every member will gain a vote weight for that proposal and the project id is returned.

function submitSurvey(string memory ipfshash, uint surveydeadline, uint numchoices, uint atmostchoice) public payable returns (uint surveyid)

A member can submit a survey by calling this function. To submit a survey the member needs to specify parameters of the function correctly according to the implementation guidelines: Since a survey will be uploaded to IPFS, the returned hash value must be passed to the survey as ipfshash parameter. Also, the user needs to specify a survey deadline in days. So, when the surveydeadline is passed as 1, then the survey deadline would be 1 day after the survey's successful submission. Users also need to specify a number of choices and a maximum choice amount when submitting a survey. Each choice will have the index starting from 0 ending at numchoices. For example, when a user specifies numchoices as 5, then the choices will be indexed as 0, 1, 2, 3, and 4. Submitting a survey also comes with a cost by itself: By submitting a survey, a member pays 2 MyGov tokens and 0.04 Ether to the contract. So, there should be enough MyGov tokens and Ether in their balance to submit a survey. For a member it is also possible to reduce their MyGov balance to zero by submitting a survey. So, if they are already voted or delegated vote for any project proposals, they need to have at least 3 MyGov tokens in their balance to submit the survey so that they remain member after submission. When a survey is successfully submitted the survey id is returned.

function takeSurvey(uint surveyid, uint[] memory choices) public

A member takes a survey by calling this function. A survey can only be taken before its deadline and every member can take a survey only once. Also, users cannot take a non-existing survey. Survey choices logic is as follows: Each survey has a certain amount of choice options, which is specified by the survey owner by submission. All these choice options are indexed from 0 and an array of results is created upon survey submission. The results array is also indexed according to the survey. So, the first element in the results array holds the result value of the first choice option. When a member takes a survey and specifies a choice, its result value is incremented, i.e. the choice is chosen by one member. Thus, to take a survey, a user needs to provide an array of choices that they would like to choose among the available ones. To that extent, they cannot give duplicate choices. Each choice can only be chosen once. Also, there is a limitation on the amount of choices a user can make, which is also specified by the survey owner upon submission. Thus, when a user takes a survey, they cannot exceed the maximum choice amount. If a user successfully takes a survey, the survey is marked as taken for that user and the total take amount for that survey is incremented.

function reserveProjectGrant(uint projectid) public

Only the owner of the project can reserve project payment by calling this function. Project payments can be reserved one by one and sequentially if the project is funded and there is enough ether for the reserved payment in the contract. This function can't be called if voting for the project proposal still continues. Voting deadline should be passed, at least 1/10 of the members voted yes and there should be enough ether in the contract in order to reserve payment. Also, to reserve payment, the deadline for that payment should be also passed and 1/100 of the members voted yes. If one of the previous payments couldn't be reserved, the project is no longer funded. Thus remaining payments are not allowed to be reserved. We checked this with project_terminated variable of the Project struct. We keep the next payment to be reserved with reserve_payment variable of the Project struct. This variable starts with 1 and increments when the reserve operation is successful, therefore we implement one by one and sequential reserving for the payments (i.e in the owner's first call of this function, payment 1 is reserved; in second call, payment 2 is reserved ...) If all payments are already reserved, calling this function will throw "All payments have been already reserved" warning.

function withdrawProjectPayment(uint projectid) public

Only the owner of the project can withdraw project payment by calling this function. Payment should be reserved first before it can be withdrawn. If the project is no longer funded due to having less 'yes' vote than 1/100 of the members or having less ether in the contract than the payment amount, calling this function will throw "Payments for this project have been terminated" warning. We keep the next payment to be withdrawn with withdraw_payment variable of the Project struct. This variable starts with 1 and increments when the withdrawal operation is successful, therefore we implement one by one and sequential withdrawal of the payments (i.e in the owner's first call of this function, payment 1 is withdrawn; in second call, payment 2 is withdrawn ...) If all payments are already withdrawn, calling this function will throw "All payments have been already withdrawn" warning.

function getSurveyResults(uint surveyid) public view returns(uint numtaken, uint[] memory results)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the results of the survey, which is an array of integers. Each element in the array corresponds to the selection amount of the choice with the same index in the survey. For example, when the third element in the results array is 24, it means that the third choice is selected 24 times. This function does not depend on the survey deadline, the current results of the survey can be seen anytime.

function getSurveyInfo(uint surveyid) public view returns(string memory ipfshash, uint surveydeadline, uint numchoices, uint atmostchoice)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the general information about a survey, such as what is the IPFS hash number of the survey, what is the survey deadline (in days) , what is the total number of choices and the maximum number of choices a user can make by taking the survey.

function getSurveyOwner(uint surveyid) public view returns(address surveyowner)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the address of the user who owns (the one who has submitted the survey) the specified survey.

function getIsProjectFunded(uint projectid) public view returns(bool funded)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get if a project proposal is funded or not. Since, a project is funded only if it can gather enough yes votes to satisfy 1/10 condition by the voting deadline, this function does not depend on the voting deadline as well. When the condition is met, the project becomes funded and this function returns true, otherwise false.

function getProjectNextPayment(uint projectid) public view returns(uint next)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the next payment scheduled.

function getProjectOwner(uint projectid) public view returns (address projectowner)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the address of the user who owns (the one who has submitted the project proposal) the specified project.

function getProjectInfo(uint projectid) public view returns(string memory ipfshash, uint votedeadline, uint[] memory paymentamounts, uint[] memory payschedule)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the general information about a project, such as what is the IPFS hash number of the survey, what is the voting deadline in days, what is the payment amounts planned to be received, and payment schedule in ascending order.

function getNoOfProjectProposals() public view returns (uint numproposals)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the total number of project proposals available in the contract, whose value is stored as the last project index number, i.e. its id.

function getNoOfFundedProjects () public view returns(uint numfunded)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the total number of funded projects, i.e. the projects that have satisfied the 1/10 condition by its deadline.

function getEtherReceivedByProject (uint projectid) public view returns(uint amount)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the total amount of Ether a project has received.

function getNoOfSurveys() public view returns(uint numsurveys)

This is a getter function and it does not alter the state variables defined in the contract. So, it is implemented as a view function. By calling this function, a user can get the total number of surveys available in the contract, whose value is stored as the last survey index number, i.e. its id.

function getVoteWeight(address voter, uint project_id) public view returns(uint weight)

Generating getters that will return big amounts of data in one go would be very expensive and might be even unusable. Therefore, it's avoided. Arrays and mappings of struct are not returned by value from getters on purpose. But we needed to see values in some mappings of structs in our tests, therefore we implemented our own getter functions.

function getCurrentPaymentVote(uint projectid) public view returns(uint payment)

Generating getters that will return big amounts of data in one go would be very expensive and might be even unusable. Therefore, it's avoided. Arrays and mappings of struct are not returned by value from getters on purpose. But we needed to see values in some mappings of structs in our tests, therefore we implemented our own getter functions.

function getTakenSurveys(address user_addr, uint survey_id) public view returns(bool isTaken)

Generating getters that will return big amounts of data in one go would be very expensive and might be even unusable. Therefore, it's avoided. Arrays and mappings of struct are not returned by value from getters on purpose. But we needed to see values in some mappings of structs in our tests, therefore we implemented our own getter functions.

Implementation

We have implemented the MyGov token's smart contract using the solidity, high-level object oriented language with version 0.8.17, which is the latest version. For the ERC20

standardization and all standard ERC20 functions we have imported OpenZeppelin's ERC20 implementation. We have developed our code using Remix IDE, since deploying a contract on Remix VM provides great accessibility. During the implementation we have continuously deployed our contract and used proxy test users provided by the Remix VM to test our newly implemented interfaces. Remix IDE also provides easy-to-use user interface for deployed contract's functions, both ERC20 and custom implemented ones.

While implementing the interfaces one by one, we have also provided detailed comments suitable for the general commenting guidelines of solidity. Each interface has an explanatory comment just above its implementation and each explanatory comment has fields of @dev, @param, and @return, explaining the general functionality of the interface, what it takes as parameter, and what it returns after execution respectively.

After we have finished with the implementation of our contract using Remix IDE, we have provided test cases using Hardhat Netwok's testing interface. It is a local Ethereum network designed for development. It comes built-in with Hardhat, and it's used as the default network. So, we didn't need to set up anything to use it, which made things easier for us to test our contract's interfaces. Details about the testing are outlined in the testing section of the report.

Testing

We set up a **Hardhat** Ethereum development environment in order to create test cases and to test them with 100s of accounts with Hardhat Network and Hardhat Chai Matchers. Hardhat Network is a local Ethereum network node designed for development. It allowed us to deploy our MyGov contract, run our tests and debug our code within the limits of our local machine. Hardhat Chai Matchers adds Ethereum-specific capabilities to the Chai assertion library, making our smart contract tests easy to write and read.

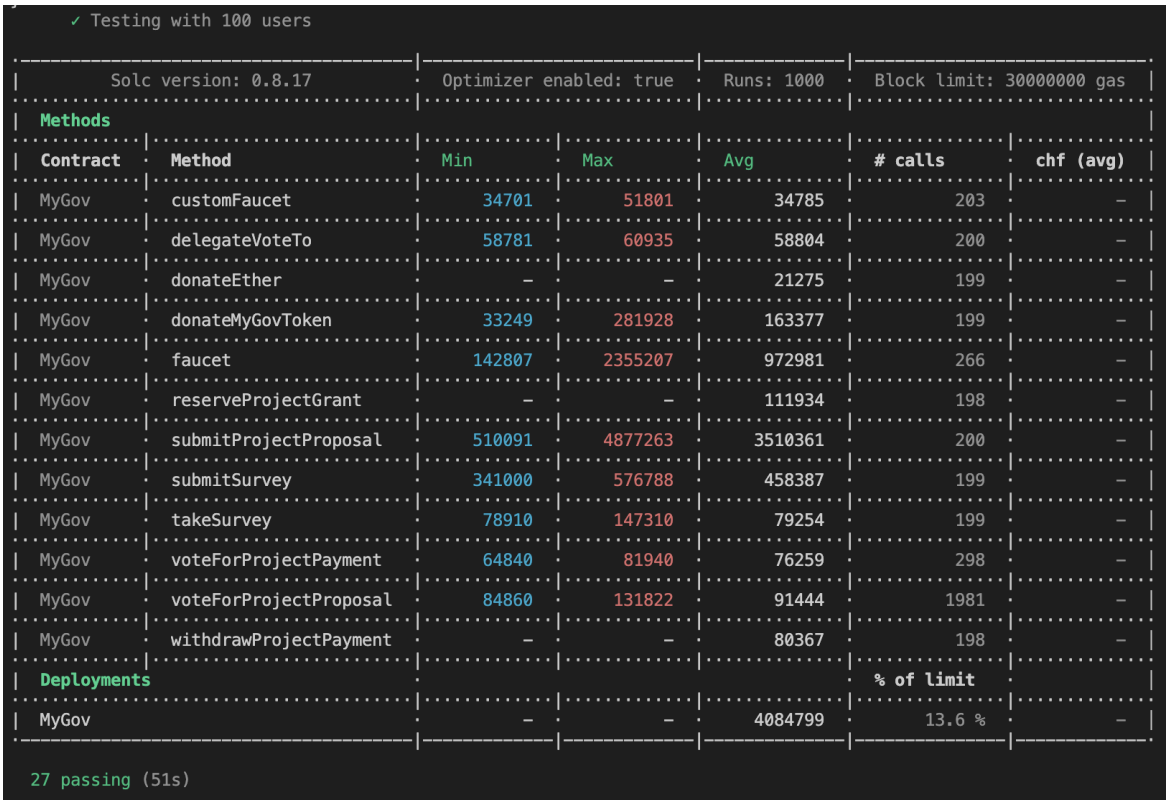
We deployed our contract with **getContractFactory()** and **deploy()** functions and got accounts addresses with **getSigners()** function from Hardhat library. Since all those operations take a little while, we should put **await** in front of them. After deploying our contract, we can call any function in the contract and **await** should be put front to be able to observe the situation after the operation/transaction is done. For assertions, we used **expect()**, **to.equal()**,and **to.be.revertedWith()** functions from Chai library. Since Hardhat returns as BigNumber, we used **utils.parseEther()** function from Hardhat library to convert BigNumber. To be able to test with other accounts than the default one which deploys the contract, we used **connect()** function with account address as parameter.

We tested each function in the contract sequentially, therefore we deployed the contract only once before all tests and then test cases run one after another. We wrote sequential test logic accordingly and used different addresses when needed.

Generating getters that will return big amounts of data in one go would be very expensive and might be even unusable. Therefore, it's avoided. Arrays and mappings of struct are not returned by value from getters on purpose. But we needed to see values in

some mappings of structs in our tests, therefore we implemented our own getter functions like getVoteWeight(), getCurrentPaymentVote(), and getTakenSurveys().

Test Results with 100 Users

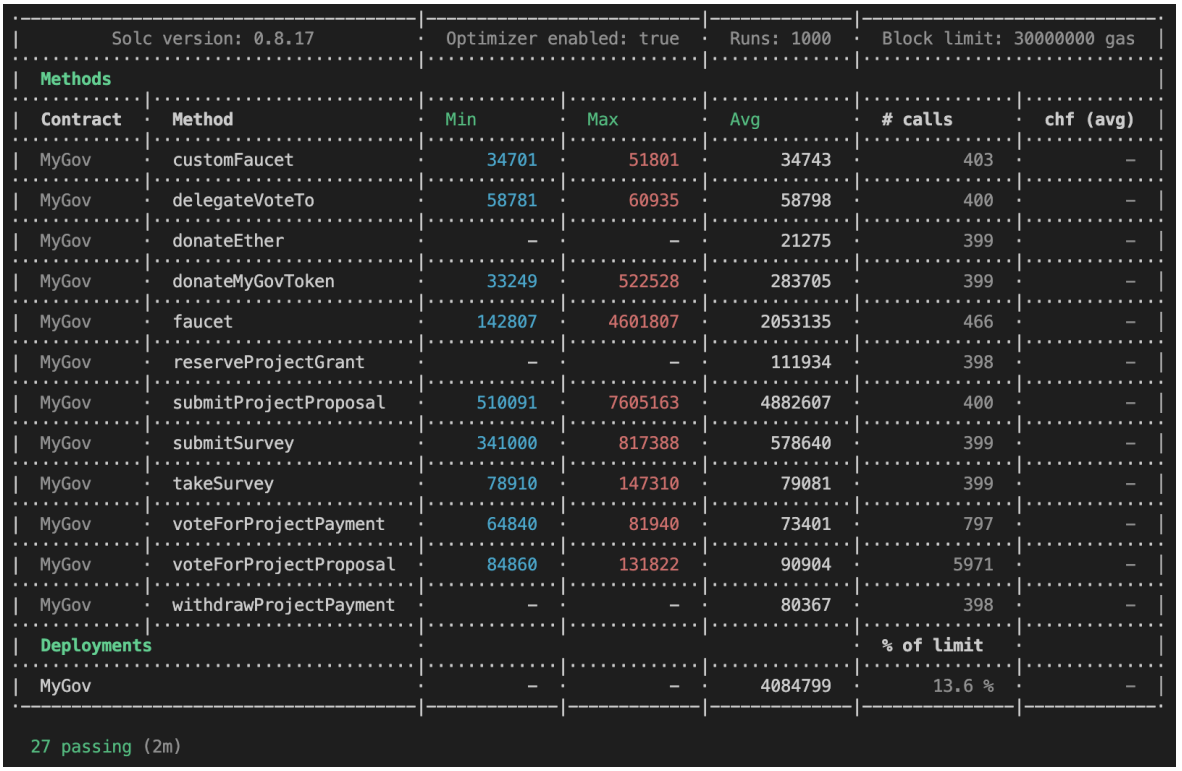


```
      ✓ Testing with 100 users

·---------------------------------------|---------------------------|-------------|-----------------------------·
|         Solc version: 0.8.17          ·  Optimizer enabled: true  ·  Runs: 1000  ·  Block limit: 30000000 gas  |
·.....................................|...........................|.............|.............................·
|  Methods                                                                                                      |
·.....................|.....................|.............|.............|.............|...............|..............·
|  Contract           ·  Method             ·  Min        ·  Max        ·  Avg        ·  # calls      ·  chf (avg)  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  customFaucet       ·      34701  ·      51801  ·      34785  ·          203  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  delegateVoteTo     ·      58781  ·      60935  ·      58804  ·          200  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  donateEther        ·          -  ·          -  ·      21275  ·          199  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  donateMyGovToken   ·      33249  ·     281928  ·     163377  ·          199  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  faucet             ·     142807  ·    2355207  ·     972981  ·          266  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  reserveProjectGrant·          -  ·          -  ·     111934  ·          198  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  submitProjectProposal ·   510091  ·    4877263  ·    3510361  ·          200  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  submitSurvey       ·     341000  ·     576788  ·     458387  ·          199  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  takeSurvey         ·      78910  ·     147310  ·      79254  ·          199  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  voteForProjectPayment ·   64840  ·      81940  ·      76259  ·          298  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  voteForProjectProposal ·  84860  ·     131822  ·      91444  ·         1981  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  withdrawProjectPayment ·      -  ·          -  ·      80367  ·          198  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  Deployments                                ·                                     ·  % of limit    ·              |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·                     ·          -  ·          -  ·     4084799  ·       13.6 %  ·          -  |
·---------------------|---------------------|-------------|-------------|-------------|---------------|--------------·

  27 passing (51s)
```

Test Results with 200 Users



```
·---------------------------------------|---------------------------|-------------|-----------------------------·
|         Solc version: 0.8.17          ·  Optimizer enabled: true  ·  Runs: 1000  ·  Block limit: 30000000 gas  |
·.....................................|...........................|.............|.............................·
|  Methods                                                                                                      |
·.....................|.....................|.............|.............|.............|...............|..............·
|  Contract           ·  Method             ·  Min        ·  Max        ·  Avg        ·  # calls      ·  chf (avg)  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  customFaucet       ·      34701  ·      51801  ·      34743  ·          403  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  delegateVoteTo     ·      58781  ·      60935  ·      58798  ·          400  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  donateEther        ·          -  ·          -  ·      21275  ·          399  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  donateMyGovToken   ·      33249  ·     522528  ·     283705  ·          399  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  faucet             ·     142807  ·    4601807  ·    2053135  ·          466  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  reserveProjectGrant·          -  ·          -  ·     111934  ·          398  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  submitProjectProposal ·   510091  ·    7605163  ·    4882607  ·          400  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  submitSurvey       ·     341000  ·     817388  ·     578640  ·          399  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  takeSurvey         ·      78910  ·     147310  ·      79081  ·          399  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  voteForProjectPayment ·   64840  ·      81940  ·      73401  ·          797  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  voteForProjectProposal ·  84860  ·     131822  ·      90904  ·         5971  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·  withdrawProjectPayment ·      -  ·          -  ·      80367  ·          398  ·          -  |
·.....................|.....................|.............|.............|.............|...............|..............·
|  Deployments                                ·                                     ·  % of limit    ·              |
·.....................|.....................|.............|.............|.............|...............|..............·
|  MyGov              ·                     ·          -  ·          -  ·     4084799  ·       13.6 %  ·          -  |
·---------------------|---------------------|-------------|-------------|-------------|---------------|--------------·

  27 passing (2m)
```

Test Results with 300 Users

Solc version: 0.8.17	Optimizer enabled: true	Runs: 1000	Block limit: 30000000 gas		
Methods					
Contract	**Method**	Min	Max	Avg	# calls	chf (avg)
MyGov	customFaucet	34701	51801	34729	603	–
MyGov	delegateVoteTo	58781	60935	58797	600	–
MyGov	donateEther	–	–	21275	599	–
MyGov	donateMyGovToken	33249	763128	404014	599	–
MyGov	faucet	142807	6848407	3159203	666	–
MyGov	reserveProjectGrant	111934	111946	111936	598	–
MyGov	submitProjectProposal	510091	10333063	6249322	600	–
MyGov	submitSurvey	341000	1057988	698924	599	–
MyGov	takeSurvey	78910	147310	79024	599	–
MyGov	voteForProjectPayment	64840	81952	71689	1496	–
MyGov	voteForProjectProposal	84860	131822	90638	11961	–
MyGov	withdrawProjectPayment	80367	80379	80369	598	–
Deployments				**% of limit**		
MyGov		–	–	4084799	13.6 %	–

27 passing (5m)

Test Results with 400 Users

Solc version: 0.8.17	Optimizer enabled: true	Runs: 1000	Block limit: 30000000 gas		
Methods					
Contract	**Method**	Min	Max	Avg	# calls	chf (avg)
MyGov	customFaucet	34701	51801	34722	803	–
MyGov	delegateVoteTo	58781	60935	58796	800	–
MyGov	donateEther	–	–	21275	799	–
MyGov	donateMyGovToken	33249	1003728	524319	799	–
MyGov	faucet	142807	9095007	4273230	866	–
MyGov	reserveProjectGrant	111934	111946	111938	798	–
MyGov	submitProjectProposal	510091	13060963	7614655	800	–
MyGov	submitSurvey	341000	1298588	819216	799	–
MyGov	takeSurvey	78910	147310	78996	799	–
MyGov	voteForProjectPayment	64840	81952	70549	2395	–
MyGov	voteForProjectProposal	84860	131822	90481	19951	–
MyGov	withdrawProjectPayment	80367	80379	80371	798	–
Deployments				**% of limit**		
MyGov		–	–	4084799	13.6 %	–

27 passing (9m)

Problems Encountered During the Implementation

Using a new programming language like solidity was a bit challenging for us at first. So, the first days of the implementation period was not easy. Most of the time we spent so much time trying to find a very easy functionality or feature of the solidity language and also tried to get used to Remix IDE's user interface: how to deploy, how test accounts work, or how the interface buttons work.

Hardhat Ethereum development environment which we used for testing was also new to us. Creating logical test scenarios and implementing them with an unfamiliar library Chai were not easy tasks but they helped us to create a more reliable MyGov contract. We realized that some functions in our contract were not working as we expected. We debugged and fixed them with the help of tests.

The next big problem for us was the vague description of the project. Since only the surface of the interfaces of the contract was provided in the homework description, the rest of the iceberg was dependent on our design choices. In the first days of the development, it was very challenging to manage trying to understand the project description's scope and trying to learn solidity at the same time. But after realizing that most of the project was up to our design choices, we have smoothly proceeded.

Task Achievement Table

Tasks	Yes	Partially	No
We have prepared documentation with at least 6 pages.	✔		
We have provided average gas usages for the interface functions.	✔		
We have provided comments in my code.	✔		
We have developed test scripts, performed tests and submitted test scripts as well documented test results.	✔		
We 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.	✔		