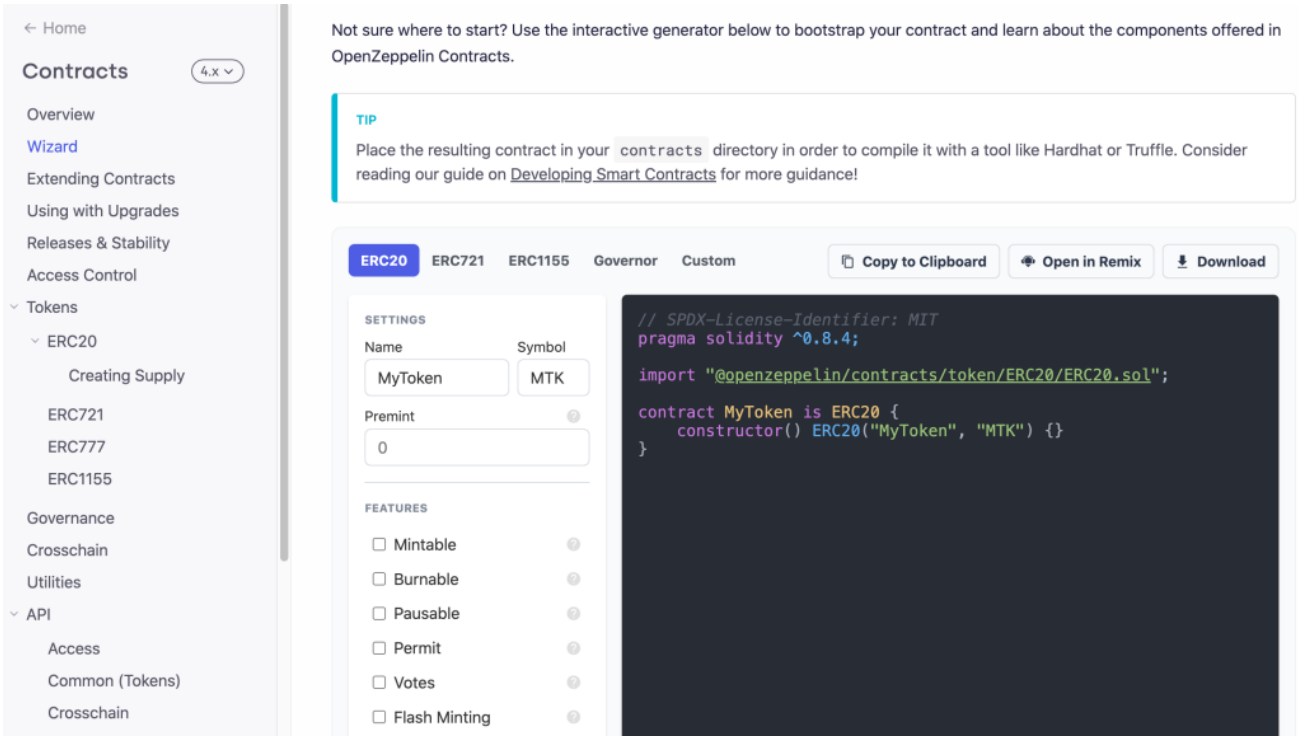
Deploying and executing Web3 Smart Contracts in Polygon using Node.js and Web3.js



The ability to have premissionless transistion functions through smart contracts makes blockchain special. These are two most commonly used smart contracts:

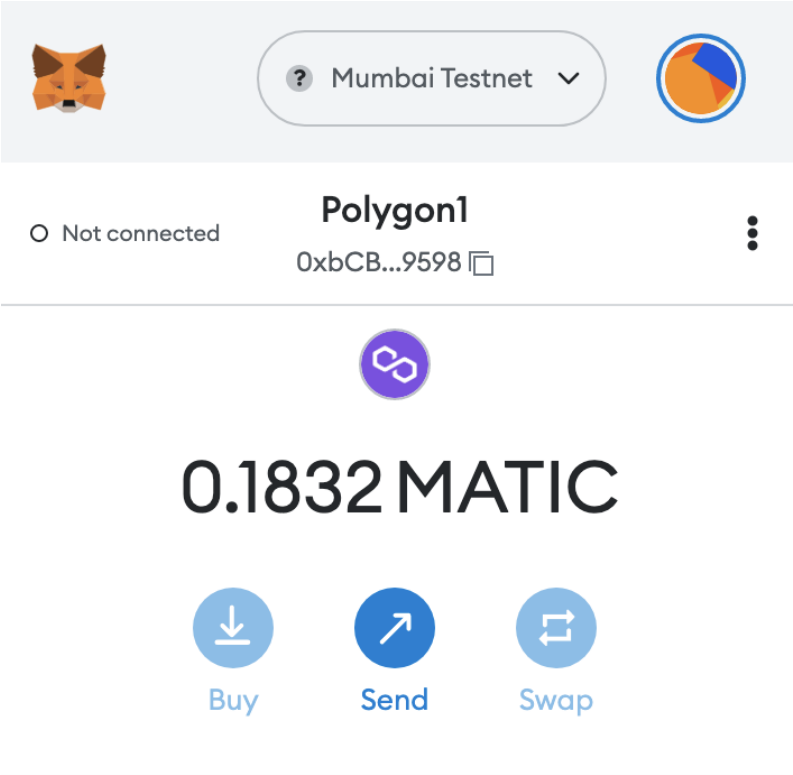
- Tokens (ERC20) – Used mainly as cryptocurrency.
- Non-fungible Tokens (ERC721) – Everyone knows NFT.

Our goal is to deploy the two smart contracts into a blockchain and interact with it externally using Node.js with Web3.js libraries in four (4) steps. For our blockchain we will be using Polygon Mumbai Testnet: https://polygonscan.com/

STEP 1: Wallet

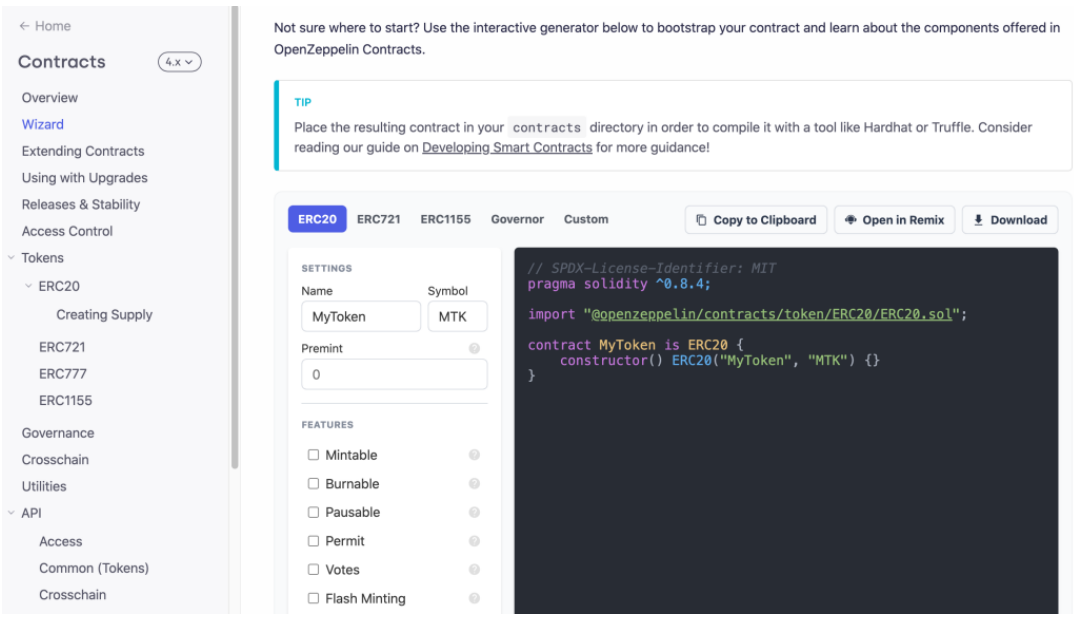
So the first step is to setup our wallet and connect it to a blockchain. The wallet that we will be using is **Metamask** and the blockchain is **Polygon**. Follow the documentation on how to setup **Metamask** and **Polygon** here.

We need to create two addresses in our wallet. For the first address we need to have MATIC (Polygon's native token) balance. We need a balance so that we can upload our smart contracts, since blockchains requires gas fees. For Polygon Testnet (Mumbai), we can ask for free MATIC using this link: https://faucet.polygon.technology/



STEP 2: Smart Contracts

Create the two smart contracts using OpenZeppelin. There is already a template for ERC20 (Token) and ERC721 (NFT). Just pick the methods, e.g., transfer, mint, burn, etc., that we need to have in our token and **OpenZeppelin** will write the **Solidity** code for us.
Once done we can compile and deploy our smart contract **Solidity** code using **Remix**. Make sure that we deploy our contracts to **Polygon Testnet (Mumbai)** using the first address.
Here is the detailed step on how to deploy smart contracts using **OpenZeppelin** and **Remix**.



Once we successfully deployed our smart contracts we can then view our deployed contract using Polygon Scan. Just search for the first address. Take note of the contract addresses in the transactions, we will use it in our Node.js project.

STEP 3: Node.js

Now that our wallet, blockchain and smart contracts are fully set up. It's time to write the code, the first step is to make a Node.js project and install Web3.js.

```
$ mkdir web3

$ cd web3

$ npm install web3

$ npm install dotenv --save
```

It is expected that we already installed Node.js and NPM, if not we follow these steps.

Make an environment file (.env) and put all the addresses, private keys and urls needed for the code.

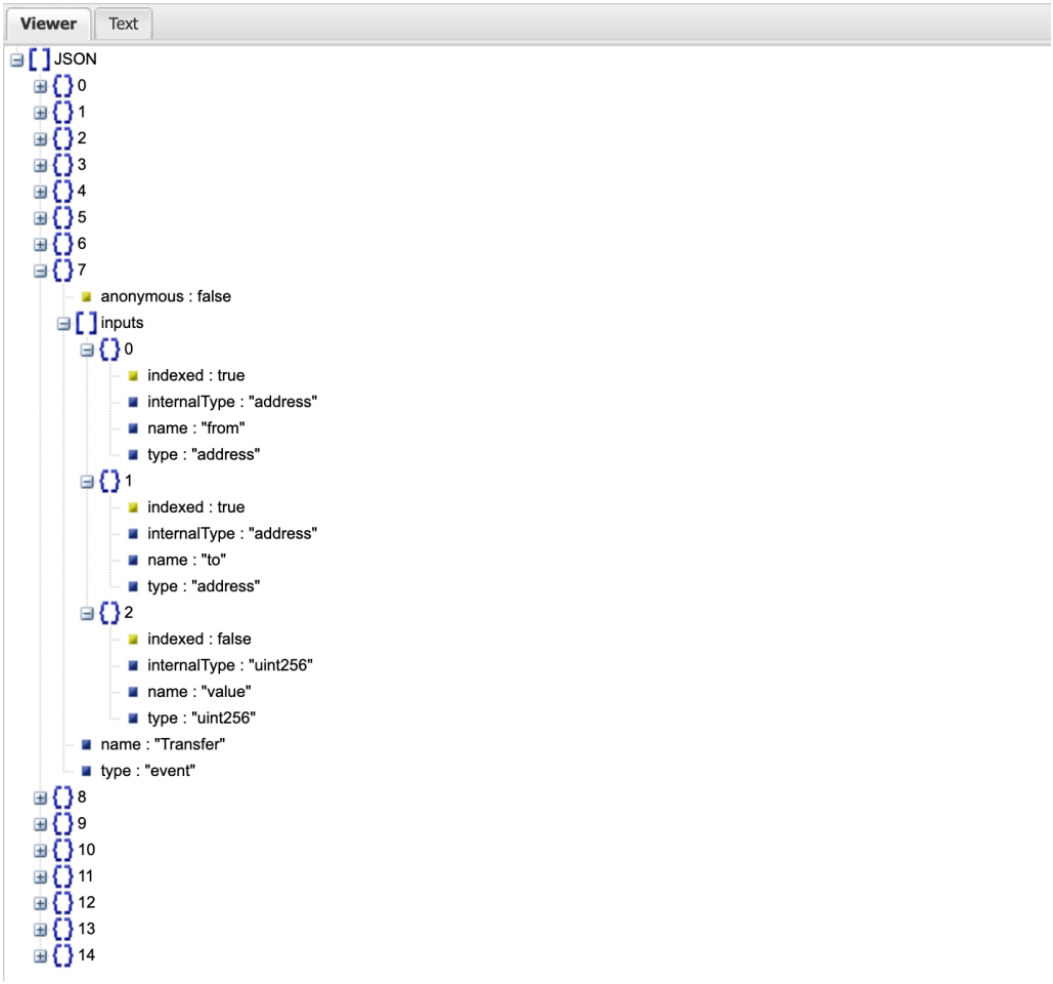
```
TESTNET=https://rpc-mumbai.maticvigil.com/

ADDRESS1=0xbCBA4xxxxxxxxxxxx2bxxxxxx

ADDRESS1_PRIVATE_KEY=8c5305axxxxxxx8e80xxx03xxxxxx

ADDRESS2=0x52Ec00xxxxxxxxxxxxx2aB162Fc8xxxxxxxxx

TOKEN_CONTRACT=0xadxxxxxxxx37818a9xxxxxxxxx

NFT_CONTRACT=0x542e8xxxxxxxxxx1842344xxxxxxxxx
```

STEP 4: Code

It's time to create the code that will interact with our blockchain smart contracts.

To interact with the blockchain smart contracts we need to have the Application Binary Inteface (ABI) of the said contracts. We can get ABI file upon smart contract deployment in **Remix**, it contains all functions, methods and properties of the deployed contract.

To view the ABI file, just uploaded it to any free JSON viewer websites.



Now that we know what functions, methods and properties to call in our smart contracts, we can proceed to writing the code on how to get the **`totalSupply`** of our Token and NFT.

```
require('dotenv').config();

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

async function main() {
    const testnet = process.env.TESTNET;
    const web3 = new Web3(new
Web3.providers.HttpProvider(testnet));
    const abi =
JSON.parse(fs.readFileSync("./token_abi.json"));
    let contract = new web3.eth.Contract(abi,
process.env.TOKEN_CONTRACT);
```

```
    let supply = await contract.methods.totalSupply().call();

    console.log(supply);
}

main().catch(console.error).finally(() => process.exit());

require('dotenv').config();

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

async function main() {
    const testnet = process.env.TESTNET;
    const walletAddress = process.env.ADDRESS1;
    const web3 = new Web3(new
Web3.providers.HttpProvider(testnet));
    const abi = JSON.parse(fs.readFileSync("./nft_abi.json"));
    let contract = new web3.eth.Contract(abi,
process.env.NFT_CONTRACT);

    let supply = await contract.methods.totalSupply().call();

    console.log(supply);
}

main().catch(console.error).finally(() => process.exit());
```

The code is so easy and self explanatory. We can then wrap this code to a REST API Node server, so that we can consume it on any App or DApp.

The **`totalSupply`** function is a mere query call, hence no encoding to hash values.
Let me give you another example that needs encoding, the function **`mint`** in our Token smart contract.

```
require('dotenv').config();

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

async function main() {
    const testnet = process.env.TESTNET;
    const walletAddress = process.env.ADDRESS1;
    const walletAddressPrivateKey =
process.env.ADDRESS1_PRIVATE_KEY;
    const tokenContract = process.env.TOKEN_CONTRACT;

    const web3 = new Web3(new
Web3.providers.HttpProvider(testnet));
    const abi =
JSON.parse(fs.readFileSync("./token_abi.json"));
    let contract = new web3.eth.Contract(abi, tokenContract);
    const nonce = await
web3.eth.getTransactionCount(walletAddress, "latest");
    const transaction = {
        from: walletAddress,
        to: tokenContract,
        nonce: nonce,
        gas: 500000,
        data: contract.methods.mint(100000).encodeABI(),
    };
    const signPromise = await
web3.eth.accounts.signTransaction(
        transaction,
        walletAddressPrivateKey
    );
    const signedTransaction = await
```

```
web3.eth.sendSignedTransaction(
        signPromise["rawTransaction"]
    );
    const hash = signedTransaction["transactionHash"];
    console.log(hash);
}

main().catch(console.error).finally(() => process.exit());
```

The mint function writes data to the blockchain, hence must be encoded.

The transaction is encoded, signed, and sent to the blockchain network. The blockchain network in response will return the hash code of the transaction. Same process is needed for all new transactions, e.g., transfer, burn, etc., in the blockchain. In the **`safeTransferFrom`** code for NFT below, we will need the second address to transfer the NFT from the first address.

```
require('dotenv').config();

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

async function main() {
    const testnet = process.env.TESTNET;
    const walletAddress1 = process.env.ADDRESS1;
    const walletAddress1PrivateKey =
process.env.ADDRESS1_PRIVATE_KEY;
    const walletAddress2 = process.env.ADDRESS2;
    const nftContract = process.env.NFT_CONTRACT;

    const web3 = new Web3(new
Web3.providers.HttpProvider(testnet));
    const abi =
JSON.parse(fs.readFileSync("./bnft_abi.json"));
    let contract = new web3.eth.Contract(abi, nftContract);
    const nonce = await
```

```
web3.eth.getTransactionCount(walletAddress1, "latest");
    const transaction = {
        from: walletAddress1,
        to: nftContract,
        nonce: nonce,
        gas: 500000,
        data:
contract.methods.safeTransferFrom(walletAddress1,
walletAddress2, 1).encodeABI(),
    };
    const signPromise = await
web3.eth.accounts.signTransaction(
        transaction,
        walletAddress1PrivateKey
    );
    const signedTransaction = await
web3.eth.sendSignedTransaction(
        signPromise["rawTransaction"]
    );
    const hash = signedTransaction["transactionHash"];
    console.log(hash);
}

main().catch(console.error).finally(() => process.exit());
```

That's it. Have fun coding. Please don't forget to donate if your find this blog helpful. And if you want the complete code, just click the Github link.