# Lecture 3

# Solidity tutorial – voting app

**Solidity**

Programming language created for developing smart contracts.

Similar to JavaScript

Statically typed variables.

**.sol Files structure**

* First line -- solidity version
* All statements end with semicolon ( ; )
* Contains a list of contracts

**//solidity version**

**pragma solidity >=0.4.22 <0.7.0;**

**contract A{}**

**contract B{}**

* A contract contains variables, modifiers, functions, and a single constructor. The variables that are not defined in functions are called **state variables**. State variables are stored directly in the Blockcahin storage.

**contract Colector {**

**…**

**address public owner;**

**…**

**modifier onlyOwner {}**

**constructor() public {**

**…**

**}**

**function sendEth() payable public{**

**…**

**}**

**}**

**Function types, visibility**

<https://docs.soliditylang.org/en/v0.6.0/contracts.html>

See examples **Functions.sol**

**external**: can be called only from another contract or using this. May not be specified for state variables

**internal**: functions or state variables that may be accessed only within the contract and in derived contracts.

**private**: functions or state variables that may be accessed only within the contract and not in derived contracts.

**public**: functions or state variables that may be accessed by external user and by the contract. For public state variable a public getter is generated by default.

**view**: a view function is one that doesn't modify state variables but can read them.

**pure**: functions that doesn't read state variables. Pure functions are functions usually includes mathematical functions or formatting functions.

<https://docs.soliditylang.org/en/v0.6.0/types.html>

**Type address**: 20 byte value/42 42-character hexadecimal text

example 0xdb8051413DC52db4afe84EFAF8C278AeE9971AcB

**Type payable address:** type address with members transfer and send. Member function **transfer** reverts on failure. If the execution of **send** fails, the current contract will not stop with an exception, but send will return false.

See examples: **Colector.sol**

**Solidity data locations**

**Storage**: permanent data written on the blockchain, expensive (see below gas costs/optimizations), data visible to all functions, default. State variable are stored in storage and have lifetime limited to the lifetime of the contract. Contract storage acts as a public database, from which values can be read with no fees.

**Memory**: Non-permanent data, accessible within the function, less expensive. For some return parameters, for example for string return parameters it is mandatory to specify memory location.

**Calldata**: non-modifiable, non-persistent data; storage for msg object; default location of parameters (not return parameters) of external functions.

**Events**

Events are special types of functions. Events log data on the blockchain. Events have two types of arguments: topics and values. Topics are indexed key, used to search for the event. An event may have at most *three indexed arguments*.

Events are not used only for logging purposes, but since the gas costs are cheap for emitting an event, they are also a good solution to store tiny amount of data on the blockchain.

**pragma solidity >=0.4.22 <0.7.0;**

**contract Voting{**

**struct Proposal {**

**string name; // short name (up to 32 bytes)**

**uint8 voteCount; // number of accumulated votes**

**}**

**mapping(address => bool) public voted;**

**Proposal[] public proposals;**

**address public chairperson;**

**event LogVote(address indexed voterAddress, string name, uint8 votes);**

**constructor() public {**

**chairperson = msg.sender;**

**}**

**function addProposal (string memory proposalName) public {**

**proposals.push(Proposal({**

**name: proposalName,**

**voteCount: 0**

**})**

**);**

**}**

**function vote(uint proposal) public {**

**address voter = msg.sender;**

**voted[voter] = true;**

**proposals[proposal].voteCount++;**

**emit LogVote(voter, proposals[proposal].name, proposals[proposal].voteCount);**

**}**

**}**

Events in web3:

* web3.eth.subscribe
* web3.eth.abi.decodeLog

See examples: **Voting.sol** and **DemoWeb3Events**

<https://docs.soliditylang.org/en/v0.6.0/contracts.html#events>

**Modifiers**

Modifiers are special types of functions that verify certain conditions before executing other functions. Usually, they are used to verify permissions or other parameters.

* “\_” indicates where the code of the function is to be executed.
* require indicates the condition to be tested.

**address public chairperson;**

**modifier onlyChairperson() {**

**require(msg.sender == chairperson, 'You must be the chairperson');**

**\_;**

**}**

**function addProposal (string memory proposalName) public onlyChairperson {**

**proposals.push(Proposal({**

**name: proposalName,**

**voteCount: 0**

**})**

**);**

**}**

**modifier voteOnce() {**

**require(voted[msg.sender] == false, 'You must vote only once!');**

**\_;**

**}**

**function vote(uint proposal) public voteOnce {**

**address voter = msg.sender;**

**voted[voter] = true;**

**proposals[proposal].voteCount++;**

**emit LogVote(voter, proposals[proposal].name, proposals[proposal].voteCount);**

**}**

All symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function.

<https://docs.soliditylang.org/en/v0.6.0/contracts.html#function-modifiers>

**Ethereum networks**

**Ropsten,** test network similar to mainnet.

Each testnet operates on separate blockchains with different set of rules and limitations. Ethereum largest upgrades, Constantinople, was first released on Ropsten before it was published on mainnet.

* PoW consensus algorithm, vulnerabilities: spam attacks, 2017 attack;
* Each Ethereum network has un id (unique identifier). The id for test network Ropsten is **3**;
* Ropsten Ether holds no corresponding real value. Obtain Ropsten Ethers from

<https://faucet.ropsten.be/>

<https://faucet.metamask.io/>

* Block rate 30sec.

**Rinkeby,** test network created to experiment a different type of mining.

* PoA consensus algorithm;
* The id for test network Rinkeby is **4**;
* Block rate 15sec;
* Obtain Rinkeby Ethers from

<https://faucet.rinkeby.io>

<https://www.rinkeby.io/#faucet>

**Kovan**, created by Parity

* PoA consensus algorithm with approved validators;
* The id for test network Kovan is **4**;
* Block rate 4sec.
* provide **Kovan Improvement Proposals KIP**
* Obtain Rinkeby Ethers from

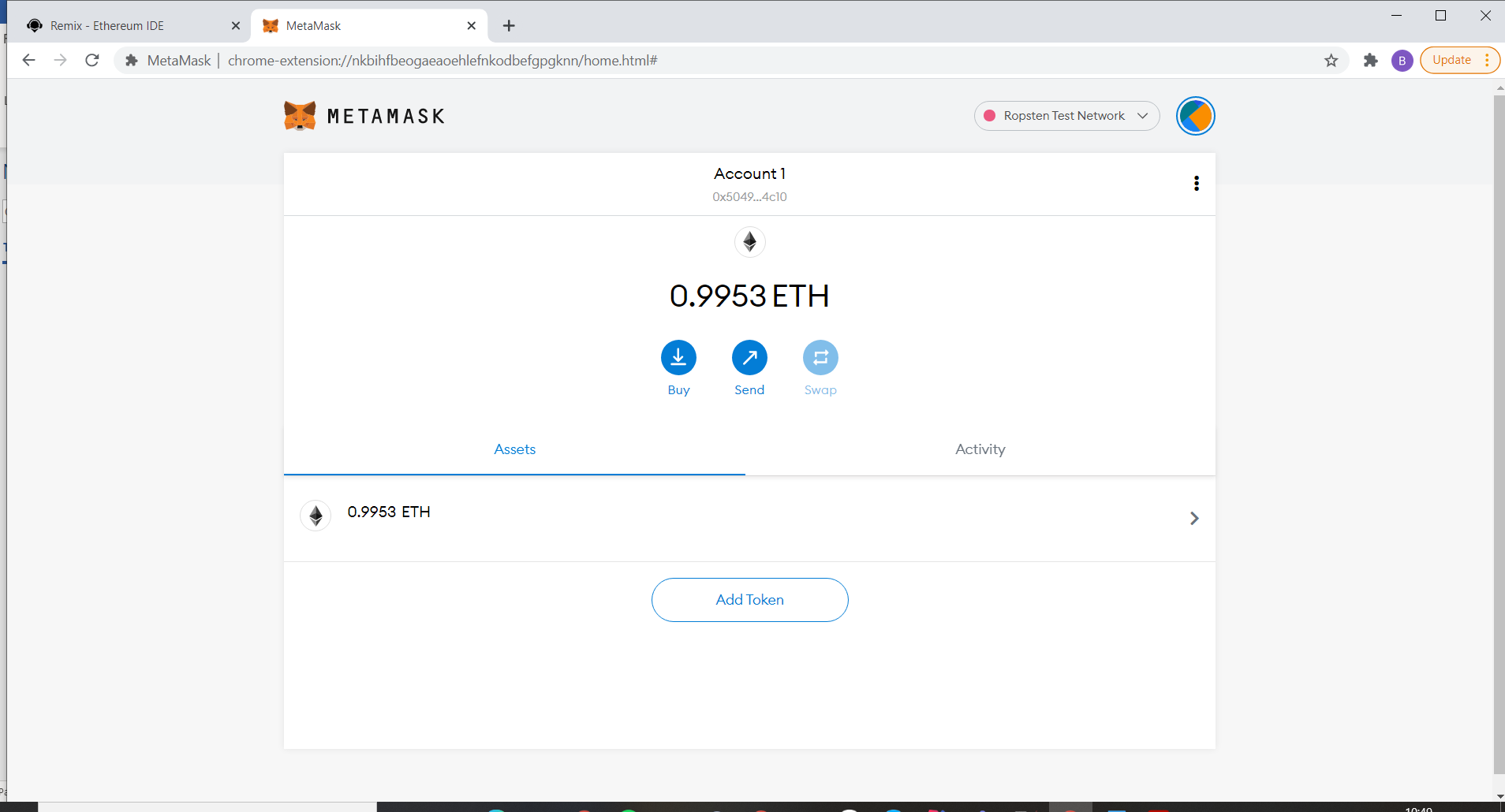
<https://gitter.im/kovan-testnet/faucet>

**Deployment on test networks**

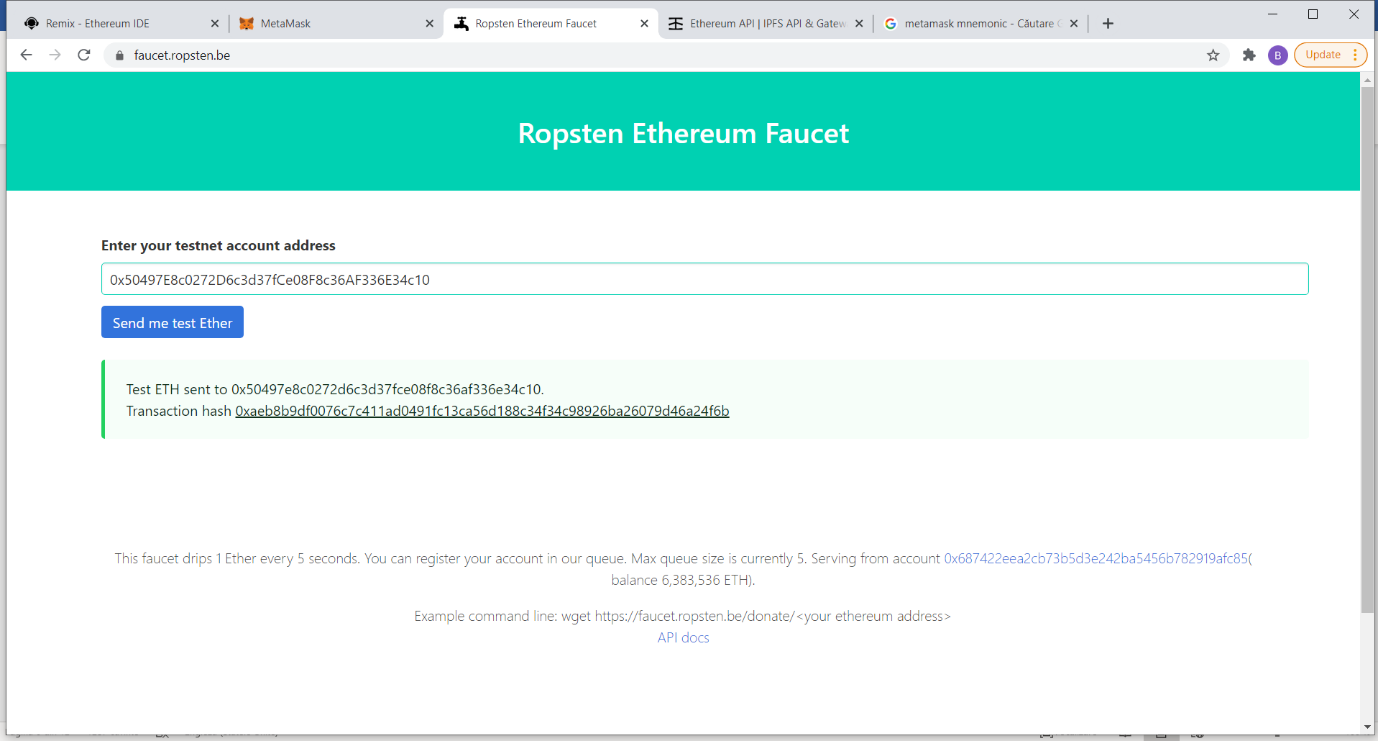
1. Get a Metamask account from:

<https://metamask.io/>

* Metamask is a popular Ethereum wallet. It stores account for different Ethereum chains.
* It may be downloaded as an extension from Chrome or Firefox.
* It uses **INFURA**. Truffle also uses INFURA. INFURA allows immediate connections to the blockchain without downloading a full node.
* It automatically injects web3.js into dApp pages.



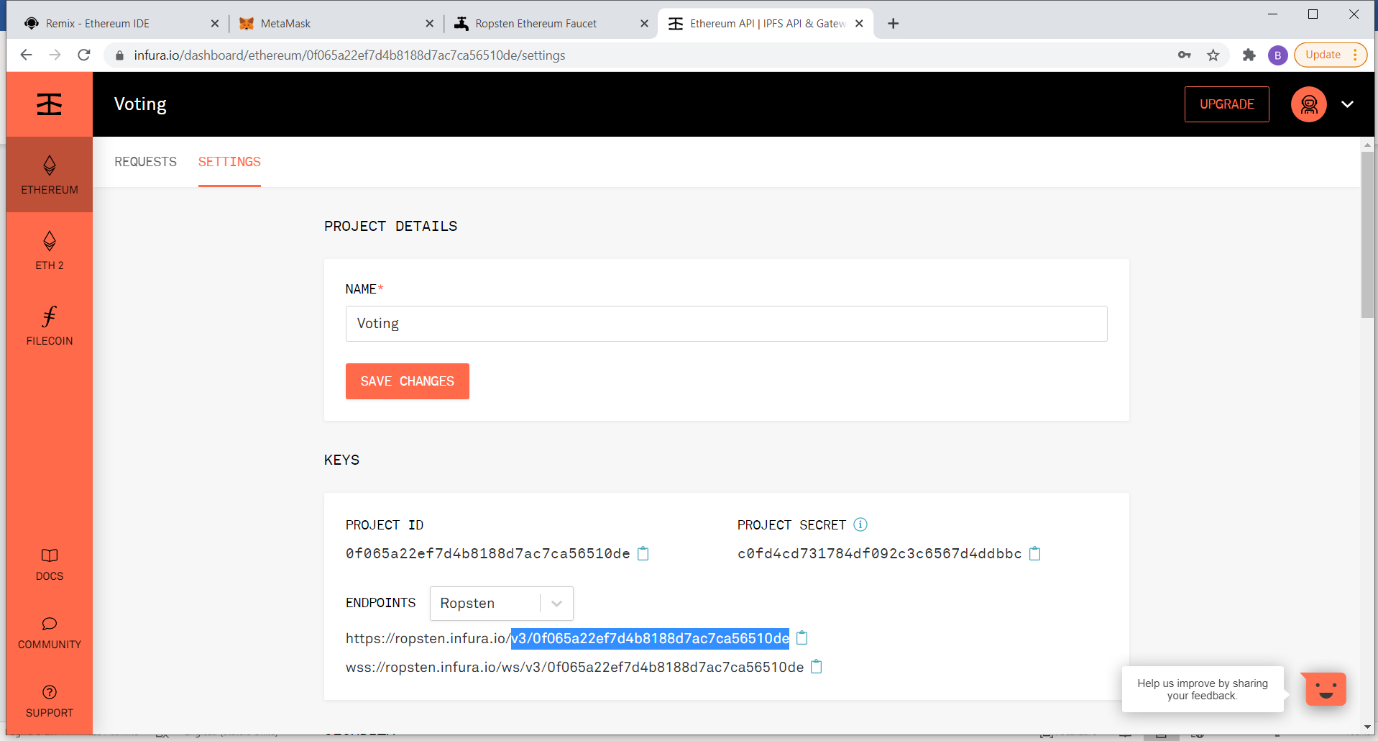
1. Get ropsten Ether from <https://faucet.ropsten.be/> Use your metamask account.



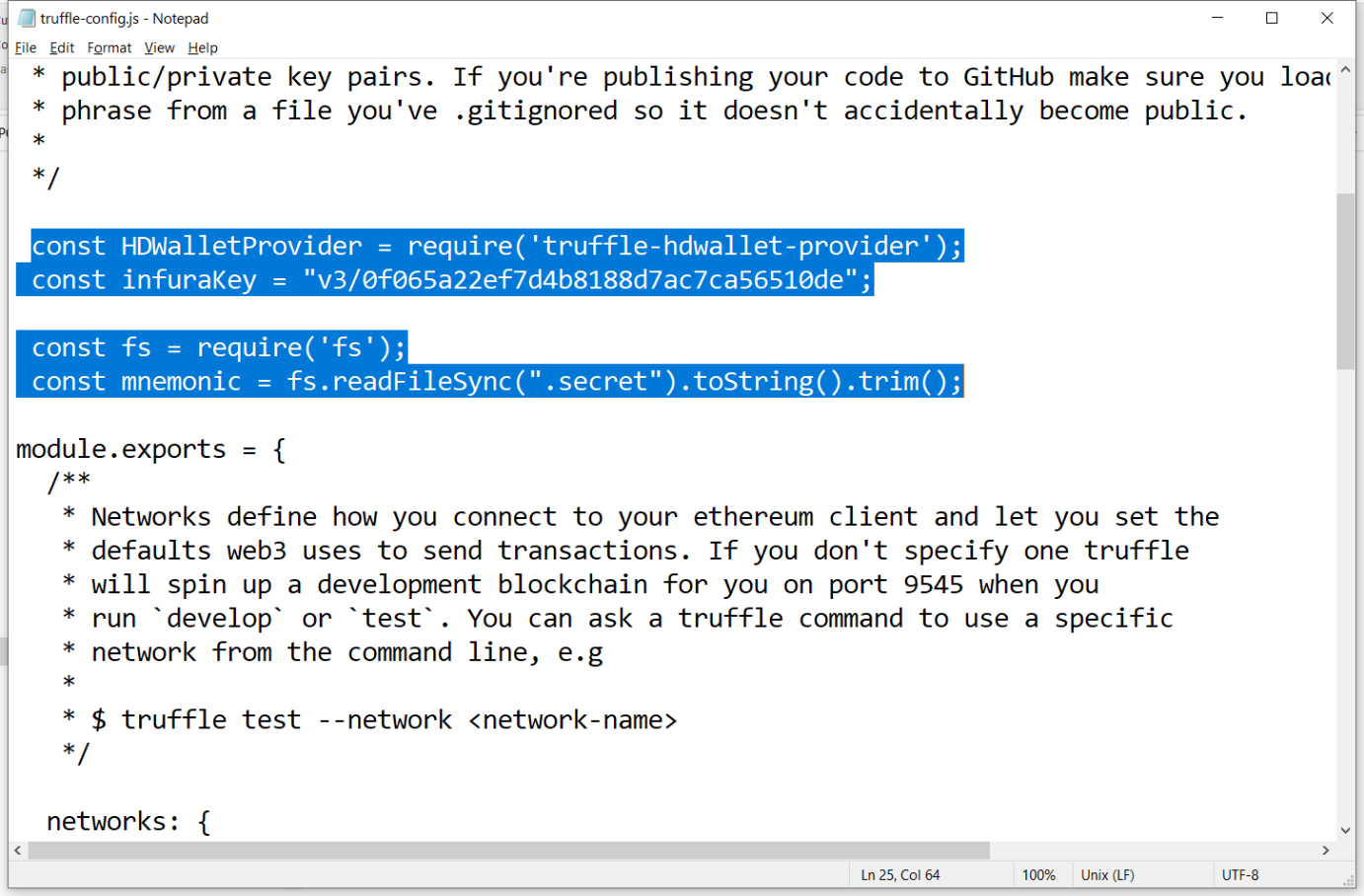
1. Init a new truffle project and add Voting.sol to contracts. Run on an empty folder:

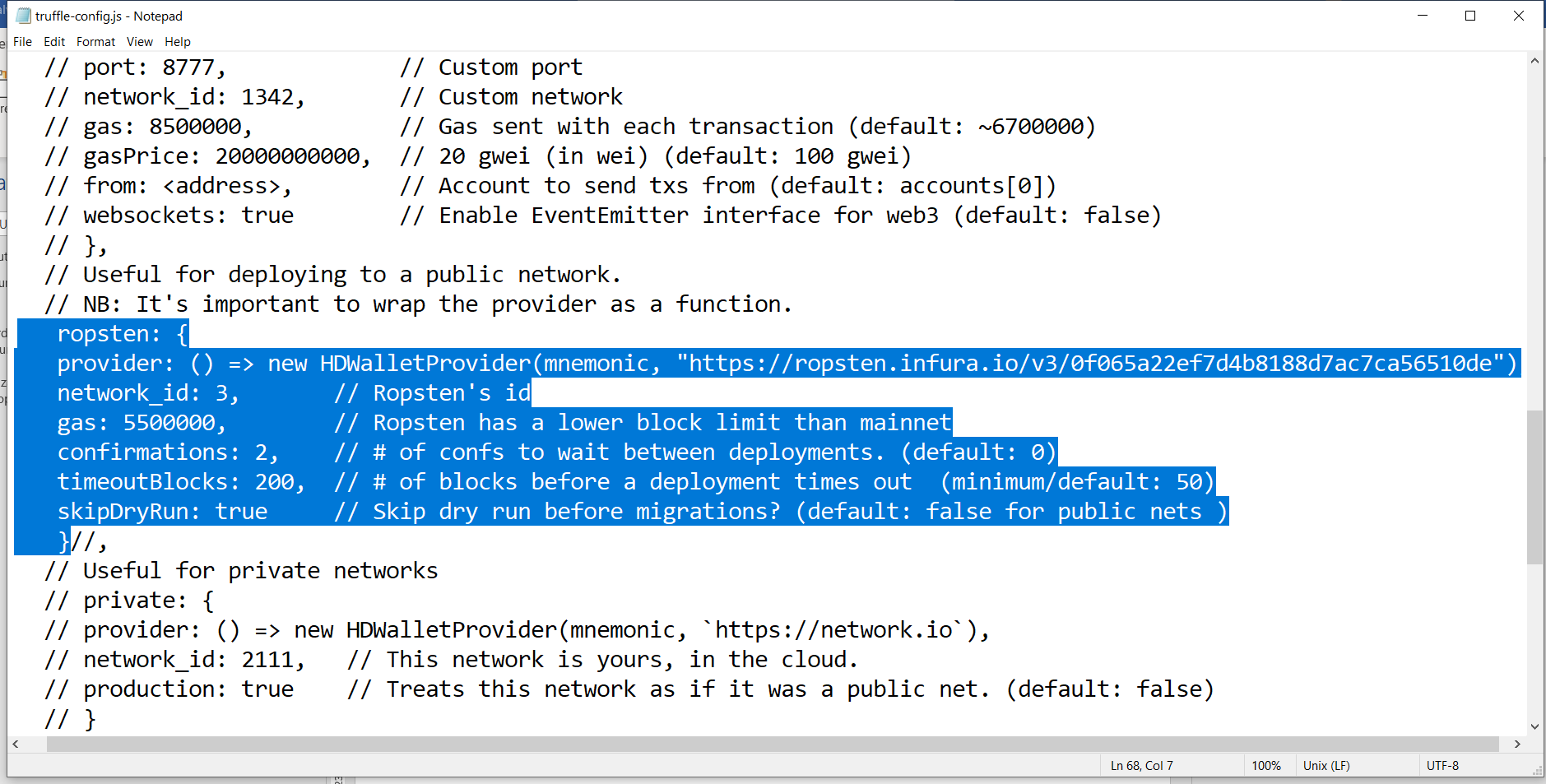
**>> truffle init**

1. Get an INFURA key from <https://infura.io/>. Create a new project and get the infura key:



1. In truffle folder add .secret file with the MetaMask mnemonic.
2. Modify truffle-config.js, uncomment infuraKey and Ropsten network:





1. Modify migrations/1\_initial\_migrations.js

**const Voting = artifacts.require("./Voting.sol");**

**module.exports = function (deployer) {**

**deployer.deploy(Voting);**

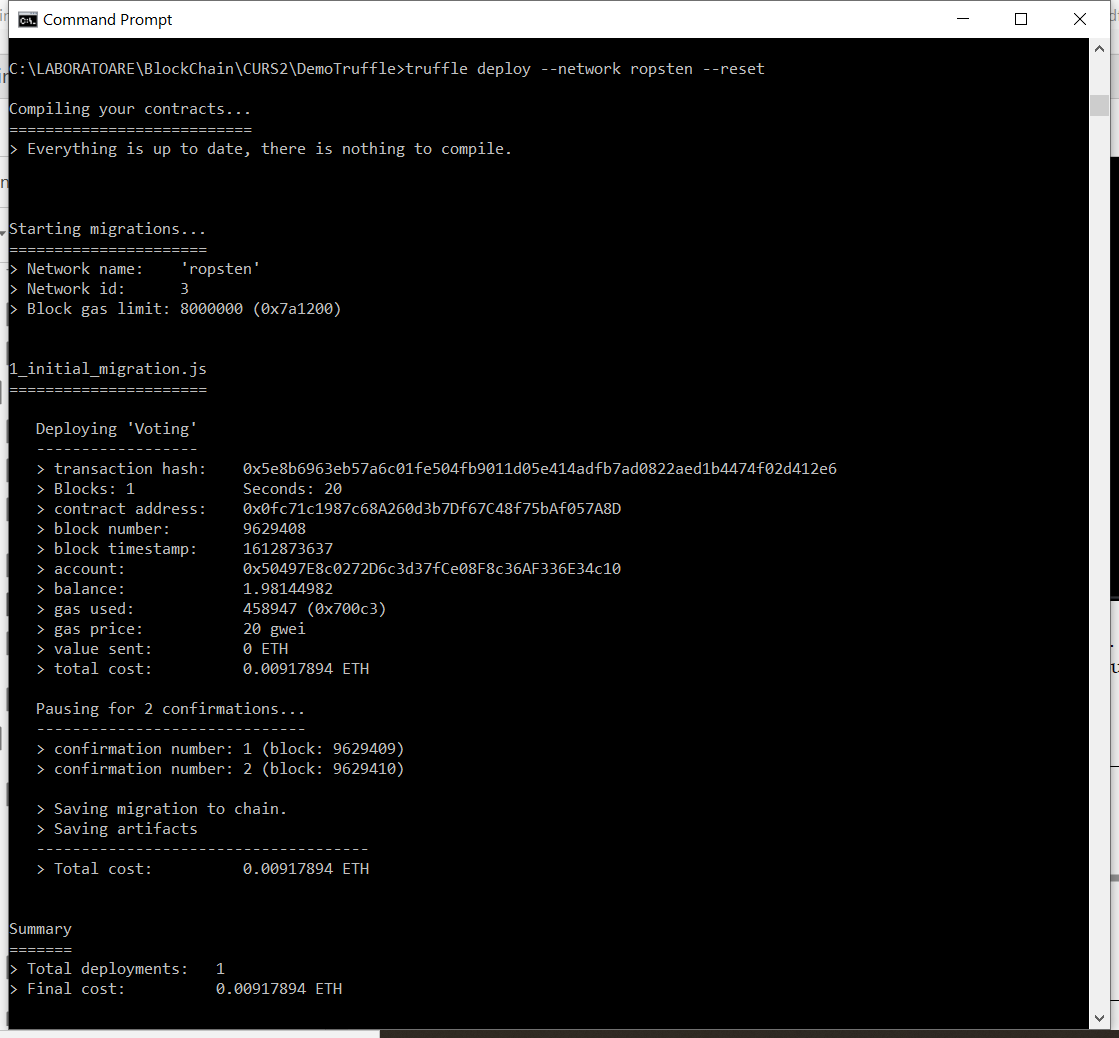
**};**

1. Install truffle wallet provider and deploy on ropsten network.

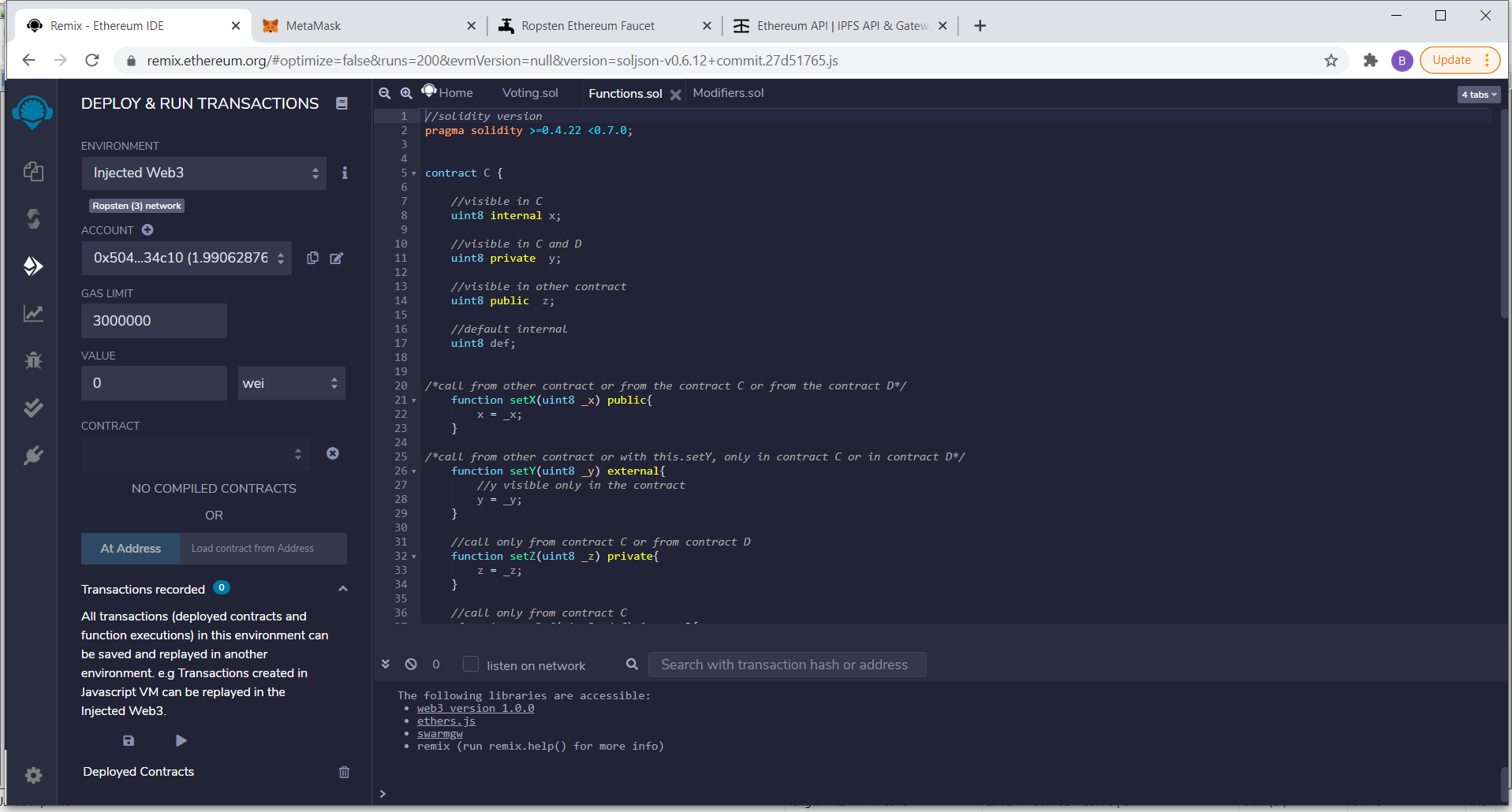
**>> npm install truffle-hdwallet-provider**

**>> truffle deploy --network ropsten**

1. Check contract address:



1. Alternatively you can deploy on ropsten with remix ide by selecting environment: Injected Web3.



**Gas optimizations:**

See gas costs in: <https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWcOlRem_mO09GtSKEKrAsfO7Frgx18pNU/edit#gid=0> and in Yellow Paper (Appendix H)

<https://ethereum.github.io/yellowpaper/paper.pdf>

Most expensive operation: SSTORE store information into the blockchain.

**Optimization techniques**:

1. order of logical operators: put first the operator most likely to be true in OR/ false in AND
2. Use memory variable (in for/while; instead of storage variables)
3. Avoid unreachable code.
4. Use smaller size for variables. (uint8 instead of uint256); use byte32 instead of string
5. Limit BALANCE calls.
6. Use **view** and **pure** functions for free.