**Objective**

**What is blockchain**

**What is a smart contract?**

**What is Binance Smart Chain?**

**What we will be building**

**Short introduction to solidity**

**Setting up the development**

**Start by looking at the structure of the codes.**

**Security implementations and best practices.**

**Deployed and testing (detailed in the video)**

**Objective**

**The objective of this tutorial is to teach you how to develop and deploy a voting application on the Binance Smart Chain which will reveal the winner at the end of the election, we will be focusing on functions implementation, data types and security.**

##### **What is a Blockchain?**

**A blockchain is a peer-to-peer network of computers, or nodes, that talk to one another. It's a distributed network where all of the participants share the responsibility of running the network. Each network participant maintains a copy of the code and the data on the blockchain. All of this data is contained in bundles of records called "blocks" which are "chained together" to make up the blockchain. All of the nodes on the network ensure that this data is secure and unchangeable, unlike a centralized application where the code and data can be changed at any time.**

##### **What is a Smart Contract?**

**A smart contract is a complete program that is mutually executed by the blockchain network.**

**All of the code on the blockchain is contained in smart contracts, which are programs that run on the blockchain. They are the building blocks of blockchain applications.**

**Smart contracts are written in a programming language called Solidity, which looks a lot like JavaScript. All of the code in the smart contract is immutable, or unchangeable. Once we deploy the smart contract to the blockchain, we won't be able to change or update any of the code. This is a design feature that ensures that the code is trustless and secure. Smart contracts are like microservices on the web. They act as an interface for reading and writing data from the blockchain, as well as executing business logic. They're publicly accessible, meaning anyone with access to the blockchain can access their interface.**

**What is BSC?**

**Binance Smart Chain is an EVM compatible blockchain which runs in tandem with the Binance Chain (which does not support smart contracts, but rather tokens). See a complete description here. BSC plays nicely with many of the familiar tools that ethereum developers use to enhance security, scalability, and user experience, in their projects. In this tutorial we will be using solidity which is used by ethereum smart contract developers to develop for the Binance Smart Chain.**

**What is solidity?**

**Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of Smart contracts within the Binance Smart Chain (BSC) network state.**

**Setting up a development environment.**

**For simplicity we will be making use of the** [**remix ide**](https://remix.ethereum.org/)**, Details can be found here “**[**Welcome to Remix documentation!**](https://remix.readthedocs.io/en/stable/index.html)**”,**

**Step 1: Installing and Configuring Metamask for the Binance Smart Chain**

**You’ll first need a BSC address and private key. Note that this address only exists on the BSC, even though it is identical to an ethereum address (in fact, they are generated the same way). We’ll explore more flexible ways to generate keypairs in later tutorials, but for now, we will use Metamask. Go to** [**www.metamask.io**](http://www.metamask.io/) **and install the metamask chrome extension.**

**By default, Metamask is configured for the ethereum Blockchain, but it does support what is called a “custom RPC”. So once you create a normal ethereum account, go up to networks>custom RPC and enter the following information:**

* **Name: Binance Smart Chain Testnet**
* **RPC URL:** [**https://data-seed-prebsc-1-s1.binance.org:8545**](https://data-seed-prebsc-1-s1.binance.org:8545/)
* **ChainID: 97**
* **Symbol: BNB**
* **Block Explorer:** [**https://explorer.binance.org/smart-testnet**](https://explorer.binance.org/smart-testnet)

**Here is a valid Metamask configuration for a BSC mainnet RPC in case you want to deploy on the Binance Smart Chain Main net:**

* **RPC URL: https://bsc-dataseed4.ninicoin.io:443**
* **ChainID: 56**
* **Symbol: BNB**
* **Block Explorer:** [**https://explorer.binance.org/smart**](https://explorer.binance.org/smart)

**Now, click your address to copy it.**

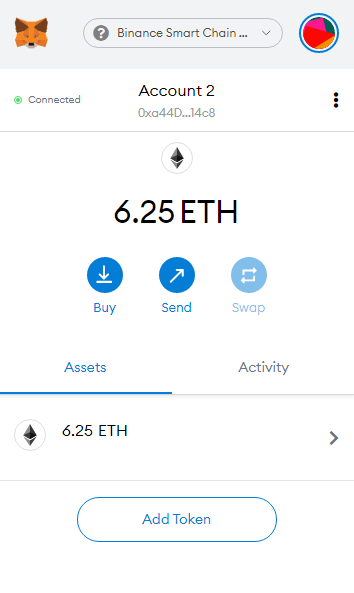
**Step 2: Testnet faucet**

**On the ethereum blockchain, transactions (including logic) are paid for using the network’s native currency binance coin (BNB) which is converted into “gas”. The native currency for the Binance smart chain is Binance coin (BNB). In general, the more complex the operation, the more gas required.**

**For a production dapp, you’ll need to buy some BNB , but for these exercises, we will be using the testnet for testing the contract using the bnb “faucet”.**

**Go to** [**https://testnet.binance.org/faucet-smart**](https://testnet.binance.org/faucet-smart) **and paste your metamask BSC address. Within seconds, you should see the BNB in your metamask wallet. Do not proceed if you do not have any BNB, as the following steps will fail.**

**For this tutorial we will need at most 10 accounts on the binance smart chain each loaded with some faucet BNB. So metamask and create more accounts then repeat step 2 to gain more faucet bnb, so that we can have enough accounts to test the voting smart contract.**

**After you are successful with the first faucet you should have something like the image below.**

**Step 3: Remix/solidity**

**For simplicity we will be using remix ide So Go to** [**http://remix.ethereum.org/**](http://remix.ethereum.org/) **Go to settings and turn on dark mode. Also, go to the “analysis” tab and make sure all boxes are checked, though you’re mainly going to use the “Compile” and “Run” tabs. Go to the “Compile” tab and make sure “Enable Optimization” is checked. Metamask will also ask if it can connect. Confirm it.**

**Now we’re ready to code! For this tutorial, we’ll be making a simple “Voting smart contract” contract that lets us conduct elections on the Binance Smart Chain.**

**Make sure the correct compiler version is selected in the dropdown (it should match the pragma specification).**

**Make sure the contract name in the dropdown matches the contract we specified in the code. Click “start to compile”. If there are no bugs, great!**

**Go to the “run” tab. Make sure Environment is set to Injected Web3 (you should be signed into Metamask with the custom RPC BSC selected). Make sure the account is selected to have 6.25 bnb (it says bnb, but it is actually BNB)**

**Select the correct contract “PayToPlay” and enter your address next to “deploy”. Click Deploy and confirm the metamask popup, and wait until the transaction is mined.**

**You will see the contract and its address appear with buttons and fields for you to use to interact with it. Go ahead and write a message and confirm the transaction.**

**Send some BNB to another address in your metamask account and interact with the contract. Question: which function will fail if you try it with this alternate address?**

**4. Block explorer**

**In your metamask address transaction history, you can see the transaction hash. Copy it, and go to** [**https://explorer.binance.org/smart-testnet**](https://explorer.binance.org/smart-testnet) **and paste it in the search field.**

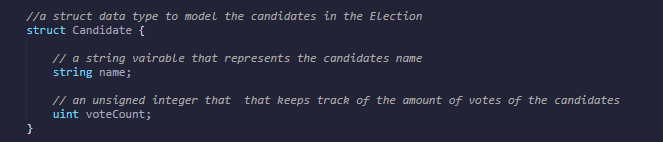
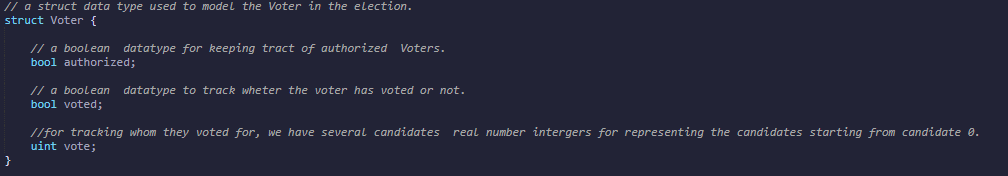
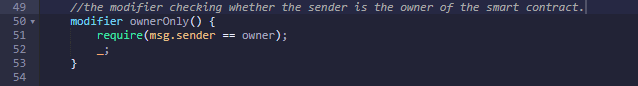
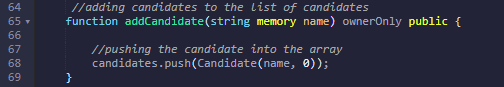
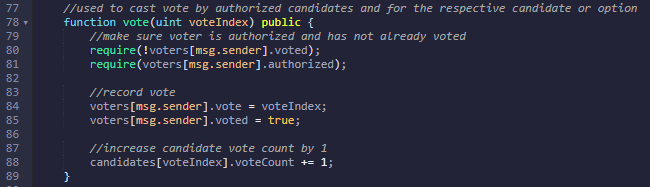
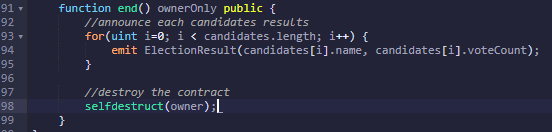
**Write the voting contract**

**Now that we have our dependencies and our development environment setup, let’s start building our Voting Smart contract.**

**A smart contract in the sense of Solidity is a collection of code (its *functions*) and data (its *state*) that resides at a specific address on the ethereum blockchain.**

**Step by step what we will do and how we will do them**

# **The structure of our voting smart contract**

* **The Line below is a machine readable source code licence specifying that its under GPL-3.0 **
* **The line below is specifying the solidity version **
* **Candidate struct: this is the data structure that is used to encapsulate the election candidates. Its shown below **
* **Voter struct: this is used to encapsulate the voters in the election, inside the voter struct we have three state variable the first one is of boolean type used to keep track of the voters authorisation status, the second one is also a boolean to track whether the user has voted or not and the third one is also to keep track of the candidates. Its shown below **
* **Address of the Owner: this is also a state variable of type address “payable” inside the election smart contract used to keep track of the wallet address of the person that deployed the smart contract into the BSC testnet/mainnet. Its shown below**
* **The name of the election: this is a state variable used to define what the election is about by naming the election we are able to directly label(US presidential election, adding a feature to a software system, who is the best UFC fighter? etc.) the purpose of the election i.e we can conduct election about different events like political election, Voting on key decisions in an organisation etc. Its shown below **
* **Mapping of the voter to the public address: this is a data type like a dictionary in python that holds key value pairs, but in this situation we are using it to keep track of the voters and their respective addresses. Its shown below **
* **List of candidates: this is a publicly accessible array used to keep track of authorized candidates, candidates are the people or things are voted for during the election. Its shown in the figure below**
* **Event of the election results: see the last message that was written. Its shown in the figure below**
* **Modifier: it is used to check if the account is the owner i.e if the account is the account used to deploy the contract on the Binance smart chain.Its shown in the figure below**
* **AddCandidate function: this is the function used to add individual candidates to the list of candidates or options that can be elected during the election. Its shown below **
* **Authorize function: this is a function used to authorize voter wallet address or accounts so that only authorized voters can vote during the election. Its shown below **
* **Vote Function: this function is used cast the vote by the authorized voters for the candidates or options of their choice. Its shown below**
* **End function: This is the function responsible for ending the election by calling the self destruct method that is only accessible to the owner of the contract. Its shown below**

# **Solidity Best Practices for Smart Contract Security**

**After developing the smart contract above we are going to improve the security of the smart contract by following some best practise and solving some bugs in our code.**

**Now create another file in remix ide and call it secured voting smart contract, we are going to follow some best practise and also use the inbuilt solidity analysis tools to analyse our smart contract.**

**The following are the list of solidity best practices which can be found** [**here**](https://docs.soliditylang.org/en/latest/security-considerations.html)**.**

1. **Using modifiers only for checks.**

**The code inside a modifier is usually executed before the function body, so any state changes or external calls will violate the** [**Checks-Effects-Interactions**](https://solidity.readthedocs.io/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern) **pattern. Moreover, these statements may also remain unnoticed by the developer, as the code for modifier may be far from the function declaration. For example, an external call in modifier can lead to the reentrancy attack.**

#### 

1. **Lock pragmas to specific versions.**

**Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors. It can be seen below as we lock the version to solidity compiler version**

**Unlocked version**

****

**Locked version**

****