

1- How to Install Truffle

For Windows 11

(WARNING: Truffle version 4 and below has some naming errors that can cause problems in running Truffle commands with Windows 11)

- Install NodeJS v8.9.4 or later
 - Check NodeJS version by using the command
 - `$ node -v`
 - If not at least v8.9.4
 - Install node.js from [Download | Node.js \(nodejs.org\)](https://nodejs.org/)
 - Make sure to run
 - `$ npm init`
 - This initializes the project
- Download Truffle by using the following command in your terminal
 - `$ npm install -g truffle`

For Mac-OS

- 1) Install Node.js and NPM
 - a) Open your terminal, and check if you have Node.js installed already by typing “node -v”. If it prints out a version number, you are in the clear. Otherwise, keep following the steps for Node.js below.
 - b) If you have homebrew, just type “brew install node” on your terminal, and you’ll be all set. Otherwise, type these commands in order to your terminal to install homebrew (and use the “brew install node” command after):
 - i) `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`
 - ii) `echo 'eval $(/opt/homebrew/bin/brew shellenv)' >> /Users/$USER/.zprofile`
 - iii) `eval $(/opt/homebrew/bin/brew shellenv)`
- 2) Install Truffle through NPM
 - a) Go to your terminal and type “sudo npm install -g truffle”. You are all set!

2- Connecting to a Local Ganache Blockchain using Truffle

Ganache is a tool to create and visualize a local blockchain. In this part, we will use ganache to create a local blockchain and connect to it through truffle. Exciting!

1) Create a Local Blockchain with Ganache

- a) Navigate to <https://trufflesuite.com/ganache/> and install Ganache for your OS.
- b) Open Ganache, and create a new workspace by pressing “new workspace” and then clicking the “save workspace” button on the right corner. Congrats, you are the owner of a blockchain now!

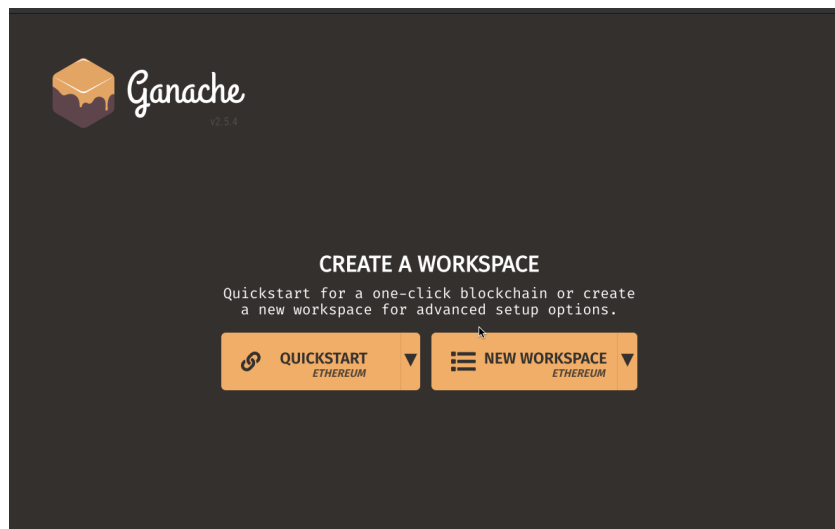


Figure 1

2) Create a Truffle project

- a) Create a new folder, and navigate to it through your terminal of choice.
- b) Type “truffle init” to your terminal inside the folder. You should have a few folders, and a truffle-config.js file in your folder now.

```
olsenbudanur@olsens-air truffletest % mkdir truffletest
olsenbudanur@olsens-air truffletest % cd truffletest
olsenbudanur@olsens-air truffletest % truffle init

Starting init...
=====
> Copying project files to /Users/olsenbudanur/.Trash/truffletest 7.50.30 PM/truffletest

Init successful, sweet!

Try our scaffold commands to get started:
$ truffle create contract YourContractName # scaffold a contract
$ truffle create test YourTestName        # scaffold a test

http://trufflesuite.com/docs

olsenbudanur@olsens-air truffletest % ls
contracts      migrations    test          truffle-config.js
olsenbudanur@olsens-air truffletest %
```

Figure 2

3) Connect to the Ganache Blockchain Through Truffle

- Now that we have a Truffle project, and a Ganache blockchain, it's time to connect to the blockchain through Truffle! Make sure your Ganache blockchain window is open prior to following these steps.
- Navigate to the folder you initiated your Truffle project in. Then, open the file labeled truffle-config.js with a text editor.
- You should see commented out lines. Navigate to the line that says "development" and uncomment the following lines: development, port, networkid, and the bracket.
- Make sure the host, and the port in the file matches the host and the port of the Ganache blockchain. Reference Figure 3 for help.

```
networks: {  
  // Useful for testing. The 'development' name is special - truffle uses it by default  
  // if it's defined here and no other network is specified at the command line.  
  // You should run a client (like ganache-cli, geth or parity) in a separate terminal  
  // tab if you use this network and you must also set the 'host', 'port' and 'network_id'  
  // options below to some value.  
  //  
  // development: {  
  //   host: "127.0.0.1",      // Localhost (default: none)  
  //   port: 8545,           // Standard Ethereum port (default: none)  
  //   network_id: "*",      // Any network (default: none)  
  // },  
  // Another network with more advanced options...
```

Figure 3

- Navigate to the folder with your truffle project through your terminal.
- Type "truffle console" to your terminal. Congrats, you are connected to your local blockchain!
- Let's test if you are connected to the local blockchain. Type "accounts" to your terminal. You should see a list of hexadecimals of the account addresses. Compare it to the addresses in your Ganache window, and if they match, you are all good!

```
olsenbudanur@olsens-air truffleTest % ls  
contracts          test  
migrations         truffle-config.js  
olsenbudanur@olsens-air truffleTest % truffle console  
truffle(development)> accounts  
[  
  '0xb0366b922999dDd839DE123D9C28B31FEab6Efb8',  
  '0xC7AF049726c2Ee09eae159e1931ff8E41644EfCa',  
  '0x95AbEC0771883FE4B083Edf10f548Ad42B5d3A0A',  
  '0x70859E50D03BA8f23e6Ab607cd8a9f19f9ab8D91',  
  '0x0122cea22fd5C18a97c565650E375Fd04524b0a1',  
  '0x8cAd0CAB2B97dC4602F4d2e8C113F9f4ef5686a6',  
  '0x4B9637275C41F4D7460EAA4d294d4B039bE36104',  
  '0x3ac88e97d0858E7a2135de0cc67339f182429C89',  
  '0x264ceA09282aCA602E1CD1a7051C569dC84dbbEe',  
  '0x9cc34e8A2bE6ec21B187A40ce312b7179b2662dC'  
]  
truffle(development)>
```

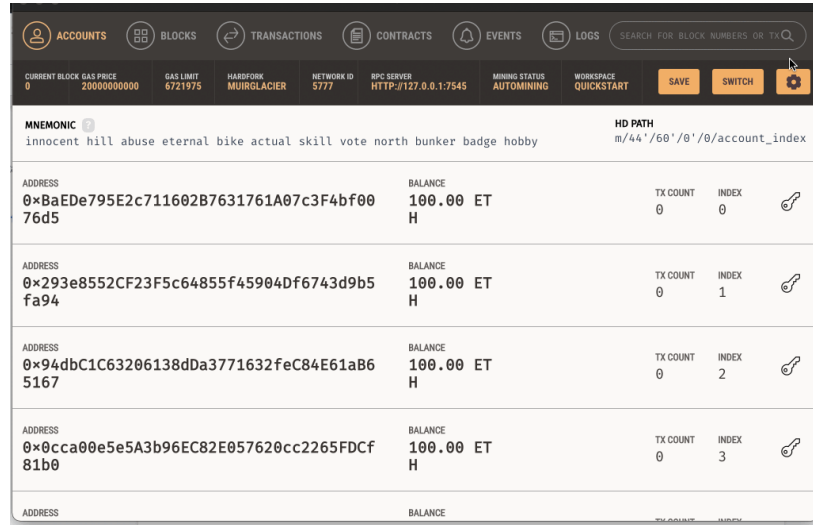
MNEMONIC

grape love tent jump trophy occur history tiger axis tribe benefit scissors

ADDRESS	BALANCE
0xb0366b922999dDd839DE123D9C28B31FEab6Efb8	100.00 ETH
0xC7AF049726c2Ee09eae159e1931ff8E41644EfCa	100.00 ETH

Figure 4

- 4) Link your Truffle project to Ganache.
- a) Go to your Ganache workspace, click the gear button (settings) on the right side. Then link your truffle project to your Ganache blockchain by clicking “add project” and selecting your truffle config file (shown in Figure 5). You are done for this part!



ACCOUNTS			
BLOCKS			
TRANSACTIONS			
CONTRACTS			
EVENTS			
LOGS			
SEARCH FOR BLOCK NUMBERS OR TX Q			
CURRENT BLOCK GAS PRICE			
0 20000000000			
GAS LIMIT			
6721975			
HARDFORK			
MUIRGLACIER			
NETWORK ID			
5777			
RPC SERVER			
HTTP://127.0.0.1:7545			
MINING STATUS			
AUTOMINING			
WORKSPACE			
QUICKSTART			
SAVE SWITCH			
Mnemonic			
innocent hill abuse eternal bike actual skill vote north bunker badge hobby			
HD PATH			
m/44'/60'/0'/0/account_index			
ADDRESS	BALANCE	TX COUNT	INDEX
0xBaEde795E2c711602B7631761A07c3F4bf0076d5	100.00 ETH	0	0
0x293e8552CF23F5c64855f45904Df6743d9b5fa94	100.00 ETH	0	1
0x94dbC1C63206138dDa3771632feC84E61aB65167	100.00 ETH	0	2
0x0cca00e5e5A3b96EC82E057620cc2265FDCf81b0	100.00 ETH	0	3
ADDRESS	BALANCE	TX COUNT	INDEX

Figure 5

3- Creating Our First Smart Contract

Now that we have a local blockchain on Ganache, and are able to interact with it using Truffle, let's create our first smart contract! Our smart contract is going to be a faucet people can deposit & withdraw Ether from. Before we get to coding anything, let's understand what our smart contract is going to do.

Once we deploy our smart contract, it will have an address of its own (just like an account!). The faucet will work this way: Once it is deployed, other accounts have the ability to send Ether to it. The faucet will store the Ether that is sent to it. Other accounts will also have the ability to trigger one of our faucet's functions, which will be to dispense Ether to whoever asks for it. Think of the faucet as a bucket people can put Ether into, or withdraw Ether from. Enough talking, let's get to it!

1) Create the Faucet Contract

- a) Navigate to the contracts folder in your Truffle project. Create a file named "Faucet.sol", and open it using the IDE of your choice. Note: If you are using VSCode, you should download the Solidity extension for syntax highlighting.

- b) Your first line should specify the version of Solidity you are using. Here is the version specifier we used at the time of creating this documentation.

- i) `pragma solidity >=0.4.22 <0.9.0;`

- c) Then, we declare our contract with its name.

- i) `contract Faucet {`

- d) Then, we enable the contract to accept incoming Ether. This will let our faucet accept & store any Ether sent to its address.

- i) `receive () external payable {}`

- e) Then, declare a function for others to request Ether from our faucet.

- i) `function withdraw(uint withdraw_amount) public {`

- f) Then, require the amount asked to withdraw be 1000000000000000000 wei max (10 Ether).

- i) `require(withdraw_amount <= 1000000000000000000);`

- g) Lastly, transfer money from our faucet to the person who requested the money using the withdraw function.

- i) `payable(msg.sender).transfer(withdraw_amount);`

- ii) `}`

- iii) `}`

```

contracts > Faucet.sol
1  pragma solidity >=0.4.22 <0.9.0;
2  // SPDX-License-Identifier: UNLICENSED
3
4
5  contract Faucet {
6      receive () external payable {}
7
8      function withdraw(uint withdraw_amount) public {
9          require(withdraw_amount <= 1000000000000000000);
10         payable(msg.sender).transfer(withdraw_amount);
11     }
12 }
13

```

Figure 6, [Click for more Information](#)

2) Deal with Migrations

- a) Navigate to the migrations folder in your truffle project. Create a new file called "2_deploy_contracts.js".
- b) Copy and paste the code below (Note: This might be outdated. If so, open the js file starting with "1_" in your migrations folder, and copy and paste it to the "2_deploy_contracts.js". Then, replace the word "Migrations" with "Faucet")

```

const faucet = artifacts.require("Faucet");

module.exports = function (deployer) {
    deployer.deploy(faucet);
};

```

3) Deploy the Contract to the Local Blockchain

- a) Open your terminal and navigate to your Truffle project folder. Then, type “truffle deploy”. If you see an error, try using this command as an admin privileges. On MacOS, this is done by typing “sudo truffle deploy”. You should see something similar to Figure 7.

```
truffleTest2 -- zsh -- 89x56
=====
Deploying 'Migrations'
> transaction hash: 0x65dc56bdbabed981994fb4252282ff72a9481269f69dd1804c0bbf8f21068
6a9
> Blocks: 0
> contract address: 0xA7B4b9021bfAE404B1f7C43073e238892E706e7C
> block number: 1
> block timestamp: 1644542855
> account: 0x58676ba6996feE8B241ca29Cf922159eF15C3ad7
> balance: 99.99502316
> gas used: 248842 (0x3cc0a)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00497684 ETH

> Saving migration to chain.
> Saving artifacts
> Total cost: 0.00497684 ETH

2_deploy_contracts.js
=====
Deploying 'Faucet'
> transaction hash: 0xe2974ba1935b04e01f7569dec1bbf39753211a0594ceac5764571f77da07a
b0f
> Blocks: 0
> contract address: 0x57AfD5C272a7b30c2Ce1215008916746963baC02
> block number: 3
> block timestamp: 1644542856
> account: 0x58676ba6996feE8B241ca29Cf922159eF15C3ad7
> balance: 99.9916416
> gas used: 126565 (0x1ee65)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.0025313 ETH

> Saving migration to chain.
> Saving artifacts
> Total cost: 0.0025313 ETH

Summary
=====
> Total deployments: 2
> Final cost: 0.00750814 ETH

olsenbudanur@olsens-air truffleTest2 %
```

Figure 7

- b) Congrats, your contract is live on your local blockchain!! You can double check this by navigating to the “Contracts” tab on your Ganache app. It should look similar to Figure 8.

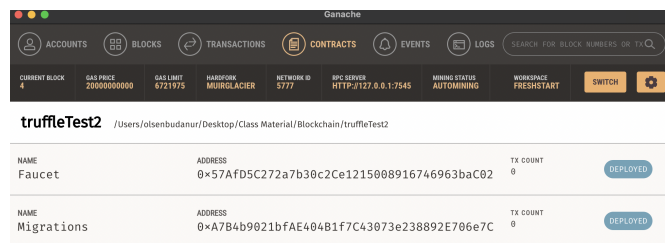


Figure 8

- 4) Send money to the Faucet Contract
 - a) Now that we created our faucet contract, let's interact with it! Since the faucet is empty right now, we cannot withdraw any Ether from it. So, let's deposit Ether to it using our first account.
 - b) Open the truffle console by typing "truffle console" to your terminal.
 - c) Store the account addresses to a variable using the following command:
 - i) `let accounts = await web3.eth.getAccounts()`
 - ii) Note: web3 is included with the truffle console. For more info about it, navigate to <https://web3js.readthedocs.io/en/v1.7.0/>
 - d) Store the Faucet contract as a variable using the following command:
 - i) `let instance = await Faucet.deployed()`
 - e) Now, send 10 Ether to the faucet from the first account. First, get the address of your faucet by navigation to the "Contracts" tab in your Ganache app. Then run the following command (replace faucet_address with your faucet address).
 - i) `web3.eth.sendTransaction({ from: accounts[0], to:'faucet_address', value:'1000000000000000000' });`
 - f) Now, your first account should've sent 10 ether to your faucet. Double check this by clicking on your faucet in the "Contracts" tab. See Figure 9.

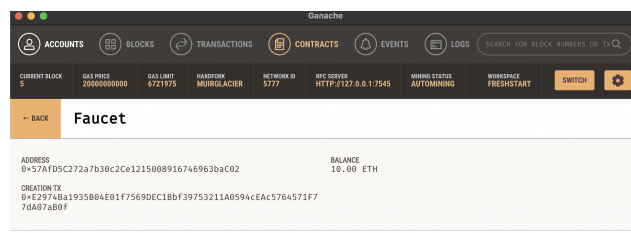


Figure 9

- 5) Withdraw money to the Faucet Contract
 - a) Let's ask the faucet for 0.1 Ether through the second account. To do this, type the following command in your terminal:
 - i) `instance.withdraw('1000000000000000000', {from: accounts[1]});`
 - b) Voila! You should see 0.1 more Ether in your second account, and 0.1 less Ether in the faucet.

ADDRESS	BALANCE	TX COUNT	INDEX	
0x0bCf0E168238EC57D5da0b12BFdCd0c38f9c73D3	100.10 ETH	5	1	

Figure 10

6) Further Reading

- a) To recap, we created a faucet contract that accepted Ether deposits and withdrawals. Then, we deployed it in our local blockchain. Lastly, we deposited Ether to it through one account, then withdrew Ether from it using another account. Fun stuff!
- b) This was by no means a comprehensive tutorial on Solidity/smart contracts. We just created a simple contract and played around with it to show what Truffle/Ganache is & as a proof of concept. You should read more to have a better grasp on this subject. Here are a few resources that we suggest (in order of importance):
 - i) <https://github.com/ethereumbook/ethereumbook>
 - ii) <https://link.springer.com/book/10.1007/978-981-15-6218-1>
 - iii) <https://drive.google.com/drive/u/1/folders/1nmrKVNXLhRAT9RKyhKTN5ZW8Rsp8SQI>
 - iv) <https://trufflesuite.com/docs/>

4- Creating An ERC-20 Token

Now that we are beginning to understand the basics of Solidity, we'll expand our knowledge by implementing an [ERC-20 token](#). ERC-20 is a smart contract standard/interface used to create fungible tokens. You can read more about the ERC-20 standards here [ERC-20 Token Standard | ethereum.org](#).

First, let's create this token from scratch in solidity. Later on in this chapter, we will use a library called OpenZeppelin to simplify the process of creating this token. Without further ado, let's get started!

Understanding the ERC-20 Interface

The ERC-20 Interface defines 6 mandatory functions, 3 optional functions, and 2 mandatory events. These are listed and explained below:

1) The 6 [Mandatory Functions](#):

- a) *function **totalSupply()** public view returns (uint256):*
 - i) Returns the total token supply.
- b) *function **balanceOf(address _owner)** public view returns (uint256 balance):*
 - i) Returns the account balance of another account with address _owner.
- c) *function **transfer(address _to, uint256 _value)** public returns (bool success):*
 - i) Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend.
- d) *function **transferFrom(address _from, address _to, uint256 _value)** public returns (bool success):*
 - i) Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event. The transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.
- e) *function **approve(address _spender, uint256 _value)** public returns (bool success):*
 - i) Allows _spender to withdraw from your account multiple times, up to the _value amount. If this function is called again it overwrites the current allowance with _value.
- f) *function **allowance(address _owner, address _spender)** public view returns (uint256 remaining):*
 - i) Returns the amount which _spender is still allowed to withdraw from _owner.

2) The 3 [Optional Functions](#):

a) *function **name()** public view returns (string):*

- i) Returns the name of the token - e.g. "MyToken". OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

b) *function **symbol()** public view returns (string):*

- i) Returns the symbol of the token. E.g. "HIX". OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

c) *function **decimals()** public view returns (uint8):*

- i) Returns the number of decimals the token uses - e.g. 8, means to divide the token amount by 100000000 to get its user representation. OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

3) The 2 [Mandatory Events](#):

a) event ***Transfer(address indexed _from, address indexed _to, uint256 _value):***

- i) MUST trigger when tokens are transferred, including zero value transfers. A token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created.

b) event ***Approval(address indexed _owner, address indexed _spender, uint256 _value):***

- i) MUST trigger on any successful call to approve(address _spender, uint256 _value).

Creating an ERC-20 Token From Scratch

Now that we understand the ERC-20 interface, we will write and deploy an ERC-20 Token from scratch. In reality, writing a token from scratch is bad practice. This is because there are industry-tried and tested libraries (like OpenZeppelin) that you can use to make sure that your code is as safe as it can be.

Anyways, since writing a token from scratch is tedious, we have done the work for you! You can just follow these steps, and download our ERC-20 token solidity file!

- 1) Create a Truffle Project
 - a) Refer to **"2- Connecting to a Local Ganache Blockchain using Truffle"** to create a Truffle/Ganache project.
- 2) Download the ERC-20 Token Code from our repository.
 - a) Download ERC20.sol file from our repository linked here(add link here). This is an implementation of the ERC 20 standard described above from scratch. **We suggest that you glance over this file, and try to somewhat understand how it works.**
- 3) Deploy the ERC20.sol File
 - a) Refer to **"3- Creating Our First Smart Contract"** to deploy the .sol file on your Truffle project.

You now have an ERC-20 token deployed on your Truffle project! You can use your Truffle console and the web3 library to play around with the ERC-20 contract!

Creating an ERC-20 Token With OpenZeppelin

OpenZeppelin is a library of industry tried and tested code that could be used for creating smart contracts, decentralized applications, and many other things.

Since the code on OpenZeppelin is used by thousands of users, using it is a lot safer than writing your own code. Although safer, any code might have vulnerabilities. So, use OpenZeppelin after consideration and at your own risk.

1) Introduction To OpenZeppelin

a) Install OpenZeppelin with this command (Need Node.js)

```
$ npm install @openzeppelin/contracts
```

Figure 11

b) Navigate to OpenZeppelin's Contract Wizard

- Go to <https://wizard.openzeppelin.com/> and create a custom ERC-20 contract! Notice how simple it is to create a contract here! Figure 12 shows how you can play around with this tool.
- After configuring your custom ERC-20 contract, copy the code.

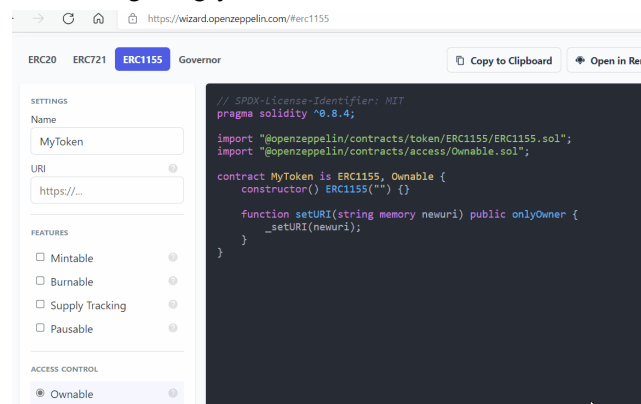


Figure 12

- Deploy this contract the way you deployed the ERC-20 contract above. You are done!

Additional:

1. How does OpenZeppelin work?
 - a. Notice we import "@openzeppelin/contracts/token/ERC20/ERC20.sol" to create an OpenZeppelin ERC-20 contract. You can check what we are importing if you just google this import statement! We are importing the code from this link: [HERE](#). Notice how this code is identical to the code we wrote for "Creating an ERC-20 Token From Scratch"!
2. Events were created in Solidity as a proposed result after a vulnerability in the design of some functions allowed a hacker to steal over three million dollars in BitCoin in what is known as the "Dao hack"
 - a. This is the original proposal to add the 'emit' keyword to Solidity [Proposal: add 'emit' keyword; make it the only way to emit events · Issue #2877 · ethereum/solidity \(github.com\)](#)
 - b. More about the DAO hack → [The DAO \(organization\) - Wikipedia](#)
3. A Simplified Guide to ERC-20 Tokens → [A Straightforward Guide to ERC20 Tokens - EthHub](#)
4. [How To Make Your Own Ethereum \(ERC-20\) Token In 30 Minutes - YouTube](#)
5. [Burn function ERC20 - Support - OpenZeppelin Community](#),
 - a. [ethereum-org/token-erc20.sol at master · ethereum/ethereum-org \(github.com\)](#)
 - b. [openzeppelin-contracts/ERC20.sol at master · OpenZeppelin/openzeppelin-contracts \(github.com\)](#)

5- Creating An ERC-721 Token

Similar to ERC-20, ERC-721 is an standard/interface to create tokens. However, while ERC-20 is to create fungible tokens, ERC-721 is the opposite. The ERC-721 is used to create [non-fungible tokens \(NFT's\)](#).

Just like for ERC-20, OpenZeppelin has tools to help you create an ERC-721 contract. We will start from creating an ERC-721 contract from scratch, then use OpenZeppelin.

Understanding the ERC-721 Interface

The ERC-721 Interface defines 8 mandatory functions, and 3 mandatory events. These are listed and explained below:

1) The 8 [Mandatory Functions](#):

- a) *function **balanceOf(address _owner)** external view returns (uint256):*
 - i) Returns the total number of NFT's owned by an address. This number is possibly zero, and all inquiries about the 0th address are thrown.
- b) *function **ownerOf(uint256 _tokenId)** external view returns (address):*
 - i) Returns the address for the owner of the token in question.
- c) *function **safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data)** external payable:*
 - i) Transfers an NFT from one owner to another. Throws error if _from doesn't actually own the token in question, or if _to is the zero address. Throws if _tokenId isn't a valid id. Data is just extra information sent to be received by the _to address.
- d) *function **transferFrom(address _from, address _to, uint256 _tokenId)** external payable*
 - i) Sends a token from wonder address if owned to _to address. Sender is responsible for making sure _to is a valid and correct address or else the token could be lost forever.
- e) *function **approve(address _approved, uint256 _tokenId)** external payable:*
 - i) Adds or reaffirms an approved address of an NFT. Throws if msg.sender isn't actually the owner of the NFT or an approved operator of the current owner
- f) *function **setApprovalForAll(address _operator, bool _approved)** external:*
 - i) Enables or removes permission for another address to help manage all of the NFT assets of 'msg.sender'. True if the the address is approved and false if permission is being revoked
- g) *function **getApproved(uint256 _tokenId)** external view returns (address):*
 - i) Throws if _tokenId is not a valid uint256id. It then returns the approved address for the NFT in question.

h) **function *isApprovedForAll*(address _owner, address _operator) external view returns (bool):**

- i) Returns true if the address _operator is an approved operator for _owner address. _owner is the address that owns the NFTs in question.

2) The 3 **Mandatory Events**

a) **event *Transfer*(address indexed _from, address indexed _to, uint256 indexed _tokenId):**

- i) Emits when _to, or _from address is the zero address. Permissions also must be reset if a token is sent from one valid address to another.

b) **event *Approval*(address indexed _owner, address indexed _approved, uint256 indexed _tokenId):**

- i) This emits when the approved address for an NFT is changed or reaffirmed. This also resets the approved addresses to none when an NFT is transferred.

c) **event *ApprovalForAll*(address indexed _owner, address indexed _operator, bool _approved):**

- i) This event emits when an operator is disabled or allowed for a given owner.

3) Where is the Minting Function?

- a) After you've read through all of the functions above, you likely noticed that there is no mint function within the ERC 721 standard. That method is left for every developer to determine for themselves how best to implement for their type of token.
- b) For reference, here is an implementation of the mint function used in OpenZeppelin:

```
238  /**
239   * @dev Safely mints `tokenId` and transfers it to `to`.
240   *
241   * Requirements:
242   *
243   * - `tokenId` must not exist.
244   * - If `to` refers to a smart contract, it must implement {IERC721Receiver-onERC721Received}, which is called upon a safe transfer.
245   *
246   * Emits a {Transfer} event.
247   */
248  function _safeMint(address to, uint256 tokenId) internal virtual {
249      _safeMint(to, tokenId, "");
250  }
251
252  /**
253   * @dev Same as {xref-ERC721-_safeMint-address-uint256-}[`_safeMint`], with an additional `data` parameter which is
254   * forwarded in {IERC721Receiver-onERC721Received} to contract recipients.
255   */
256  function _safeMint(
257      address to,
258      uint256 tokenId,
259      bytes memory _data
260  ) internal virtual {
261      _mint(to, tokenId);
262      require(
263          _checkOnERC721Received(address(0), to, tokenId, _data),
264          "ERC721: transfer to non ERC721Receiver implementer"
265      );
266  }
```

Figure 13


```

268     /**
269     * @dev Mints `tokenId` and transfers it to `to`.
270     *
271     * WARNING: Usage of this method is discouraged, use {_safeMint} whenever possible
272     *
273     * Requirements:
274     *
275     * - `tokenId` must not exist.
276     * - `to` cannot be the zero address.
277     *
278     * Emits a {Transfer} event.
279     */
280     function _mint(address to, uint256 tokenId) internal virtual {
281         require(to != address(0), "ERC721: mint to the zero address");
282         require(!_exists(tokenId), "ERC721: token already minted");
283
284         _beforeTokenTransfer(address(0), to, tokenId);
285
286         _balances[to] += 1;
287         _owners[tokenId] = to;
288
289         emit Transfer(address(0), to, tokenId);
290
291         _afterTokenTransfer(address(0), to, tokenId);
292     }

```

Figure 14

- c) Open Zeppelin begins to mint a token by calling `_safeMint`, which then calls a version of itself with memory `_data` set to "" if not specified. This `_safeMint` method ensures that an existing token is being transferred. That the token is not being transferred to a contract which doesn't inherit the appropriate interfaces to then be able to transfer said token.
- d) The `_mint` method called by `_safeMint` requires that the token is not being minted to the zero address and that the token Id isn't already minted.
 - i) Additional Reading on Mint -> [blockchain - Why does the minting function of ERC721 have an access control? - Stack Overflow](#)

Creating an ERC-721 Token From Scratch

For this part, follow the steps on "Creating an ERC-20 Token From Scratch" using the .sol file for the ERC-721 contract.

Creating an ERC-721 Token With OpenZeppelin

For this part, follow the steps on "Creating an ERC-721 Token With OpenZeppelin" selecting ERC-721 in the OpenZeppelin wizard.

Additional:

1. [OpenZeppelin documents for ERC-721](#)
2. [Official ERC-721 standard](#)
3. [How to Code a Crypto Collectible: ERC-721 NFT Tutorial \(Ethereum\)](#)
4. [What is ERC-721?](#)

6- Creating An ERC-1155 Token

The ERC-1155 standard is a combination of the ERC-20 and ERC-720. It is a smart contract that is designed to manage fungible, non-fungible and semi-fungible tokens. Each new token created through an ERC-1155 contract is able to have different attributes and metadata than other tokens minted from the same contract. Read more [here](#).

Understanding the ERC-1155 Interface

The ERC-1155 Interface defines 8 mandatory functions, and 4 mandatory events. These are listed and explained below:

1. The 8 [Mandatory Functions](#):
 - a.
2. The 4 [Mandatory Events](#):
 - a. ***event TransferSingle(address indexed _operator, address indexed _from, address indexed _to, uint256 _id, uint256 _value):***
 - i. Transfers _value amount of _id type tokens from _from address as long as __operator is approved to make transactions on their behalf.
 1. When minting __from must be set to the zero address
 2. When burning __to must be set to the zero address
 - b. ***event TransferBatch(address indexed _operator, address indexed _from, address indexed _to, uint256[] _ids, uint256[] _values):***
 - i. Transfers _values amount of type __ids tokens with indices matching respectively to __to address from __from balance as long as the request is being sent by __operator who is approved to act on these funds.
 1. When minting __from must be set to the zero address
 2. When burning __to must be set to the zero address
 - c. ***event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved):***
 - i. Sets the _approved permission (true or false) for _operator over the tokens of __owner
 - d. ***event URI(string _value, uint256 indexed _id):***
 - i. Is emitted when the URI for token type _id is changed to __value
- **function safeTransferFrom(address _from, address _to, uint256 _id, uint256 _value, bytes calldata _data) external;**
 - Transfers __value amount of token type __id to __to address from __from address with additional __data if __to is the address of another contract
 - MUST revert if:

- `__to` is zero address
 - balance for `__from` is less than `__value` requested to be sent
 - On any other error
 - MUST
 - Emit the Transfer Single event to show balance change
 - **function** `safeBatchTransferFrom(address __from, address __to, uint256[] calldata __ids, uint256[] calldata __values, bytes calldata __data) external;`
 - Transfers `__values` amount of `__ids` from `__from` to `__to` address with `__data` and initial safety call
 - MUST revert if
 - `__to` is zero address
 - Length of `__ids` and `__values` aren't equal
 - balance for and of `__ids` is lower than requested value in `__values`
 - Or any other error
 - MUST
 - Emit TransferSingle or TransferBatch event to reflect balances
 - **function** `balanceOf(address __owner, uint256 __id) external view returns (uint256);`
 - Returns the number of `__id` type tokens owned by address `__owner`
 - **function** `balanceOfBatch(address[] calldata __owners, uint256[] calldata __ids) external view returns (uint256[] memory);`
 - Returns the an array of balances for each owner in `__owners` balance of type `id` token in `__ids`
 - **function** `setApprovalForAll(address __operator, bool __approved) external;`
 - Approves or removes ability through `__approved` for `__operator` to manage `msg.sender`'s token balances
 - **function** `isApprovedForAll(address __owner, address __operator) external view returns (bool);`
 - Returns true or false depending on `__operators` permissions to act on behalf of tokens owned by `__owner`
-
- A contract must implement all of the following functions in order to be able to receive a token
 - **function** `onERC1155Received(address __operator, address __from, uint256 __id, uint256 __value, bytes calldata __data) external returns(bytes4);`
 - Compliant contracts make this call at the end of SafeTransferFrom after the balance has been updated. If received, must return
 - ``bytes4(keccak256("onERC1155Received(address,address,uint256,uint256,bytes))")``
 - Otherwise, the call is rejected and must revert

- **function** onERC1155BatchReceived(**address** _operator, **address** _from, **uint256[]** calldata _ids, **uint256[]** calldata _values, **bytes** calldata _data) **external returns(bytes4)**;
 - Compliant contracts make this call at the end of SafeTransferFrom after the balance has been updated. If received, must return
 - ``bytes4(keccak256("onERC1155Received(address,address,uint256,uint256,bytes)")``
 - Otherwise, the call is rejected and must revert

Additionally reading on the ERC 1155 can be found:

[EIP-1155: Multi Token Standard \(ethereum.org\)](#)

[ERC 1155 - OpenZeppelin Docs](#)

[ERC1155 - Exploring the ERC-1155 Token Standard » Moralis - The Ultimate Web3 Development Platform](#)

[What is the ERC-1155 Token Standard? - YouTube](#)

165

ERC 165-

<https://medium.com/@chiqing/ethereum-standard-erc165-explained-63b54ca0d273>

The ERC-165 standard is a forward thinking interface that helps a user identify what version of a certain standard is being implemented.

A contract that is ERC-165 compliant will implement an interface with the single function

- **function supportsInterface(bytes4 interfaceID) external view returns (bool);**
 - Returns true if the given function implements the interfaceID supplied
 - MUST not use more than 30,000 gas
- Any contract that wants to have an interface Id must utilize a selector contract to calculate the Id of all functions in a given interface following the use of the XOR expression in Solidity '^' as demonstrated below

```
pragma solidity ^0.4.20;

interface Solidity101 {
    function hello() external pure;
    function world(int) external pure;
}

contract Selector {
    function calculateSelector() public pure returns (bytes4) {
        Solidity101 i;
        return i.hello.selector ^ i.world.selector;
    }
}
```

And again used here

```
// Selector.sol

pragma solidity 0.5.8;

import "./StoreInterface.sol";

contract Selector {
    // 0x75b24222
    function calcStoreInterfaceId() external pure returns (bytes4) {
        StoreInterface i;
        return i.getValue.selector ^ i.setValue.selector;
    }
}
```

- By making an interface object `i` and then calling `i.functionx.selector` it is calculating a unique `bytes4(keccak256('supportsInterface(bytes4)'))`; value
- The XOR ensures that each one function can be independently identified.

References:

[EIP-165: Standard Interface Detection \(ethereum.org\)](#)

[Ethereum Tokens: ERC165 \(smart contract interfaces\) - YouTube](#)

[Ethereum Standard ERC165 Explained | by Leo Zhang | Medium](#)

SECURITY:

rekt.news