Conflux Developer Workshop – Session 3

Conflux Developer Workshop – Session 3

Lesson 5 – Smart Contract Development I (Token Standards)

Token Standards

The Power of Tokens: Redefining Digital Value Exchange on the Blockchain

Tokens are undoubtedly one of the most powerful tools in the realm of blockchain technology, enabling a new paradigm of digital value exchange. Essentially, a token represents a unit of value within the blockchain network, which could denote anything from currency to services, assets, or even intangible entities like reputation or voting rights. By leveraging tokens to symbolize such entities, smart contracts can interact with them, exchange them, and even generate or destruct them as needed. This particular feature holds immense potential for the blockchain industry, facilitating a decentralized system of transactions and contracts without the need for intermediaries.

To utilize a token effectively, it is imperative to implement a corresponding smart contract. This contract is separate from the token itself, and instead contains a range of methods that users can leverage to interact with the token. These users can be service providers, merchants, insurance companies, content creators, game developers, or any other entity that needs to engage with the token in some way. For instance, a merchant may use the smart contract's payment method to receive payment in tokens, while a content creator may use the smart contract's royalty method to receive a percentage of the revenue generated by their content. At its core, the token smart contract functions as an address-to-balance mapping that can be modified through the contract's established methods. These balances are representative of the actual tokens, with token possession determined by the non-zero balance within the token contract. Balances may take on various forms, such as carbon credits, intellectual property rights, energy units, or loyalty points. Each token resides within a designated token contract, emphasizing the pivotal role of token contracts within the blockchain ecosystem. In sum, tokens and their accompanying smart contracts have the potential to revolutionize the way we exchange value and interact with digital assets, providing a secure, efficient, and decentralized platform for a wide range of transactions and contracts.

Coins vs Tokens

While crypto coins and tokens share some similarities, it's important to note that they are not the same. The key difference between the two is that crypto coins are a native cryptocurrency of their blockchain, whereas tokens operate on other blockchains. To provide an example, Bitcoin (BTC) is a coin since it is a cryptocurrency native to the Bitcoin blockchain. In contrast, DAI is a token as it does not have its blockchain but rather runs on the Ethereum blockchain. Understanding this distinction is vital to comprehend the complexities of the cryptocurrency market and its underlying technologies.

Fungible vs. Non-Fungible Tokens

A fungible token is a cryptocurrency or digital asset that is easily exchangeable and interchangeable with other tokens of the same type and value. The term "fungible" means that one unit of the token has the same value as another unit of the same token, and can be divided into smaller units without losing its value. In contrast, a non-fungible token (NFT) is a unique digital asset that is indivisible, non-interchangeable, and irreplaceable. Each NFT has a unique ID that makes it easily distinguishable from other tokens, and it is owned by only one person. The value of an NFT is based on its uniqueness and scarcity, and it cannot be replaced or exchanged for another NFT without losing its value. Examples of NFTs include unique digital art, collectibles, and other one-of-a-kind digital assets.

Unpacking Ethereum's Token Standards: ERC-20, ERC-721, ERC-777, and ERC-1155

Despite their apparent simplicity, tokens can be quite complex in their implementation. The Ethereum platform operates solely on smart contracts, with no inherent restrictions on what these contracts can do. As a result, the Ethereum community has established a range of standards, known as EIPs, short for Ethereum Improvement Proposals ERCs short for Ethereum Request for Comments, to define how contracts can interact with each other.

Among the existing token standards, ERC-20 is currently the most widely adopted for fungible assets. However, its simplicity also limits its potential applications. For non-fungible tokens, ERC-721 is widely considered the standard solution, primarily utilized for collectibles and games. In contrast, ERC-777 is a more advanced standard for fungible tokens, enabling new possibilities while building on past experiences. In addition, ERC-777 is backward compatible with ERC-20. Moreover, the ERC-1155 standard is a revolutionary multi-token standard that allows a single contract to represent

various fungible and non-fungible tokens. This standard also allows for batched operations, increasing transaction efficiency and gas savings. By permitting multiple tokens to be represented within one contract. ERC-1155 greatly simplifies token management, lowering the complexity of developing and managing token-based systems. Now let’s take a closer look into each one of the standards.

ERC-20

While Ether (ETH) is the native cryptocurrency of the Ethereum network, the ERC-20 token represents a specific standard. Developers can follow this standard to make Ethereum-based tokens. These tokens are a form of token that can be issued on Ethereum and represent a common list of rules that all tokens on the Ethereum blockchain must abide by. As a result, they are designed to work with smart contracts, and all tokens that adhere to the ERC-20 standard can be used interchangeably.

The ERC-20 standard has made many initial coin offerings (ICOs) possible in recent years, and the standard makes it easy for developers to create decentralized applications (dApps) on Ethereum. ERC-20 tokens are used as a means of exchange within these dApps, allowing for easy and seamless transactions. Additionally, since the ERC-20 standard has been adopted by the Ethereum network, it has served as a guiding light for Ethereum developers.

ERC-20 is only for fungible tokens, and not non-fungible tokens (NFTs). As such, one ERC-20 token can be exchanged with another as they have equal value. ERC-20 smart contracts use ERC-20 tokens to facilitate transactions when its protocol calls for it. Any smart contract that utilizes transaction functionality will therefore pay the user in the form of an ERC-20 token.

Many popular stablecoins, like USDC and DAI, are ERC-20 tokens. Stablecoins are designed to be more “stable” than other, often volatile cryptos. They are popular among traders looking to lock in profits quickly without converting to fiat currency, as well as those seeking to earn a yield on their crypto.

Several decentralized finance (DeFi) and metaverse tokens are also ERC-20 tokens. Decentraland (MANA) and Enjin Coin (ENJ) help users perform functions or create items in video games and virtual or augmented realities. Uniswap (UNI), the native token of one of the largest DeFi platforms, allows users to borrow and lend funds to one another.

The ERC-20 protocol specifies the essential features that Ethereum-based tokens should have and how they should function. Tokens that don’t comply cannot be issued, traded, or listed on exchanges. The ERC-20 standard has nine rules in total, with six of them being mandatory and three optional. The

mandatory rules include TotalSupply, Approve, BalanceOf, TransferFrom, Transfer, and Allowance.

The TotalSupply rule outlines the total number of tokens to be created. The Approve rule helps to eliminate the possibility of counterfeit tokens being created by requiring approval of smart contract functions. The BalanceOf rule allows users to check their balances by returning the total number of tokens held by an address. The TransferFrom rule allows for the automation of transactions when desired. The Transfer rule allows for the transfer of tokens from one address to another, like any other blockchain-based transaction. Now let’s take a closer look into the syntax and key points regarding each of the methods and events that are defined for an ERC-20 token contract:

"name" method: This method returns the name of the token as a string. It is optional and is not expected to be present by interfaces and contracts.

function name() public view returns (string)

"symbol" method: This method returns the symbol of the token as a string. It is optional and is not expected to be present by interfaces and contracts.

function symbol() public view returns (string)

"decimals" method: This method returns the number of decimals the token uses, which is used to represent the token amount. It is optional and is not expected to be present by interfaces and contracts.

function decimals() public view returns (uint8)

//"totalSupply" method: This method returns the total supply of tokens.

function totalSupply() public view returns (uint256)

"balanceOf" method: This method returns the balance of the specified account with address _owner.

function balanceOf(address _owner) public view returns (uint256 balance)

"transfer" method: This method transfers _value amount of tokens to address _to and emits the Transfer event. If the message caller's account balance is not enough to spend the amount, the function should throw an error. Transfers of 0 values are treated as normal transfers and emit the Transfer event.

function transfer(address _to, uint256 _value) public returns (bool success)

"transferFrom" method: This method transfers _value amount of tokens from address _from to address _to and emits the Transfer event. It is used for a withdraw workflow and can be used to allow contracts to transfer tokens on behalf of the user. The function should throw an error unless _from has authorized the sender via some mechanism. Transfers of 0 values are treated as normal transfers and emit the Transfer event.

function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)

"approve" method: This method allows _spender to withdraw from the caller's account multiple times, up to the _value amount. If this function is called again, it overwrites the current allowance with _value. Clients should set the allowance first to 0 before setting it to another value for the same spender to prevent attack vectors. The contract should not enforce it to allow backwards compatibility with contracts deployed before.

function approve(address _spender, uint256 _value) public returns (bool success)

"allowance" method: This method returns the amount which _spender is still allowed to withdraw from _owner.

function allowance(address _owner, address _spender) public view returns (uint256 remaining)

"Transfer" event: This event is triggered when tokens are transferred, including zero value transfers. If a token contract creates new tokens, it should trigger a Transfer event with _from address set to 0x0.

event Transfer(address indexed _from, address indexed _to, uint256 _value)

"Approval" event: This event is triggered on any successful call to approve.

event Approval(address indexed _owner, address indexed _spender, uint256 _value)

You can also find example implementations of ERC-20 at OpenZeppelin implementation and ConsenSys implementation.

ERC-721

ERC-721 is an open standard that has become a pillar of the Ethereum ecosystem, allowing the creation of unique and verifiably cryptographic tokens known as Non-Fungible Tokens (NFTs). The most common use case for ERC-721 NFTs is for digital art, where users buy these tokens to support artists, invest long-term, or quickly flip/trade NFTs for a profit. However, use cases for NFTs extend beyond digital art, such as in blockchain-based games to represent unique assets within the game, and in real-world use cases, such as the ability to mint the deed to your house as an NFT to make real estate transactions more efficient.

In this part, we will take a closer look at some of the noteworthy functions and events defined in the ERC-721 standard and explore its use cases.

ERC-721 defines a set of functions with compliance to ERC-20 to make it easier for existing wallets to display simple token information. These functions include "name", which defines the token's name, "symbol", which defines the token's shorthand name, and "totalSupply", which defines the total number of tokens on the blockchain.

```
function name() external view returns (string _name)
```

```
function symbol() external view returns (string _symbol)
```

```
function totalSupply() external view returns (uint256)
```

“balanceOf”: One of the key functions of ERC-721 is "balanceOf", which returns the number of NFTs owned by a given address. This function is essential for tracking ownership and enables the transfer of tokens between users through the "transfer" function.

```
function balanceOf(address _owner) external view returns (uint256)
```

“ownerOf”: The "ownerOf" function returns the address of the owner of a token. Since each ERC-721 token is unique and non-fungible, it is represented on the blockchain by an ID. Other users, contracts, and applications can use

this ID to determine the owner of the token.

function ownerOf(uint256 _tokenId) external view returns (address)

“safeTransferFrom”: This function transfers the ownership of an NFT from one address to another address. It throws an exception if the transfer fails, and can also call the onERC-721Received function on the recipient address.

function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data) external payable

“safeTransferFrom”: This function works identically to the previous one, but without the data parameter.

function safeTransferFrom(address _from, address _to, uint256 _tokenId) external payable

“transferFrom”: This function transfers the ownership of an NFT from one address to another address. It throws an exception if the transfer fails.

function transferFrom(address _from, address _to, uint256 _tokenId) external payable

"approve": The "approve" function grants or approves another entity permission to transfer tokens on the owner's behalf. This function is essential for the creation of marketplaces where users can buy and sell NFTs.

function approve(address _approved, uint256 _tokenId) external payable

“setApprovalForAll”: This function enables or disables approval for a third party ("operator") to manage all of an owner's assets.

function setApprovalForAll(address _operator, bool _approved) external

“getApproved”: this function retrieves the approved address for a specific NFT and will throw an error if the provided _tokenId is not a valid NFT. The

function takes _tokenId as a parameter and returns the approved address for that NFT or the zero address if there is none.

function getApproved(uint256 _tokenId) external view returns (address)

“isApprovedForAll”: This function checks if a given address _operator is authorized to act on behalf of another address _owner. It takes _owner and _operator as parameters and returns true if _operator is an approved operator for _owner and false otherwise.

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

“tokenOfOwnerByIndex”: This is an optional but recommended function. Each owner can own more than one NFT at the same time. Its unique ID identifies every NFT, and eventually, it can become difficult to keep track of IDs. So the contract stores these IDs in an array and the tokenOfOwnerByIndex function lets us retrieve this information from the array.

function tokenOfOwnerByIndex(address _owner, uint256 _index) external view returns (uint256)

ERC-721 also defines three events.

“Transfer”: This event is emitted whenever ownership of an NFT changes, either because it has been transferred from one address to another, or because it has been created or destroyed.

event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId)

“Approval”: This event is emitted whenever the approved address for an NFT is changed or reaffirmed.

event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId)

“ApprovalForAll”: This event is emitted whenever an operator is enabled or disabled for an owner.

event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved)

While ERC-721 has limited functionality when it comes to interacting with other standards like ERC-20, and it can cause network congestion, leading to high transaction fees that can make them unreasonably expensive to purchase, ERC-721 will hold a place in the future of NFTs. As the use cases of NFTs continue to expand and evolve, ERC-721 will continue to play a vital role in the Ethereum ecosystem.

ERC-777

ERC-777 is a token standard for fungible tokens introduced on the Ethereum network. It is similar to the ERC-20 standard but offers more complex interactions when trading tokens. The ERC-777 standard brings tokens and Ether closer together by providing the equivalent of a msg.value field, but for tokens. In addition to this, it provides quality-of-life improvements such as getting rid of the confusion around decimals, minting and burning with proper events, among others. The ERC-777 standard's killer feature is receive hooks, which allows accounts and contracts to react to receiving tokens.

ERC-777 is fully compatible with existing decentralized exchanges and is backwards compatible with ERC-20. This means that existing ERC-20 tokens can be interacted with as if they were ERC-20 tokens, while still getting all the niceties, including send hooks. ERC-777 tokens are significantly less likely to get stuck in a contract, which is traditionally seen as a problem with ERC-20 tokens.

ERC-777 offers several benefits that distinguish it from ERC-20. Firstly, it functions similarly to Ether, allowing tokens to be sent via the function send (dest, value, data). Secondly, a tokensReceived code is specified in a contract or ordinary address that is activated when tokens are received. This feature streamlines how accounts and contracts communicate when receiving tokens, which is not the case with ERC-20 tokens. Thirdly, ERC-777 employs the new ERC-820 standard, which enables the registration of metadata for contracts to enable a basic form of introspection. This backward compatibility allows for additional functionality expansions.

Another advantage of ERC-777 is the introduction of a new transfer function that includes a field called bytes, allowing for the addition of any identification information to the transfer. This feature instantly notifies the recipient contract that the transfer occurred, enabling anyone to add extra functionality to tokens, such as a mixer contract for greater transaction confidentiality, or an emergency recovery feature to help if you lose your private keys.

ERC-777 had some early security problems that have since been resolved, but it may still have certain drawbacks, like any other token. Although it enables the transmission of tokens to any Ethereum address, it also implies that tokens can be sent to contracts that do not support them, thereby locking them forever. This problem is mitigated by the fact that the standard requires contracts to implement receive hooks to receive tokens, so no tokens can get stuck in a contract that is unaware of the ERC-777 protocol. However, it is still difficult to determine which tokens originated from whom and who sent them back, even if the tokens may be manually moved.

Here’s the syntax for the mandatory functions and events of ERC-777 that should be implemented:

```
function name() external view returns (string memory)

function symbol() external view returns (string memory)

function totalSupply() external view returns (uint256)

function balanceOf(address holder) external view returns (uint256)

function granularity() external view returns (uint256)

function defaultOperators() external view returns (address[] memory)

function isOperatorFor(address operator, address holder) external view
returns (bool)

function authorizeOperator(address operator) external

function revokeOperator(address operator) external

function send(address to, uint256 amount, bytes calldata data) external

function operatorSend(address from, address to, uint256 amount, bytes
calldata data, bytes calldata operatorData) external

function burn(uint256 amount, bytes calldata data) external

function operatorBurn(address from, uint256 amount, bytes calldata data,
bytes calldata operatorData) external
```

```
event Sent(address indexed operator, address indexed from, address indexed
to, uint256 amount, bytes data, bytes operatorData)

event Minted(address indexed operator, address indexed to, uint256 amount,
bytes data, bytes operatorData)

event Burned(address indexed operator, address indexed from, uint256 amount,
bytes data, bytes operatorData)


event AuthorizedOperator(address indexed operator, address indexed holder)

event RevokedOperator(address indexed operator, address indexed holder)
```

ERC-1155

ERC-1155, is a novel token standard that combines the best aspects of previous standards to create a fungibility-agnostic and gas-efficient token contract. Prior to the creation of ERC-1155, fungible tokens were represented by ERC-20 and non-fungible tokens by ERC-721. However, these standards were limited in that they could not be combined in the same smart contract, making the transfer of multiple token types in a single transaction impossible. This issue made executing such transactions inefficient and costly. ERC-1155's approach significantly reduces deployment costs and complexity, making it an ideal solution for projects that require multiple tokens.

The distinctive feature of ERC-1155 is that it uses a single smart contract to represent multiple tokens simultaneously. This means that its balanceOf function differs from ERC-20 and ERC-777's because it has an additional id argument that identifies the token to query the balance of. Similarly, ERC-721's balanceOf function refers to how many different tokens an account has, not how many of each.

ERC-1155 balances each account for every token ID, even for non-fungible tokens. For example, for an NFT, the contract will simply mint a single token ID. This approach leads to significant gas savings as it eliminates the need to deploy a new contract for each token type.

The ERC-1155 standard provides two functions, balanceOfBatch and safeBatchTransferFrom, that make querying multiple balances and transferring multiple tokens simpler and less gas-intensive. The safeBatchTransferFrom function has a built-in safety feature that ensures transactions are only executed if all transfers can be completed successfully, thus preventing failed transactions.

In addition to supporting fungible and non-fungible tokens, ERC-1155 also supports semi-fungible tokens. These tokens are like general admission concert tickets. They are interchangeable and can be sold for money before the show (fungible). But after the show, they lose their pre-show value and become collectibles (non-fungible).

ERC-1155 has no limit on the number of tokens it can support, making it ideal for use cases such as gaming, where large numbers of items are traded between players. By using ERC-1155, developers can efficiently store and manage large numbers of items in a single contract, significantly reducing costs and increasing efficiency.

Here’s the syntax for the mandatory functions and events of ERC-1155 that should be implemented:

```
event TransferSingle(address indexed _operator, address indexed _from, address indexed _to, uint256 _id, uint256 _value)
```

```
event TransferBatch(address indexed _operator, address indexed _from, address indexed _to, uint256[] _ids, uint256[] _values)
```

```
event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved)
```

```
event URI(string _value, uint256 indexed _id)
```

```
function safeTransferFrom(address _from, address _to, uint256 _id, uint256 _value, bytes calldata _data) external
```

```
function safeBatchTransferFrom(address _from, address _to, uint256[] calldata _ids, uint256[] calldata _values, bytes calldata _data) external
```

```
function balanceOf(address _owner, uint256 _id) external view returns (uint256)
```

```
function balanceOfBatch(address[] calldata _owners, uint256[] calldata _ids) external view returns (uint256[] memory)
```

```
function setApprovalForAll(address _operator, bool _approved) external
```

```
function isApprovedForAll(address _owner, address _operator) external view returns (bool)
```

Lesson 6 – Web3 Development I (Web3 APIs) - Interacting with Smart Contracts

Exploring Essential Libraries for Web3 Development

To interact with the Conflux blockchain through a web app, it is necessary to connect to a Conflux node. This can be done using the JSON-RPC specification, which is implemented by every Conflux client, providing a uniform set of methods for applications to rely on.

While it is possible to use vanilla JavaScript to connect with a Conflux node, there are convenient libraries within the Conflux ecosystem that simplify the process. These libraries allow developers to write intuitive, one-line methods to initialize JSON-RPC requests that interact with Conflux, abstracting away much of the complexity of directly interacting with a Conflux node and providing utility functions for easier development.

Some of the features offered by these libraries include the ability to connect to Conflux nodes using providers, such as JSON-RPC, Unifra, ConfluxScan, or MetaMask, as well as wallet functionality for creating wallets, managing keys, and signing transactions. Additionally, these libraries enable interaction with smart contract functions by reading the Application Binary Interface (ABI) of a compiled contract. The ABI is a JSON format that explains the functions of the contract and allows developers to use it like a normal JavaScript object.

Utility functions offered by these libraries also provide handy shortcuts for building with Conflux, such as converting CFX values to DRIP, as 1 CFX is equal to 1,000,000,000,000,000,000 DRIP, and working with numbers in this format can be challenging. For example, the web3.utils.toWei function can be used to convert CFX to DRIP.

Some of the most popular libraries for interacting with Conflux using JavaScript include Web3.js, which is the Ethereum JavaScript API, and Ethers.js, which is a complete Ethereum wallet implementation and utilities in JavaScript and TypeScript.

Here are the links to the documentation and GitHub repositories of the most popular libraries:

Web3.js

Web3.js is a comprehensive set of libraries that provides seamless interactions with Ethereum nodes, whether local or remote, through various protocols such as HTTP, IPC, or WebSocket. It offers a wide range of functionalities for developers to interact with the Ethereum blockchain, making it a versatile tool for building decentralized applications (dApps) and working with Ethereum-based projects.

Documentation: https://web3js.readthedocs.io/en/v1.8.2/

Web3.py

Web3.py is a widely used Python library that facilitates interactions with the Ethereum blockchain, particularly in decentralized applications (dApps). It provides functionalities for sending transactions, interacting with smart contracts, reading block data, and more. While its initial API was inspired by Web3.js, it has evolved to cater to the preferences and requirements of Python developers, offering a Pythonic way of interacting with Ethereum.

Documentation: https://web3py.readthedocs.io/

Ethers.js

The ethers.js library is designed to be a comprehensive and concise solution for interacting with the Ethereum Blockchain and its ecosystem. It is widely utilized for building decentralized applications (dApps), creating wallets (e.g., MetaMask and Tally), and developing other tools and scripts that involve reading and writing to the blockchain. The documentation provides detailed information on how to effectively leverage the functionalities offered by ethers.js in a variety of use cases.

Documentation: https://docs.ethers.org/v6/

Web3 Libraries Overview

Web3.py and Web3.js are a set of libraries that facilitate the interaction of developers with Ethereum nodes via the HTTP, IPC, or WebSocket communication protocols, using the Python and JavaScript programming languages respectively. This tutorial will provide you with the expertise to leverage the Web3.py and Web3.js libraries for transmitting transactions and deploying smart contracts on the Ethereum network.

Initiate a Python Project

To embark on your journey, first create a directory where all the relevant files generated throughout this guide can be stored. Execute this task with the following command:

```
mkdir web3-examples && cd web3-examples
```

For the successful implementation of the upcoming sections, you'll need to procure and install the Web3 library and the Solidity compiler. To obtain both packages, execute the following command:

Python:

```
pip3 install web3 py-solc-x
```

JavaScript:

```
npm install web3 solc@0.8.0
```

Setup the HTTP connection

Prepare your Web3 HTTP connection to align with any evm-powered network. To synchronize your project, you must secure an endpoint and API key of your own. Here's how you can get started with each network, step by step.

Python:

Import Web3 library into your project:

```
from web3 import Web3
```

Then, to configure the HTTP connection to your RPC endpoint, insert the provided RPC URL into this code:

```
web3 = Web3(Web3.HTTPProvider('RPC-API-ENDPOINT-HERE'))
```

JavaScript:

Utilize the following code to create a Web3 instance:

```
const Web3 = require('web3');
```

Insert your RPC URL into this code snippet to establish an HTTP connection to your RPC endpoint:

```
const web3 = new Web3('RPC-API-ENDPOINT-HERE');
```

You can find Conflux eSpace Network Endpoints here: https://doc.confluxnetwork.org/docs/espace/build/network-endpoints/

Send a Transaction and check the balances

In this section, you will learn how to create two scripts in order to send a transaction between two accounts. The first script will be used to check the balances of the accounts before and after the transaction is sent. The second script will actually send the transaction.

Check the balances

Python:

To begin with the first script, you need to create a file called "balances.py" using the command "vim balances.py". In this file, you will set up the Web3 provider, define the variables "sender_address" and "recipient_address", and retrieve the balances of these accounts using the "web3.eth.get_balance" function. You will then format the results using the

"web3.fromWei" function and print the balances in your terminal using the "print" function.

To run the balances script, enter the command "python3 balances.py" in your terminal. The balances for the sender and recipient addresses will be displayed in your terminal in ETH.

Put the Web3 provider snippet

```
from web3 import Web3

web3 = Web3(Web3.HTTPProvider('RPC-API-ENDPOINT-HERE'))
```

Define address variables

```
sender_address = "SENDER-ADDRESS-HERE"

recipient_address = "RECIPIENT-ADDRESS-HERE"
```

Retrieve balance data

```
balance_sender = web3.fromWei(web3.eth.get_balance(sender_address), "ether")

balance_recipient = web3.fromWei(web3.eth.get_balance(recipient_address), "ether")
```

Display balances in the terminal

```
print(f"The balance of { sender_address } is: { balance_sender } ETH")

print(f"The balance of { recipient_address } is: { balance_recipient } ETH")
```

JavaScript:

To check the account balances before and after the transaction, you just need to create a single file. You can start by creating a balances.js file by executing:

```
vim balances.js
```

After that, you can create the script for the file.Here's the code for the script:

Add the Web3 provider snippet

```
const Web3 = require('web3');

const web3 = new Web3('RPC-API-ENDPOINT-HERE');
```

Create address variables

```
const sender_address = 'ADDRESS-FROM-HERE';

const recipient_address = 'ADDRESS-TO-HERE';
```

Create balances function

```
const balances = async () => {
```

Get balance info

```
const balance_sender = web3.utils.fromWei(await
web3.eth.getBalance(sender_address), 'ether');

const balance_recipient = web3.utils.fromWei(await
web3.eth.getBalance(recipient_address), 'ether');

console.log(The balance of ${sender_address} is: ${balance_sender} ETH);

console.log(The balance of ${recipient_address} is: ${balance_recipient}
ETH);

};
```

Call balances function

```
balances();
```

To fetch the account balances, simply run the following command:

```
node balances.js
```

If successful, the balances for the origin and receiving addresses will be displayed in your terminal in ETH.

Send a transaction

Python:

To begin with the second script, you need to create a "transaction.py" file using the command "vim transaction.py". Within this file, you will import "rpc_gas_price_strategy" to obtain the gas price for the transaction. Then you will establish the Web3 provider, define the "sender" and "receiver" variables (including the private key for the "sender" account), establish the gas price strategy using the "web3.eth.set_gas_price_strategy" function, create and sign the transaction using the "web3.eth.account.sign_transaction" function, and send the transaction using the "web3.eth.send_raw_transaction" function. You will then wait for the transaction receipt using the "web3.eth.wait_for_transaction_receipt" function and print the transaction hash in your terminal.

To execute the transaction script, run the command "python3 transaction.py" in your terminal. If the transaction is successful, the transaction hash will be displayed in your terminal. You can also use the balances script to verify that the balances for the origin and receiving accounts have changed.

Import the gas strategy

```
from web3.gas_strategies.rpc import rpc_gas_price_strategy
```

Put the Web3 provider snippet here

```
from web3 import Web3

web3 = Web3(Web3.HTTPProvider('RPC-API-ENDPOINT-HERE'))
```

```
Define variables for addresses

sender = {

"private_key": "YOUR-PRIVATE-KEY-HERE",

"address": "PUBLIC-ADDRESS-OF-PK-HERE",

}

receiver = "ADDRESS-TO-HERE"

print(f'Attempting to send transaction from { sender["address"] } to {
receiver}')



Establish the gas price strategy

web3.eth.set_gas_price_strategy(rpc_gas_price_strategy)



Sign transaction with private key

tx_create = web3.eth.account.sign_transaction(

{

"nonce": web3.eth.get_transaction_count(sender["address"]),

"gasPrice": web3.eth.generate_gas_price(),

"gas": 21000,

"to": receiver,

"value": web3.toWei("1", "ether"),

},

sender["private_key"],

)



Send transaction and wait for receipt

tx_hash = web3.eth.send_raw_transaction(tx_create.rawTransaction)

tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
```

```
print(f"Transaction successful with hash: { tx_receipt.transactionHash.hex()
}")
```

JavaScript:

The steps for JavaScript are almost the same as Python. Create a .js file by "vim transaction.js" and fill it with the code below. By "node transaction.js" you can execute it.

```
Add the Web3 provider snippet

const Web3 = require('web3');

const web3 = new Web3('RPC-API-ENDPOINT-HERE');


Create account variables

const sender = {

  privateKey: 'YOUR-PRIVATE-KEY-HERE',

  address: 'PUBLIC-ADDRESS-OF-PK-HERE',

};

const receiver = 'ADDRESS-TO-HERE';



Create send function

const send = async () => {

  console.log(`Attempting to send transaction from ${sender.address} to
${receiver}`);



Sign tx with PK

  const createTransaction = await web3.eth.accounts.signTransaction(

    {

      gas: 21000,

      to: receiver,

      value: web3.utils.toWei('1', 'ether'),
```

```
    },
    sender.privateKey
  );
```

Send tx and wait for receipt

```
  const createReceipt = await
web3.eth.sendSignedTransaction(createTransaction.rawTransaction);
  console.log(`Transaction successful with hash:
${createReceipt.transactionHash}`);
};
```

Call send function

```
send();
```

Deploy a Contract

Initialize a Smart Contract

Within the following sections, you will be initializing and executing a straightforward incremental contract named Incrementer.sol. You may commence the process by generating a file for the contract:

```
vim Incrementer.sol
```

Once you have created the file, the subsequent step is to input the Solidity code into the file:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Incrementer {
uint256 public numericVal;
constructor(uint256 _startVal) {
```

```
            numericVal = _startVal;
}
function increaseVal(uint256 _inputVal) public {
numericVal = numericVal + _inputVal;
}
function resetVal() public {
            numericVal = 0;
}
}
```

When the contract is deployed, the constructor function executes and assigns the starting value to the numericVal variable that is stored on the blockchain (the default is 0). By invoking the increaseVal function, the supplied _inputVal is added to the existing value. Note that executing this function requires sending a transaction, which alters the stored data. Lastly, the resetVal function reassigns the stored value to zero.

Compile the Contract

This section will guide you through the process of building a script that leverages the Solidity compiler to produce the ABI and bytecode for the Incrementer.sol contract. Start by generating a compile.py/.js and fill it as bellow:

Python:

Import the compiler library

```
import solcx
```

If you have already installed the Solidity compiler, comment next line

```
solcx.install_solc()

Compile contract
comp_output = solcx.compile_files('Incrementer.sol')


Export contract dat
abi = comp_output['Incrementer.sol:Incrementer']['abi']
bytecode = comp_output['Incrementer.sol:Incrementer']['bin']
```

JavaScript:

```
Import packages

const fs = require('fs');
const solc = require('solc');

Load contract

const source = fs.readFileSync('Incrementer.sol', 'utf8');


Create input

const inputObject = {
   language: 'Solidity',
   sources: {
      'Incrementer.sol': {
         content: source,
      },
```

```
   },
   settings: {
      outputSelection: {
         '*': {
            '*': ['*'],
         },
      },
   },
};
```

Compile the contract

```
const compOutput = JSON.parse(solc.compile(JSON.stringify(inputObject)));
const contract = compOutput.contracts['Incrementer.sol']['Incrementer'];
```

Export contract data

```
module.exports = contract;
```

Deployment

To deploy the Incrementer.sol contract, you need to first compile the contract using a script and then create a deployment script file called deploy.py/.js. The deployment script file must complete several steps, including importing the ABI and bytecode, setting up the Web3 provider, defining the account_from with the private_key, creating a contract instance, building a constructor transaction, signing the transaction, sending it using web3.eth.send_raw_transaction function, and waiting for the transaction receipt by using web3.eth.wait_for_transaction_receipt function. It's essential to note that the private key should never be stored in a Python file. With these steps completed, you can successfully deploy the Incrementer.sol contract.

Python:

```
from compile import abi, bytecode

from web3 import Web3
web3 = Web3(Web3.HTTPProvider('RPC-API-ENDPOINT-HERE'))

account_from = {
    'private_key': 'YOUR-PRIVATE-KEY-HERE',
    'address': 'PUBLIC-ADDRESS-OF-PK-HERE',
}

print(f'Attempting to deploy from account: { account_from["address"] }')

Incrementer = web3.eth.contract(abi=abi, bytecode=bytecode)

construct_txn = Incrementer.constructor(5).buildTransaction(
    {
        'from': account_from['address'],
        'nonce': web3.eth.get_transaction_count(account_from['address']),
    }
)

tx_create = web3.eth.account.sign_transaction(construct_txn,
account_from['private_key'])

tx_hash = web3.eth.send_raw_transaction(tx_create.rawTransaction)
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)

print(f'Contract deployed at address: { tx_receipt.contractAddress }')
```

JavaScript:

```
const contractFile = require('./compile');

const Web3 = require('web3');
const web3 = new Web3('RPC-API-ENDPOINT-HERE');

const accountFrom = {
  privateKey: 'YOUR-PRIVATE-KEY-HERE',
  address: 'PUBLIC-ADDRESS-OF-PK-HERE',
};

const bytecode = contractFile.evm.bytecode.object;
const abi = contractFile.abi;


const deploy = async () => {
  console.log(`Attempting to deploy from account ${accountFrom.address}`);


  const incrementer = new web3.eth.Contract(abi);


  const incrementerTx = incrementer.deploy({
    data: bytecode,
    arguments: [5],
  });


  const createTransaction = await web3.eth.accounts.signTransaction(
    {
      data: incrementerTx.encodeABI(),
```

```
      gas: await incrementerTx.estimateGas(),
    },
    accountFrom.privateKey
  );


  const createReceipt = await
web3.eth.sendSignedTransaction(createTransaction.rawTransaction);
  console.log(`Contract deployed at address:
${createReceipt.contractAddress}`);
};
deploy();
```

Call methods

When you interact with a contract through call methods, the contract's storage remains unchanged. This means that no transaction needs to be sent. Rather, call methods simply read various storage variables of the deployed contract. To create a script for this purpose, start by creating a file named get.py/.js. Then, import the ABI, set up the Web3 provider, and define the account_from, including the private_key, which is required to sign the transaction. Note that this is only for example purposes, and you should never store your private keys in a Python/JS file. After that, create a contract instance using the web3.eth.contract function and passing in the ABI and address of the deployed contract. Finally, use the contract instance to call the number function.

Python:

```
from compile import abi


from web3 import Web3
web3 = Web3(Web3.HTTPProvider('RPC-API-ENDPOINT-HERE'))
```

```
contract_address = 'CONTRACT-ADDRESS-HERE'
print(f'Making a call to contract at address: { contract_address }')

Incrementer = web3.eth.contract(address=contract_address, abi=abi)

number = Incrementer.functions.number().call()
print(f'The current number stored is: { number } ')
```

JavaScript:

```
const { abi } = require('./compile');
const Web3 = require('web3');
const web3 = new Web3('RPC-API-ENDPOINT-HERE');

const contractAddress = 'CONTRACT-ADDRESS-HERE';

const incrementer = new web3.eth.Contract(abi, contractAddress);

const get = async () => {
  console.log(`Making a call to contract at address: ${contractAddress}`);


  const data = await incrementer.methods.number().call();
  console.log(`The current number stored is: ${data}`);
};

get();
```

Send Methods

In this section, we'll cover the send methods used to modify a contract's storage, which requires signing and sending a transaction. The purpose is to create a script to increment the incrementer. To get started, you can create a file named increment.py/.js. Begin by importing the ABI and setting up the Web3 provider. Define the account_from, including the private_key, contract_address of the deployed contract, and the value to increment by. However, it's not recommended to store private keys in a Python/JS file. Next, create a contract instance using web3.eth.contract function by passing in the ABI and address of the deployed contract. Generate the increment transaction using the contract instance, passing in the value to increment by. Use the buildTransaction function to include the transaction details, such as the from address and the nonce for the sender. Obtain the nonce by calling web3.eth.get_transaction_count function. Sign the transaction by calling the web3.eth.account.sign_transaction function and passing in the increment transaction and the private_key of the sender. Finally, send the signed transaction using web3.eth.send_raw_transaction function and wait for the transaction receipt by calling web3.eth.wait_for_transaction_receipt function.

Python:

```
from compile import abi

from web3 import Web3

web3 = Web3(Web3.HTTPProvider('RPC-API-ENDPOINT-HERE'))


account_from = {

    'private_key': 'YOUR-PRIVATE-KEY-HERE',

    'address': 'PUBLIC-ADDRESS-OF-PK-HERE',

}

contract_address = 'CONTRACT-ADDRESS-HERE'

value = 3

print(

    f'Calling the increment by { value } function in contract at address: { contract_address }'

)
```

```
Incrementer = web3.eth.contract(address=contract_address, abi=abi)


increment_tx = Incrementer.functions.increment(value).buildTransaction(
    {
        'from': account_from['address'],
        'nonce': web3.eth.get_transaction_count(account_from['address']),
    }
)

tx_create = web3.eth.account.sign_transaction(increment_tx,
account_from['private_key'])


tx_hash = web3.eth.send_raw_transaction(tx_create.rawTransaction)
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
print(f'Tx successful with hash: { tx_receipt.transactionHash.hex() }')
```

JavaScript:

```
const { abi } = require('./compile');


const Web3 = require('web3');
const web3 = new Web3('RPC-API-ENDPOINT-HERE');


const accountFrom = {
  privateKey: 'YOUR-PRIVATE-KEY-HERE',
};
const contractAddress = 'CONTRACT-ADDRESS-HERE';
const _value = 3;

const incrementer = new web3.eth.Contract(abi, contractAddress);
```

```
const incrementTx = incrementer.methods.increment(_value);


const increment = async () => {
  console.log(
    `Calling the increment by ${_value} function in contract at address: 
${contractAddress}`
  );


  const createTransaction = await web3.eth.accounts.signTransaction(
    {
      to: contractAddress,
      data: incrementTx.encodeABI(),
      gas: await incrementTx.estimateGas(),
    },
    accountFrom.privateKey
  );

  const createReceipt = await 
web3.eth.sendSignedTransaction(createTransaction.rawTransaction);
  console.log(`Tx successful with hash: ${createReceipt.transactionHash}`);
};



increment();
```

Differences between Web3.js and Ethers.js

As the world of blockchain and decentralized applications (dApps) continues to evolve, JavaScript libraries such as web3.js and ethers.js have emerged as powerful tools for web3 developers. These libraries provide developers with the necessary functions and interfaces to interact with blockchain networks, including Ethereum. However, it is important to understand the advantages and setbacks of these libraries in order to make an informed choice based on project requirements. In this article, we will compare web3.js and ethers.js, highlighting their respective advantages and setbacks.

Advantages of web3.js:
Web3.js, developed by the Ethereum Foundation, has the distinction of being one of the earliest JavaScript libraries for web3 development. This lends credibility to its effectiveness in design, as it benefits from a large base of contributors and regular updates. Additionally, web3.js has gained popularity due to its widespread adoption in the blockchain community and the availability of a large community of experienced developers.

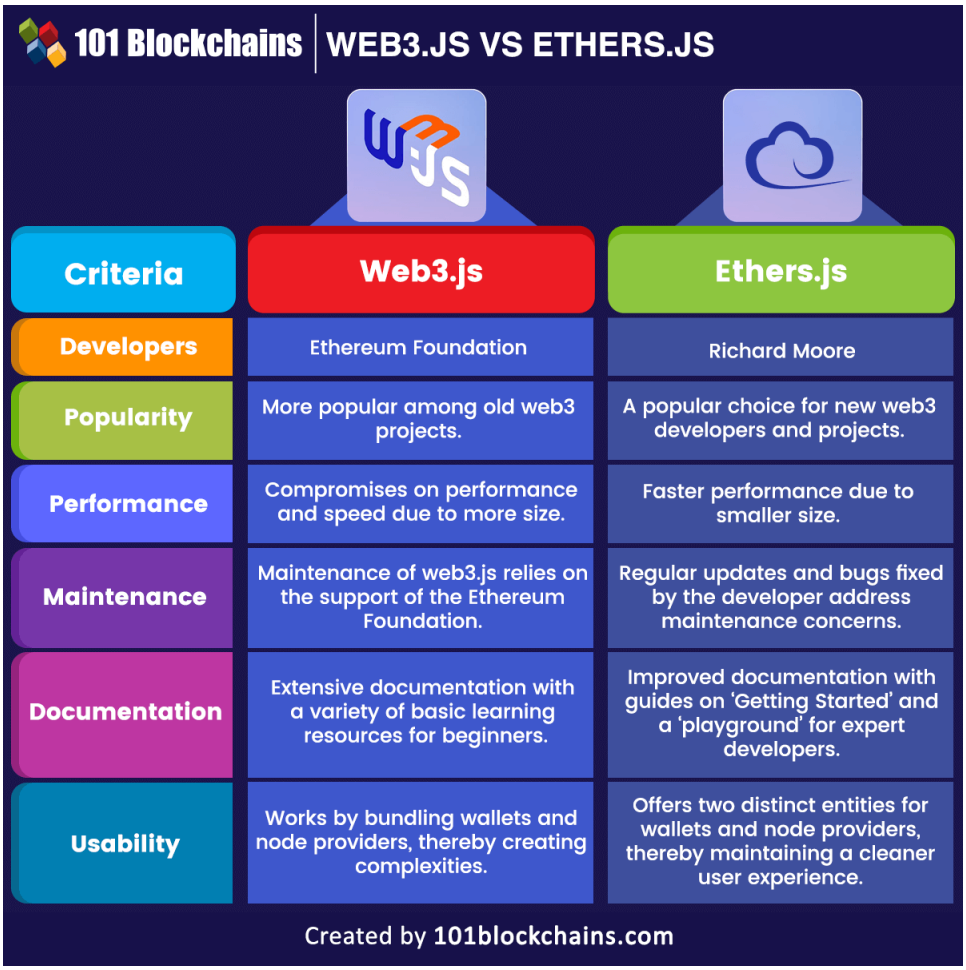
Setbacks of web3.js:
Despite its advantages, web3.js also has some setbacks. It may not be suitable for all new projects, as it may not be the best fit for different requirements or use cases. Additionally, web3.js relies on a license with restrictive limitations, such as the need to release modifications to the public, which may not be suitable for some projects. Another potential drawback is its larger size compared to other web3 libraries, which may impact the performance of a web3 site or app.

Advantages of ethers.js:
Ethers.js, on the other hand, offers several advantages for web3 development. It has a broader license that allows for free usage and modifications, with the requirement to release source code with implemented modifications. Ethers.js is also a lightweight library (only 77kb vs 4.5MB) that supports ENS domain names and has proven support for a large number of test cases, providing developers with flexibility and reliability. Ethers.js separates the roles of a "wallet" for key management and a "provider" for connecting to the Ethereum network. This allows developers to have more flexibility in managing keys and transactions, such as using a hardware device for wallet functions and Unifra as the provider.

Setbacks of ethers.js:

While ethers.js presents noticeable improvements over the setbacks of web3.js, it also has some potential drawbacks. Being a comparatively new library, developers may encounter difficulties in using ethers.js for older companies or projects that may have been built using other libraries. However, no specific setbacks were mentioned in the provided text.

web3.js and ethers.js are two popular JavaScript libraries for web3 development, each with its own set of advantages and setbacks. Ultimately, the choice between web3.js and ethers.js would depend on the specific needs of the project and the preferences of the developers involved.



Additional Resources:

https://github.com/adrianmcli/web3-vs-ethers