FUNGIBLE

STUDIOS

Fungible Studios

Blockchain Development Guide

ERC20, ERC721, IPFS, Polygonscan

Document details

Document Title	Blockchain_Development_Guide_v0.1
File name	Blockchain_Development_Guide_v0.1.docx

Document version

Version	**Release date**	**Authors**	**Summary of changes**
0.1	1/05/2022	Nicholas Chai	Full write up first edition

Document circulation

Name / Position	**Organisation/Title**
Nicholas Chai	COO Fungible Studios
Aaron Cross	CEO Fungible Studios
Jimmy Cheung	CCO Fungible Studios
Dana Ahmadi	CTO Fungible Studios
Steve Win	CFO Fungible Studios

Contents

1 Executive Summary

This document outlines and describes step by step how to create fungible and non-fungible tokens on to a live blockchain network in the quickest and easiest way possible without compromising on customizability and programmatic methodology.

Technologies involved include:

- **NodeJS** – An asynchronous event-driven javascript runtime environment for back end, serverside processing.
- **Hardhat** – An Ethereum development environment for building, deploying, running and testing Ethereum projects.
- **Openzeppelin Contracts** – A library of prewritten contracts that can be imported into projects to very quickly create ERC20 and ERC721 compliant smart contracts.
- **Interplanetary File Storage (IPFS)** – A decentralized file storage service.
- **Polygonscan –** A polygon network chain explorer and hosting service for validating deployed contracts. Verification on polygonscan is important for legitimizing smart contracts.
- **OpenSea –** A well known NFT marketplace.

ERC20 is a smart contract standard which defines a list of common rules and required functions for a fungible token smart contract. Any token currency must be compliant with the ERC-20 standard for it to be accepted by popular exchanges and services such as uniswap so compliance with the ERC20 standard is extremely important. Similarly, the ERC721 standard is a list of common rules and required functions for a non-fungible token (NFT) smart contract. Any NFT that has plans on being listed properly on OpenSea a should be ERC721 compliant. The detailed specifications of these standards can be found within the Ethereum developer page.

To quickly create ERC20 and ERC721 compliant smart contracts, developers can take advantage of the openzeppelin contracts. These are prewritten contracts of boilerplate solidity code which can be imported into a project to make any smart contract compliant with the required standard.

NodeJS and Hardhat are development tools which will allow developers to easily download required software libraries/packages and to create local development environments including the creation of local blockchains to deploy, run and test scripts locally first before deploying to a live blockchain network.

This document has been written to be used as a reference document and to explain the full and completely written code hosted on a github project at the below location:

Nixxs/EthereumSmartContractExamples: Simple example projects for creating smart contracts on the etherium blockchain and how to interact with them. (github.com)

Upon completion of this document, developers will have:

- Written and deployed a new token currency creation contract on the polygon network and minted a number of fungible tokens ready to be traded
- Written and executed a script for creating metadata records on the IPFS network
- Written and deployed a new NFT contract on the polygon network and executed it to create a new NFT token with reference to the IPFS metadata for NFT artwork and attributes

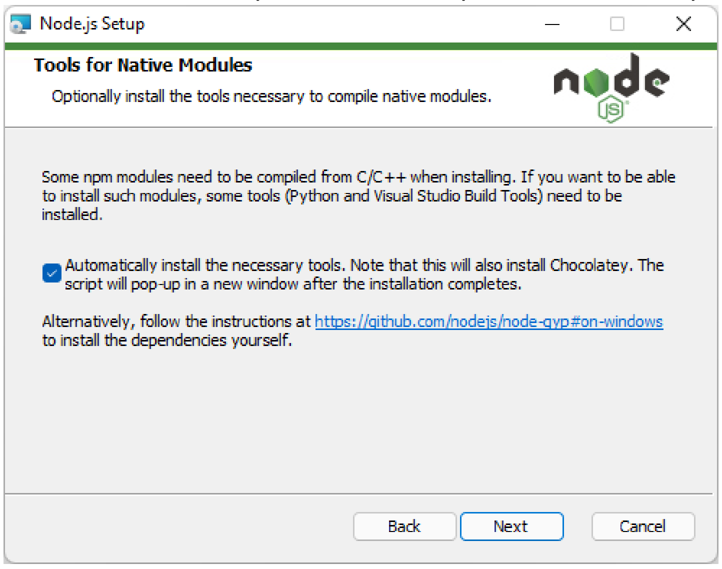
Finally, the appendix of this document outlines the general costs of transactions and gas required to perform all of these tasks. Please refer to the gas transactions chart in the appendix for details.

2 ERC20 Tokens

2.1 Setting up the local dev environment

Before we can get started we'll need to make sure we have all the prerequisite software installed. Follow the steps below to get your local dev environment up and running ready for blockchain development.

1. Install NodeJS it comes with NPM which we will use to install all the required libraries
    a. Node.js (nodejs.org)
    b. Tick the automatically install necessary tool so it has everything it needs to compile things:



Node.js Setup

Tools for Native Modules

Optionally install the tools necessary to compile native modules.

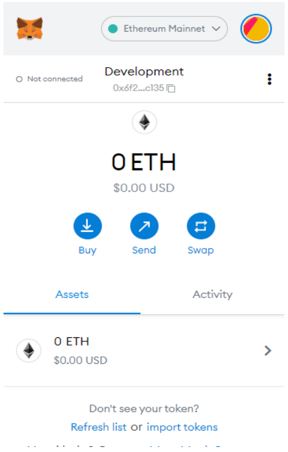
Some npm modules need to be compiled from C/C++ when installing. If you want to be able to install such modules, some tools (Python and Visual Studio Build Tools) need to be installed.

[✓] Automatically install the necessary tools. Note that this will also install Chocolatey. The script will pop-up in a new window after the installation completes.

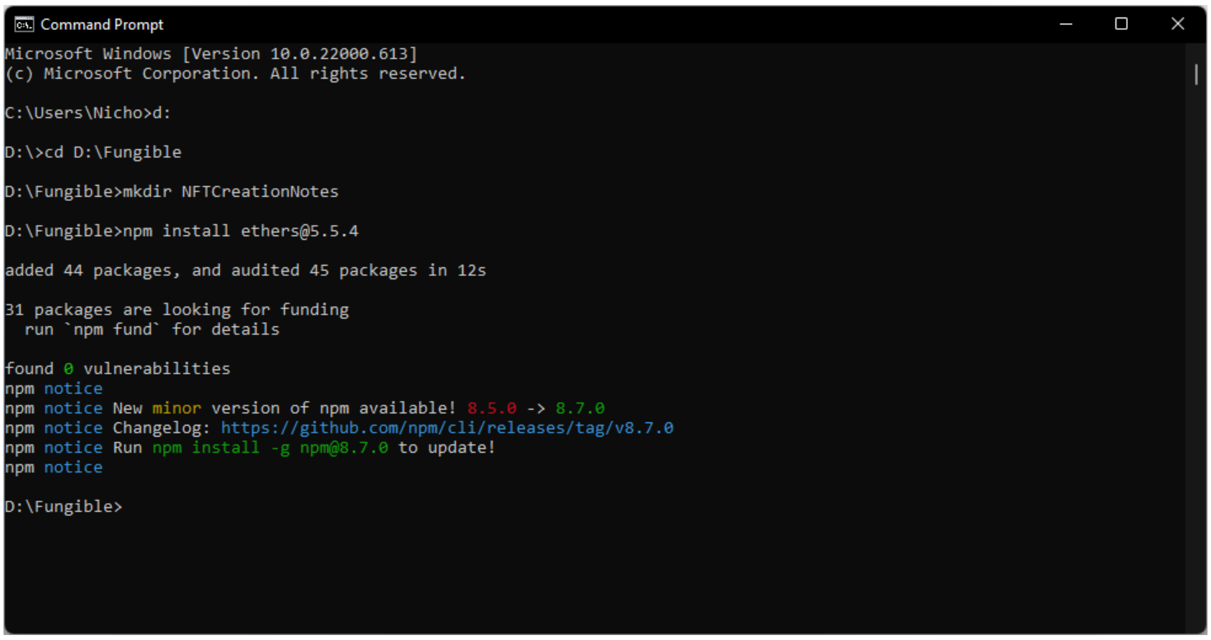
Alternatively, follow the instructions at https://github.com/nodejs/node-gyp#on-windows to install the dependencies yourself.

2. Install hardhat – hardhat is a set of tools that will allow you to create a local blockchain so that you can test deployment of contracts in a dev environment without having to deploy out to a real blockchain that would cost you gas.
    a. Now that you have nodejs/npm installed you can just open your console and run this command to install it:
        i. Npm install –save-dev hardhat@2.8.4
        ii. Installing this version of hardhat specifically because that's the one I setup my demo project with

3. Next install metmask extension onto your browser, you need this so that you can have a wallet for hardhat to use for deployment.
    a. The crypto wallet & gateway to Web3 blockchain apps | MetaMask
    b. Once installed setup an account to create your wallet, you can just call it development or whatever:



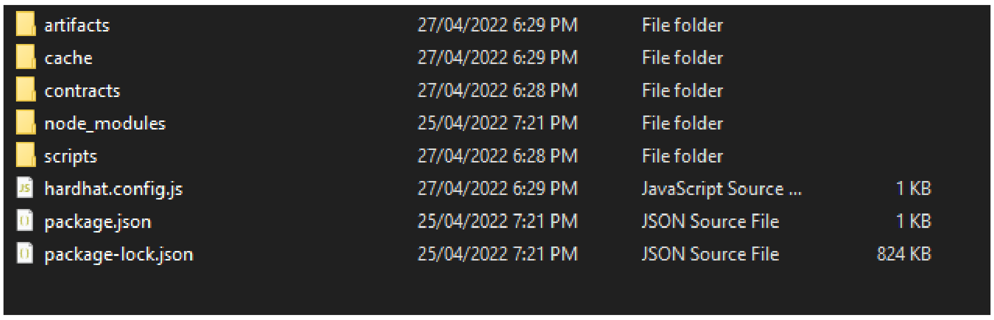
4. Now we are ready to make our project folder so create the folder, change directory into it then install all the required packages using NPM commands below:
   a. npm install ethers@5.5.4
   b. npm install hardhat@2.8.4
   c. npm install @nomiclabs/hardhat-ethers@2.0.5
   d. npm install @nomiclabs/hardhat-waffle@2.0.2
   e. npm install @openzeppelin/contracts@4.5.0



```
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Nicho>d:

D:\>cd D:\Fungible

D:\Fungible>mkdir NFTCreationNotes

D:\Fungible>npm install ethers@5.5.4

added 44 packages, and audited 45 packages in 12s

31 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
npm notice
npm notice New minor version of npm available! 8.5.0 -> 8.7.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v8.7.0
npm notice Run npm install -g npm@8.7.0 to update!
npm notice

D:\Fungible>
```

   f. Ethers is a library that will allow your front-end application to interact with the blockchain and openzeppelin contracts are a set of standardized contract code we can import into our smart contract to make them compliant with ERC-20 standards without having to write all the functionality ourselves.

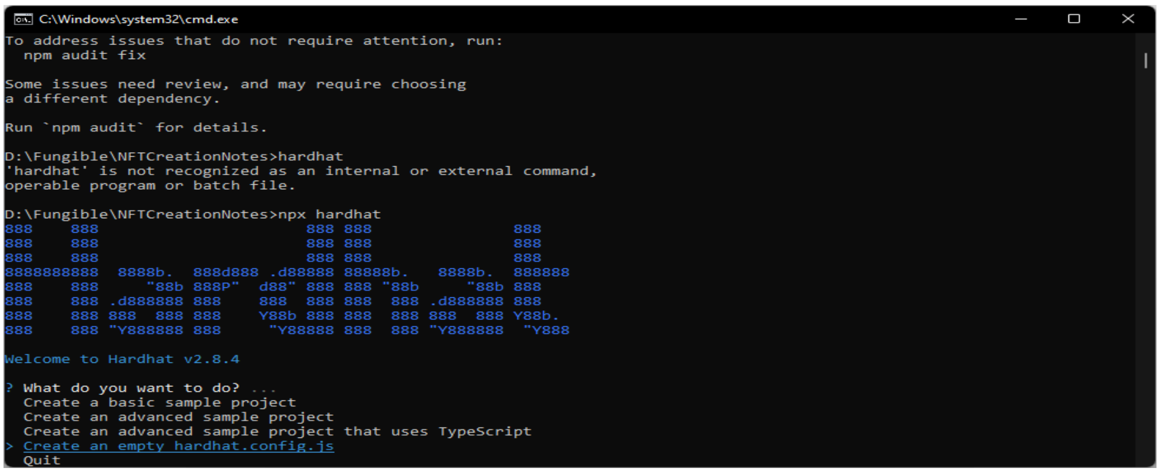
5. Now all the dependencies are installed, we will next setup some folders for hardhat to use. Set up your project folders like the below:



artifacts	27/04/2022 6:29 PM	File folder	
cache	27/04/2022 6:29 PM	File folder	
contracts	27/04/2022 6:28 PM	File folder	
node_modules	25/04/2022 7:21 PM	File folder	
scripts	27/04/2022 6:28 PM	File folder	
hardhat.config.js	27/04/2022 6:29 PM	JavaScript Source ...	1 KB
package.json	25/04/2022 7:21 PM	JSON Source File	1 KB
package-lock.json	25/04/2022 7:21 PM	JSON Source File	824 KB

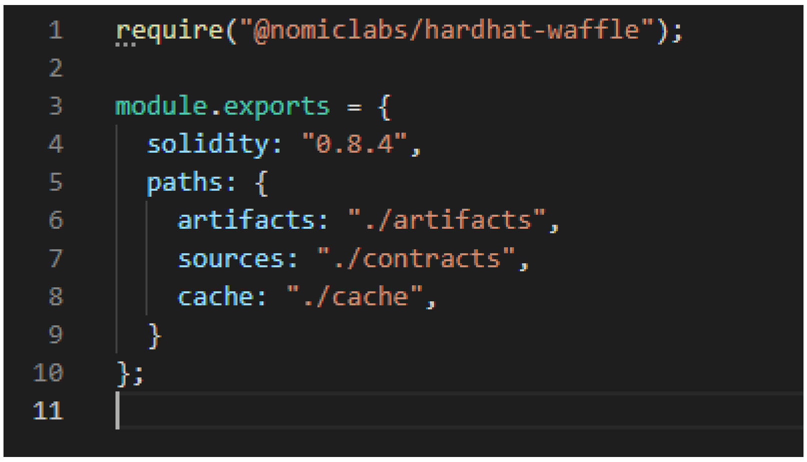
6. Now we will run hardhat in our project directory and tell it to setup a hardhat.config.js file so that we can setup hardhat to run properly in our environment. From the project root directory run:

   npx hardhat and select “Create and empty hardhat.config.js”:



```
C:\Windows\system32\cmd.exe
To address issues that do not require attention, run:
  npm audit fix

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.

D:\Fungible\NFTCreationNotes>hardhat
'hardhat' is not recognized as an internal or external command,
operable program or batch file.

D:\Fungible\NFTCreationNotes>npx hardhat
888    888                      888 888               888
888    888                      888 888               888
888    888                      888 888               888
8888888888  8888b.  888d888 .d88888 88888b.   8888b.  888888
888    888     "88b 888P"  d88" 888 888 "88b     "88b 888
888    888 .d888888 888    888  888 888  888 .d888888 888
888    888 888  888 888    Y88b 888 888  888 888  888 Y88b.
888    888 "Y888888 888     "Y88888 888  888 "Y888888  "Y888

Welcome to Hardhat v2.8.4

? What do you want to do? ...
  Create a basic sample project
  Create an advanced sample project
  Create an advanced sample project that uses TypeScript
> Create an empty hardhat.config.js
  Quit
```

7. Edit your new hardhat.config.js file so that it looks like the below, basically we are just telling hardhat what version of solidity we are running and what folders it should use:



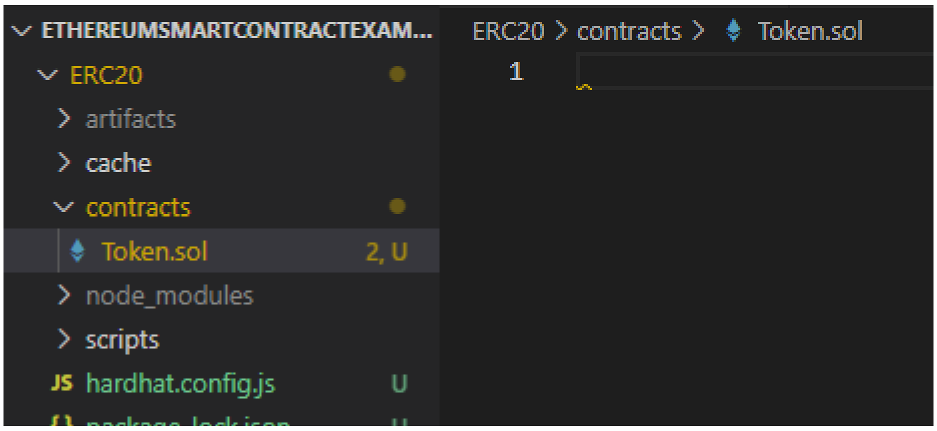
```javascript
require("@nomiclabs/hardhat-waffle");

module.exports = {
  solidity: "0.8.4",
  paths: {
    artifacts: "./artifacts",
    sources: "./contracts",
    cache: "./cache",
  }
};
```

   Also make sure you include the require statement seen above otherwise deploying later will return an error as hardhat won’t have access to ethers. A library that we need for deploying.

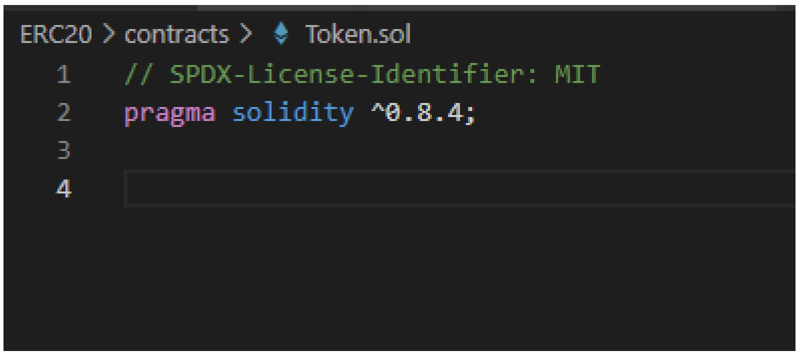
2.2 Writing the ERC-20 Smart Contract

Now that everything is ready to go we will start by creating the smart contract code. The contract for the token is written locally and then deployed onto the blockchain where the EVM (Ethereum virtual machine) will read the contract code and execute it on deployment. You can think of the smart contract as a piece of code that is always running on the EVM machine in perpetuity.

1. Hardhat it ready to go, next we will write our token contract into the contracts folder we created. Contracts are written in the “solidity” language and have a .sol extension so create a file inside the contracts folder called “Token.sol” or whatever you want.

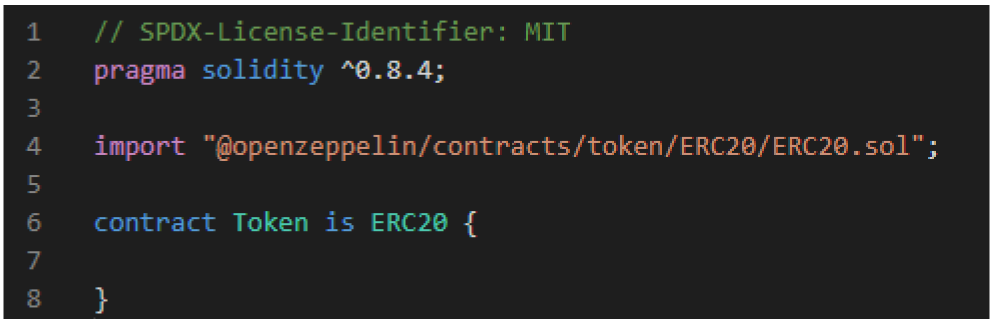


2. First thing is to tell the EVM which version of solidity this is written in and what license:



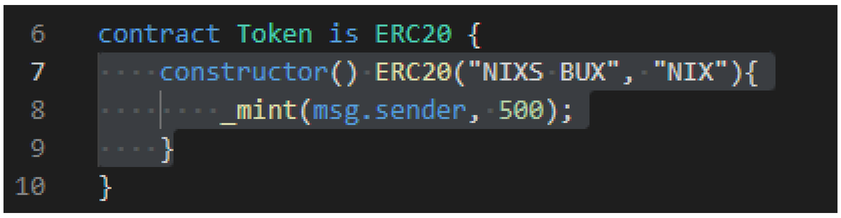
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

```

3. All solidty code lives inside a contract so define the contract and then we'll also use the import statement to import the open zeppelin contract for ERC20 tokens like this:|



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract Token is ERC20 {

}
```

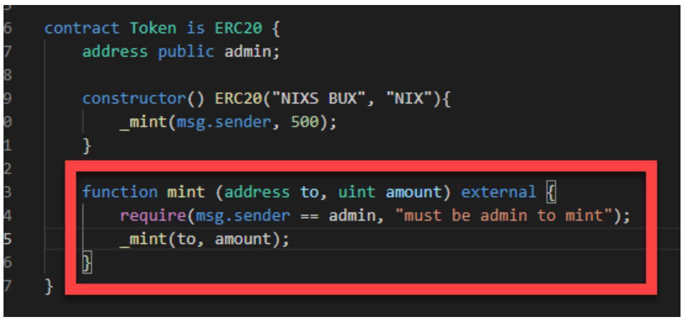
4. Next we will call the built in ERC20 constructor which requires a token name an token sy mbol as arguments. In side the constructor we will call the _mint() function which requires and address and the number of tokens we want to mint.



```
contract Token is ERC20 {
    constructor() ERC20("NIXS BUX", "NIX"){
        _mint(msg.sender, 500);
    }
}
```

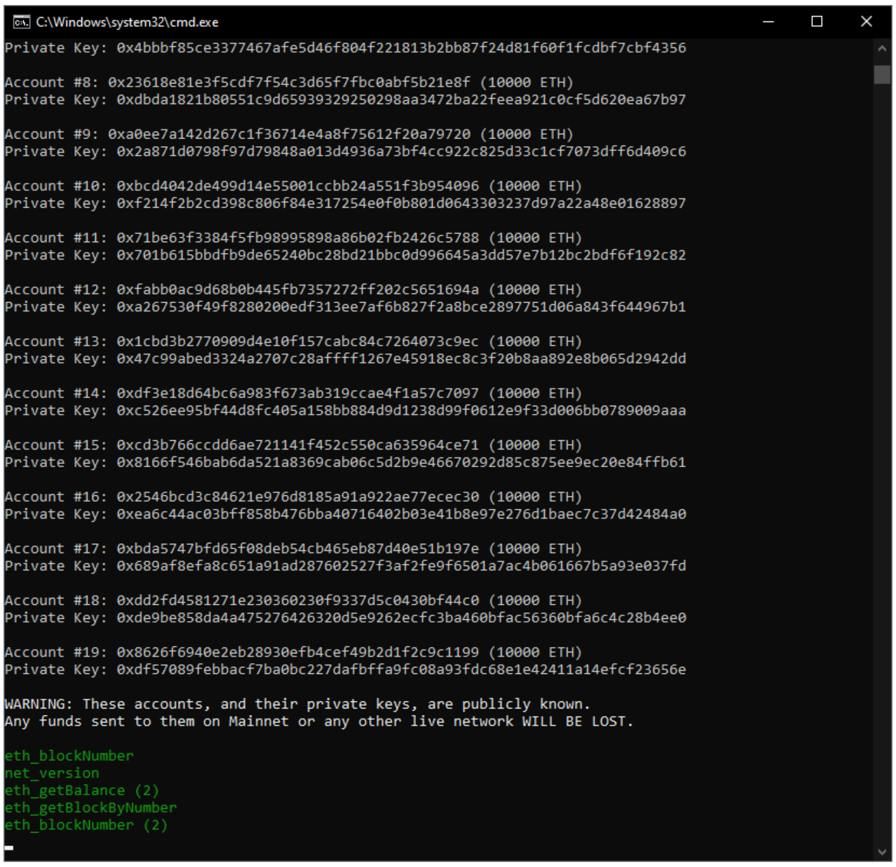
And that's it, once we deploy this 500 NIX will be created on the block chain and assigned to the deployer of the contract.

5. If we wanted to do more with this contract, we could also add in a mint function ourselves so that more tokens can be created after deployment. We will also want this function to only be accessible by certain accounts though like say an admin account so in our constructor we will define the admin account and store that away.



```
contract Token is ERC20 {
    address public admin;

    constructor() ERC20("NIXS BUX", "NIX"){
        _mint(msg.sender, 500);
    }

    function mint (address to, uint amount) external {
        require(msg.sender == admin, "must be admin to mint");
        _mint(to, amount);
    }
}
```

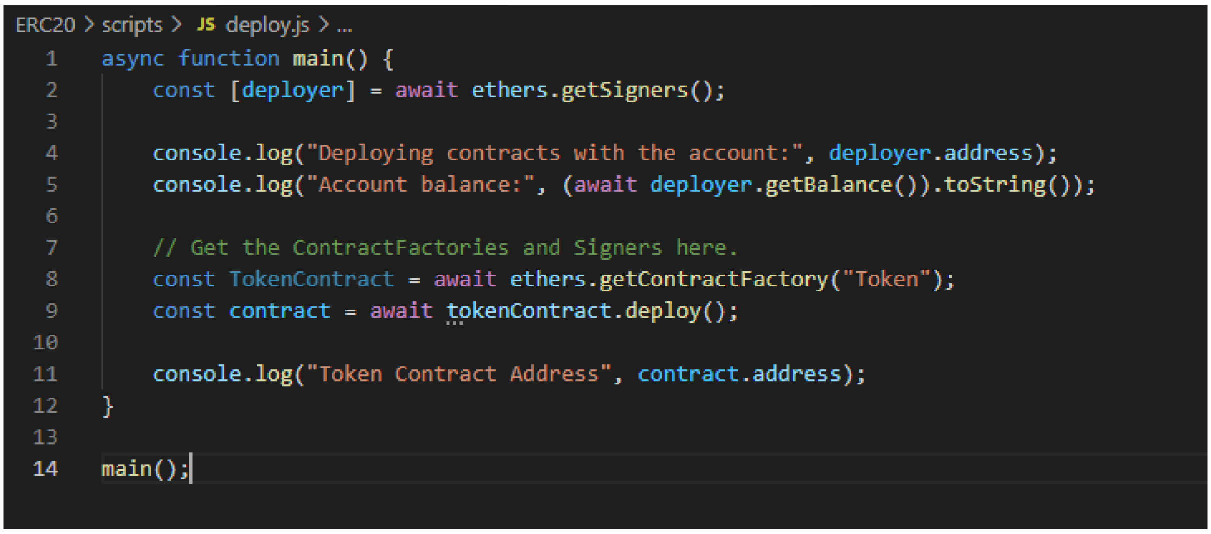
6. Now we need a blockchain to deploy this contract to, we will first make a local blockchain using hardhat to make sure our contract works. In the console type “npx hardhat node” to start a local blockchain:



C:\Windows\system32\cmd.exe

```
Private Key: 0x4bbbf85ce3377467afe5d46f804f221813b2bb87f24d81f60f1fcdbf7cbf4356

Account #8: 0x23618e81e3f5cdf7f54c3d65f7fbc0abf5b21e8f (10000 ETH)
Private Key: 0xdbda1821b80551c9d65939329250298aa3472ba22feea921c0cf5d620ea67b97

Account #9: 0xa0ee7a142d267c1f36714e4a8f75612f20a79720 (10000 ETH)
Private Key: 0x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6

Account #10: 0xbcd4042de499d14e55001ccbb24a551f3b954096 (10000 ETH)
Private Key: 0xf214f2b2cd398c806f84e317254e0f0b801d0643303237d97a22a48e01628897

Account #11: 0x71be63f3384f5fb98995898a86b02fb2426c5788 (10000 ETH)
Private Key: 0x701b615bbdfb9de65240bc28bd21bbc0d996645a3dd57e7b12bc2bdf6f192c82

Account #12: 0xfabb0ac9d68b0b445fb7357272ff202c5651694a (10000 ETH)
Private Key: 0xa267530f49f8280200edf313ee7af6b827f2a8bce2897751d06a843f644967b1

Account #13: 0x1cbd3b2770909d4e10f157cabc84c7264073c9ec (10000 ETH)
Private Key: 0x47c99abed3324a2707c28affff1267e45918ec8c3f20b8aa892e8b065d2942dd

Account #14: 0xdf3e18d64bc6a983f673ab319ccae4f1a57c7097 (10000 ETH)
Private Key: 0xc526ee95bf44d8fc405a158bb884d9d1238d99f0612e9f33d006bb0789009aaa

Account #15: 0xcd3b766ccdd6ae721141f452c550ca635964ce71 (10000 ETH)
Private Key: 0x8166f546bab6da521a8369cab06c5d2b9e46670292d85c875ee9ec20e84ffb61

Account #16: 0x2546bcd3c84621e976d8185a91a922ae77ecec30 (10000 ETH)
Private Key: 0xea6c44ac03bff858b476bba40716402b03e41b8e97e276d1baec7c37d42484a0

Account #17: 0xbda5747bfd65f08deb54cb465eb87d40e51b197e (10000 ETH)
Private Key: 0x689af8efa8c651a91ad287602527f3af2fe9f6501a7ac4b061667b5a93e037fd

Account #18: 0xdd2fd4581271e230360230f9337d5c0430bf44c0 (10000 ETH)
Private Key: 0xde9be858da4a475276426320d5e9262ecfc3ba460bfac56360bfa6c4c28b4ee0

Account #19: 0x8626f6940e2eb28930efb4cef49b2d1f2c9c1199 (10000 ETH)
Private Key: 0xdf57089febbacf7ba0bc227dafbffa9fc08a93fdc68e1e42411a14efcf23656e

WARNING: These accounts, and their private keys, are publicly known.
Any funds sent to them on Mainnet or any other live network WILL BE LOST.

eth_blockNumber
net_version
eth_getBalance (2)
eth_getBlockByNumber
eth_blockNumber (2)
```

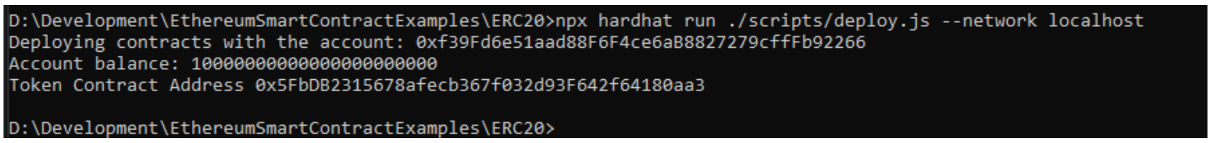
7. Now we will write a new deploy.js script that we will use to deploy the contract:



```
ERC20 > scripts > JS deploy.js > ...
async function main() {
    const [deployer] = await ethers.getSigners();

    console.log("Deploying contracts with the account:", deployer.address);
    console.log("Account balance:", (await deployer.getBalance()).toString());

    // Get the ContractFactories and Signers here.
    const TokenContract = await ethers.getContractFactory("Token");
    const contract = await tokenContract.deploy();

    console.log("Token Contract Address", contract.address);
}

main();
```

8. Now we will run the deploy.js script in npm to deploy out contract to the local blockchain with the below command:

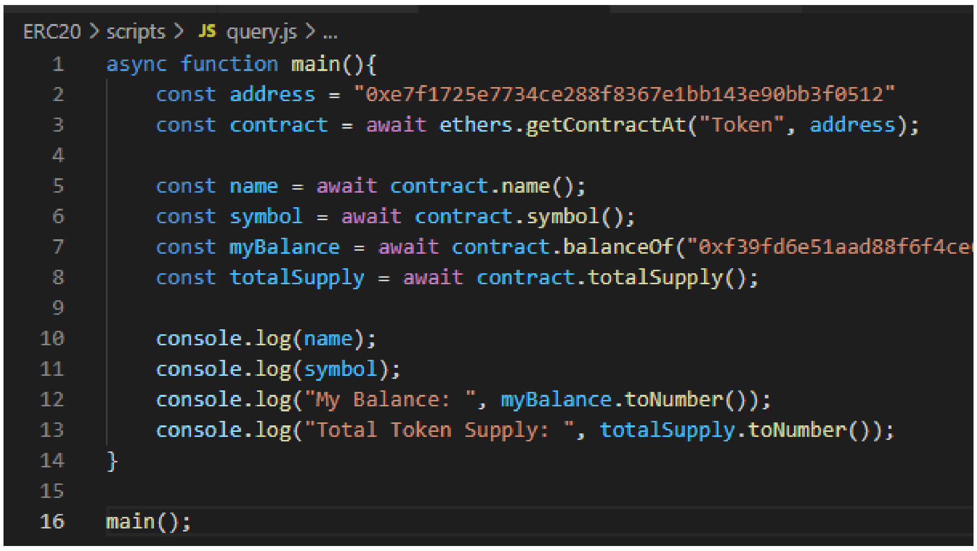
   npx hardhat run ./scripts/deploy.js –network localhost



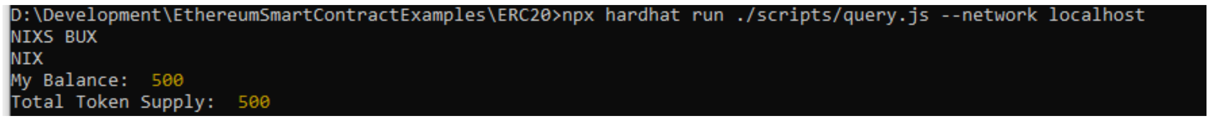
```
D:\Development\EthereumSmartContractExamples\ERC20>npx hardhat run ./scripts/deploy.js --network localhost
Deploying contracts with the account: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
Account balance: 10000000000000000000000
Token Contract Address 0x5FbDB2315678afecb367f032d93F642f64180aa3

D:\Development\EthereumSmartContractExamples\ERC20>
```

   The contract has now been deployed to the local block chain.

9. To test the contract, we will write another js script that we will use to query the contract for information like ask it how many tokens we are currently holding or to transfer the tokens to another account. Write the below into another .js file in your scripts dir, replace the contract address with your contracts address:



```
ERC20 > scripts > JS query.js > ...
async function main(){
    const address = "0xe7f1725e7734ce288f8367e1bb143e90bb3f0512"
    const contract = await ethers.getContractAt("Token", address);

    const name = await contract.name();
    const symbol = await contract.symbol();
    const myBalance = await contract.balanceOf("0xf39fd6e51aad88f6f4ce
    const totalSupply = await contract.totalSupply();

    console.log(name);
    console.log(symbol);
    console.log("My Balance: ", myBalance.toNumber());
    console.log("Total Token Supply: ", totalSupply.toNumber());
}

main();
```



```
D:\Development\EthereumSmartContractExamples\ERC20>npx hardhat run ./scripts/query.js --network localhost
NIXS BUX
NIX
My Balance:   500
Total Token Supply:   500
```

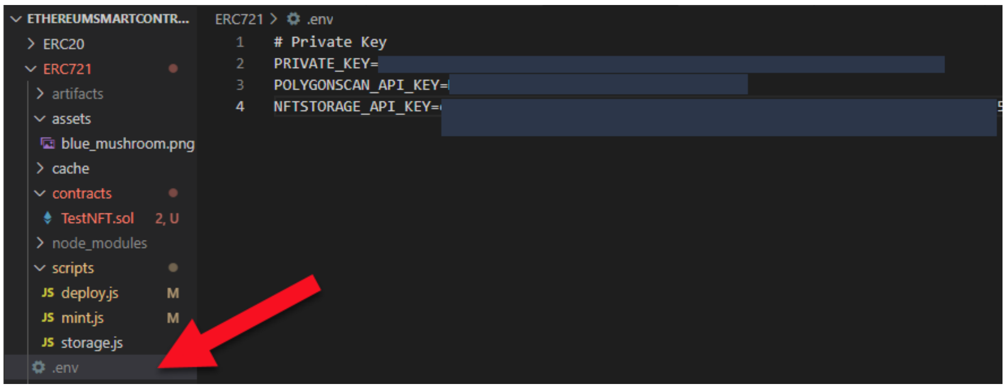
Well done, we've been able to deploy the contract and read information from it.

2.3 Deploying to the Polygon network

Now that we have deployed and tested our contract on the local test environment we are ready to deploy to a real network. We will be using the polygon network because it's actually affordable to do transactions on. Anytime you want to change the state of the blockchain (ie deploy a contract, run a contract's function, change the owner of a thing) you will need to pay gas fees. Follow the brief documentation below to learn how to deploy to the polygon network or refer to section 3.6 of this document on how setup in detail.

Next to deploy to a real network like polygon refer to the documentation here:

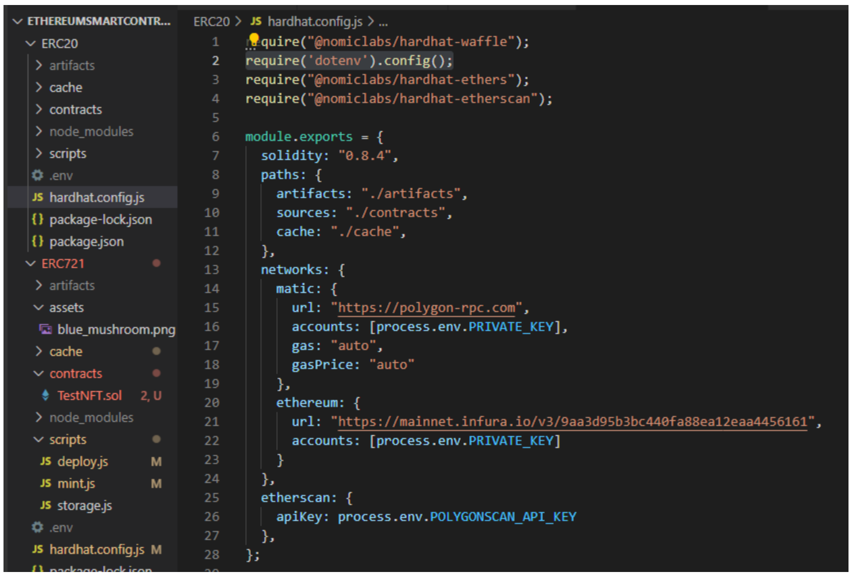
Using Hardhat | Polygon Technology | Documentation
Make a .env file in the root of the project dir and install the dotenv package via npm then make it like this:



```
# Private Key
PRIVATE_KEY=[redacted]
POLYGONSCAN_API_KEY=[redacted]
NFTSTORAGE_API_KEY=[redacted]
```

Get your polygon scan API key from your polygon scan account and private key from your metamask wallet.

Hardhat config:



```
quire("@nomiclabs/hardhat-waffle");
require('dotenv').config();
require("@nomiclabs/hardhat-ethers");
require("@nomiclabs/hardhat-etherscan");

module.exports = {
  solidity: "0.8.4",
  paths: {
    artifacts: "./artifacts",
    sources: "./contracts",
    cache: "./cache",
  },
  networks: {
    matic: {
      url: "https://polygon-rpc.com",
      accounts: [process.env.PRIVATE_KEY],
      gas: "auto",
      gasPrice: "auto"
    },
    ethereum: {
      url: "https://mainnet.infura.io/v3/9aa3d95b3bc440fa88ea12eaa4456161",
      accounts: [process.env.PRIVATE_KEY]
    }
  },
  etherscan: {
    apiKey: process.env.POLYGONSCAN_API_KEY
  },
};
```

Commands:
npx hardhat run .\scripts\deploy.js --network matic
npx hardhat verify --network matic [address of contract]

Or the code in the git here:

Nixxs/EthereumSmartContractExamples: Simple example projects for creating smart contracts on the etherium blockchain and how to interact with them. (github.com)

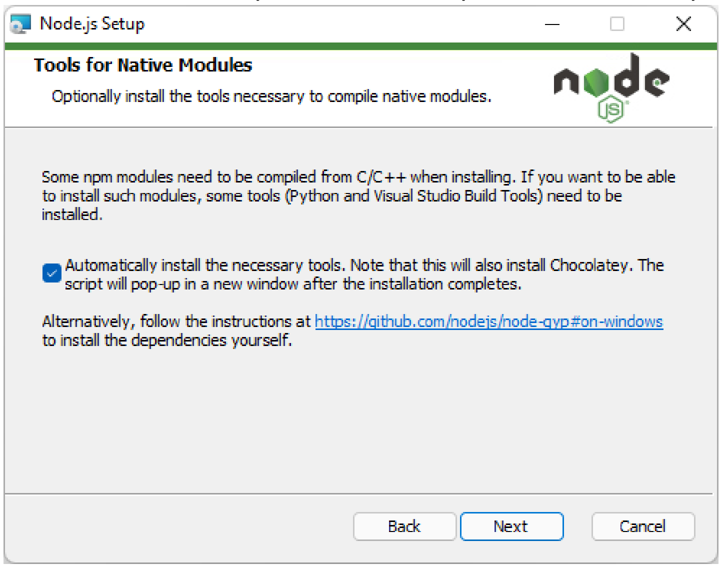
The token on polygonscan:
NIXS BUX (NIX) Token Tracker | PolygonScan

3 ERC721 NFT Tokens

3.1 Setting up the local dev environment

Before we can get started we'll need to make sure we have all the prerequisite software installed. Follow the steps below to get your local dev environment up and running ready for blockchain development.

1. Install NodeJS it comes with NPM which we will use to install all the required libraries
    a. Node.js (nodejs.org)
    b. Tick the automatically install necessary tool so it has everything it needs to compile things:



Tools for Native Modules

Optionally install the tools necessary to compile native modules.

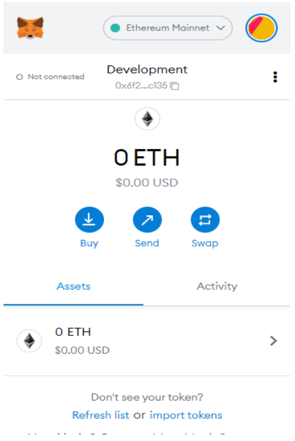
Some npm modules need to be compiled from C/C++ when installing. If you want to be able to install such modules, some tools (Python and Visual Studio Build Tools) need to be installed.

[✓] Automatically install the necessary tools. Note that this will also install Chocolatey. The script will pop-up in a new window after the installation completes.

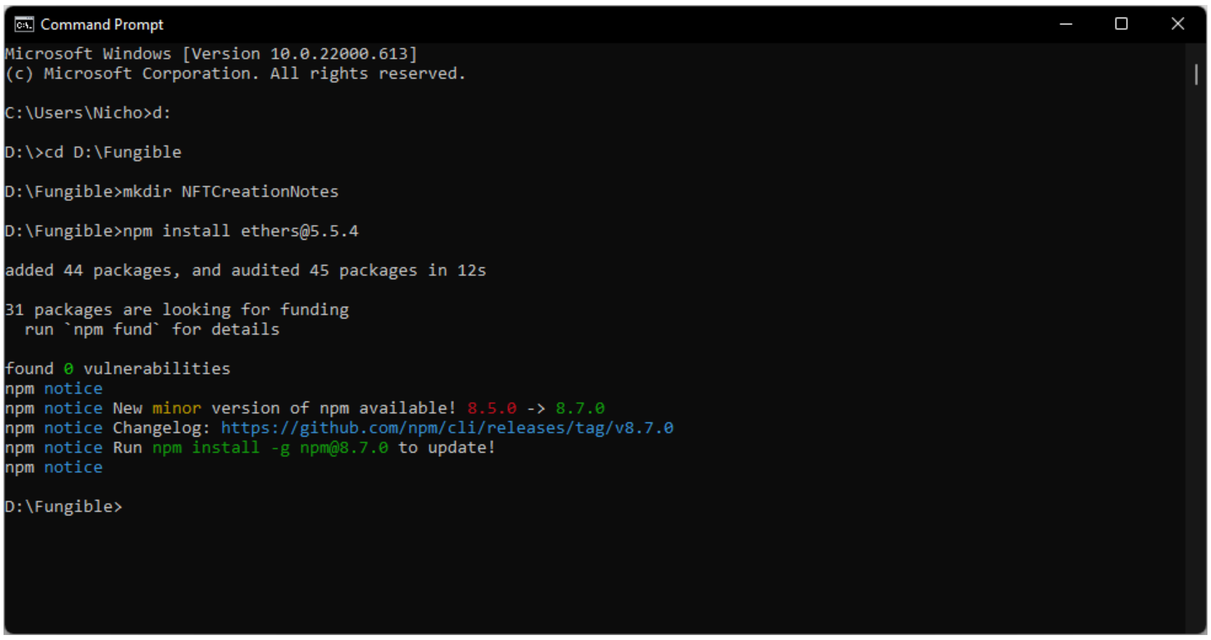
Alternatively, follow the instructions at https://github.com/nodejs/node-gyp#on-windows to install the dependencies yourself.

2. Install hardhat – hardhat is a set of tools that will allow you to create a local blockchain so that you can test deployment of contracts in a dev environment without having to deploy out to a real blockchain that would cost you gas.
    a. Now that you have nodejs/npm installed you can just open your console and run this command to install it:
        i. Npm install –save-dev hardhat@2.8.4
        ii. Installing this version of hardhat specifically because that's the one I setup my demo project with

3. Next install metmask extension onto your browser, you need this so that you can have a wallet for hardhat to use for deployment.
    a. The crypto wallet & gateway to Web3 blockchain apps | MetaMask
    b. Once installed setup an account to create your wallet, you can just call it development or whatever:



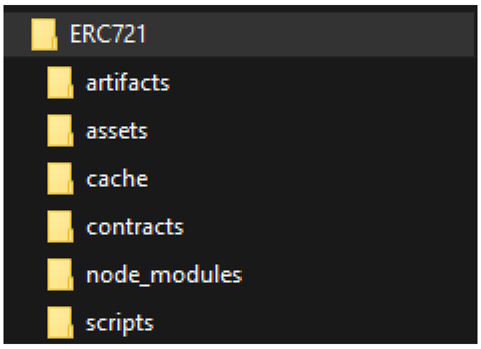
4. Now we are ready to make our project folder so create the folder, change directory into it then install all the required packages using NPM commands below:
   a. Npm install ethers@5.5.4
   b. Npm install hardhat@2.8.4
   c. Npm install @nomiclabs/hardhat-ethers@2.0.5
   d. Npm install @nomiclabs/hardhat-waffle@2.0.2
   e. Npm install @openzeppelin/contracts@4.5.0



```
Command Prompt
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Nicho>d:

D:\>cd D:\Fungible

D:\Fungible>mkdir NFTCreationNotes

D:\Fungible>npm install ethers@5.5.4

added 44 packages, and audited 45 packages in 12s

31 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
npm notice
npm notice New minor version of npm available! 8.5.0 -> 8.7.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v8.7.0
npm notice Run npm install -g npm@8.7.0 to update!
npm notice

D:\Fungible>
```

   f. Ethers is a library that will allow your front-end application to interact with the blockchain and openzeppelin contracts are a set of standardized contract code we can import into our smart contract to make them compliant with ERC-721 NFT standards without have to write all the functionality ourselves.

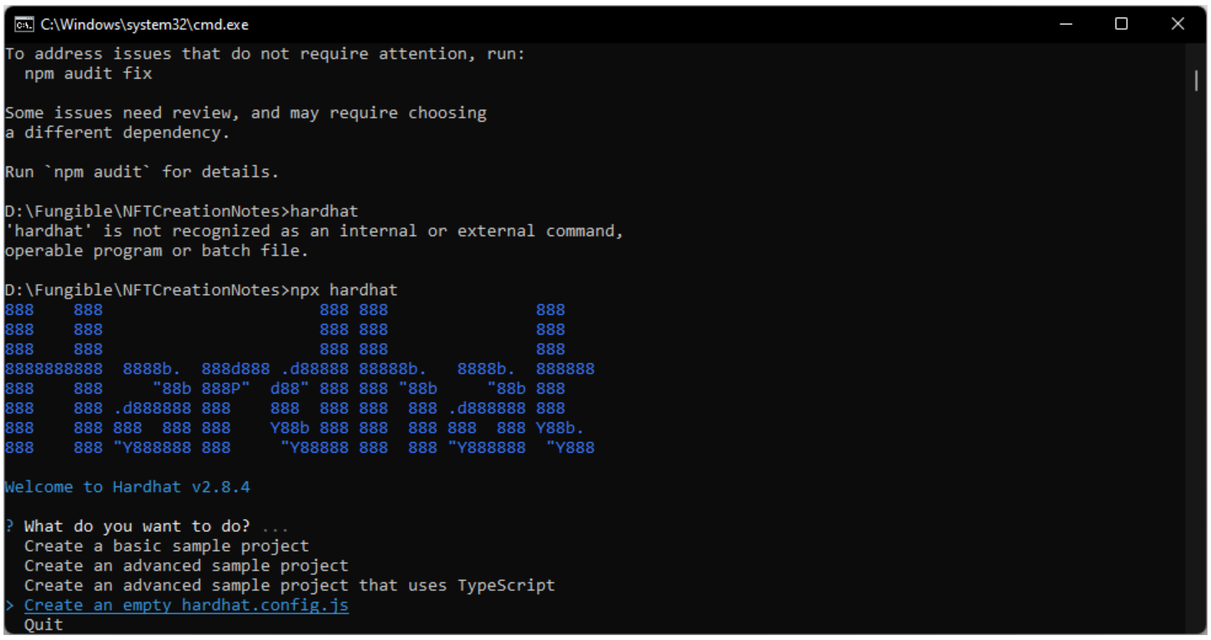
5. All your dependencies are now installed and ready to go, next we will make some folders to store our contract and a deployment script which will be used to deploy our contract to the blockchain. Hardhat will also need a folder to store "artifacts" as well as "cache":
   a. Setup your folders so they look something like this:



```
ERC721
  artifacts
  assets
  cache
  contracts
  node_modules
  scripts
```

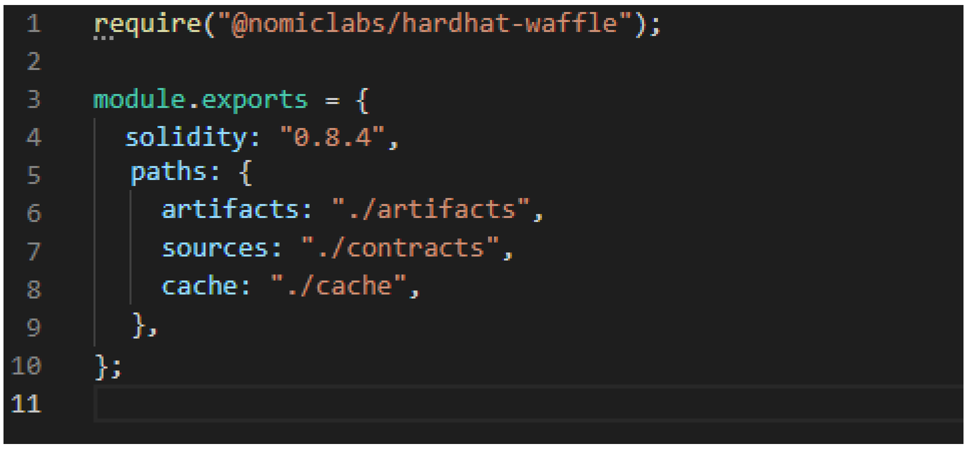
6. Now we will run hardhat in our project directory and tell it to setup a hardhat.config.js file so that we can setup hardhat to run properly in our environment. From the project root directory run:

   npx hardhat and select “Create and empty hardhat.config.js”:



```
C:\Windows\system32\cmd.exe
To address issues that do not require attention, run:
  npm audit fix

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.

D:\Fungible\NFTCreationNotes>hardhat
'hardhat' is not recognized as an internal or external command,
operable program or batch file.

D:\Fungible\NFTCreationNotes>npx hardhat
888    888                      888 888               888
888    888                      888 888               888
888    888                      888 888               888
8888888888  8888b.  888d888 .d88888 88888b.   8888b.  888888
888    888     "88b 888P"  d88" 888 888 "88b     "88b 888
888    888 .d888888 888    888  888 888  888 .d888888 888
888    888 888  888 888    Y88b 888 888  888 888  888 Y88b.
888    888 "Y888888 888     "Y88888 888  888 "Y888888  "Y888

Welcome to Hardhat v2.8.4

? What do you want to do? ...
  Create a basic sample project
  Create an advanced sample project
  Create an advanced sample project that uses TypeScript
> Create an empty hardhat.config.js
  Quit
```

7. Edit your new hardhat.config.js file so that it looks like the below, basically we are just telling hardhat what version of solidity we are running and what folders it should use:



```js
require("@nomiclabs/hardhat-waffle");

module.exports = {
  solidity: "0.8.4",
  paths: {
    artifacts: "./artifacts",
    sources: "./contracts",
    cache: "./cache",
  },
};
```

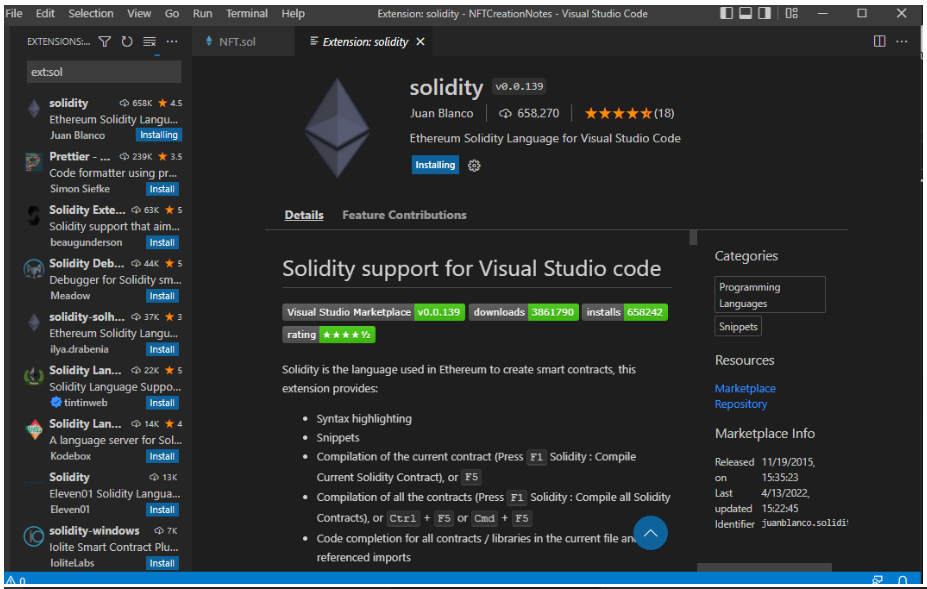
   Also make sure you include the require statement seen above otherwise deploying later will return an error as hardhat won’t have access to ethers. A library that we need for deploying.

3.2 Writing a ERC721 NFT Contract

Next, we will write the ERC721 NFT contract file and we will be using the openzeppelin contracts to make it quick and easy to comply to with the ERC721 standard.

8. Hardhat is ready to go, next we will write our NFT contract into the contracts folder we created. Contracts are written in a language called "Solidity" and have a ".sol" extension. So create a file inside the contracts folder called "NFT.sol":

   Install the solidity extension for VSCode as well if that's what you are using:


File
Edit
Selection
View
Go
Run
Terminal
Help
Extension: solidity - NFTCreationNotes - Visual Studio Code
EXTENSIONS:...
NFT.sol
Extension: solidity
ext:sol
solidity 658K 4.5 Ethereum Solidity Langu... Juan Blanco Installing
Prettier - ... 239K 3.5 Code formatter using pr... Simon Siefke Install
Solidity Exte... 63K 5 Solidity support that aim... beaugunderson Install
Solidity Deb... 44K 5 Debugger for Solidity sm... Meadow Install
solidity-solh... 37K 3 Ethereum Solidity Langu... ilya.drabenia Install
Solidity Lan... 22K 5 Solidity Language Suppo... tintinweb Install
Solidity Lan... 14K 4 A language server for Sol... Kodebox Install
Solidity 13K Eleven01 Solidity Langua... Eleven01 Install
solidity-windows 7K Iolite Smart Contract Plu... IoliteLabs Install
solidity v0.0.139
Juan Blanco | 658,270 | ★★★★★(18)
Ethereum Solidity Language for Visual Studio Code
Installing
Details
Feature Contributions
Categories
Programming Languages
Snippets
Resources
Marketplace
Repository

Solidity support for Visual Studio code

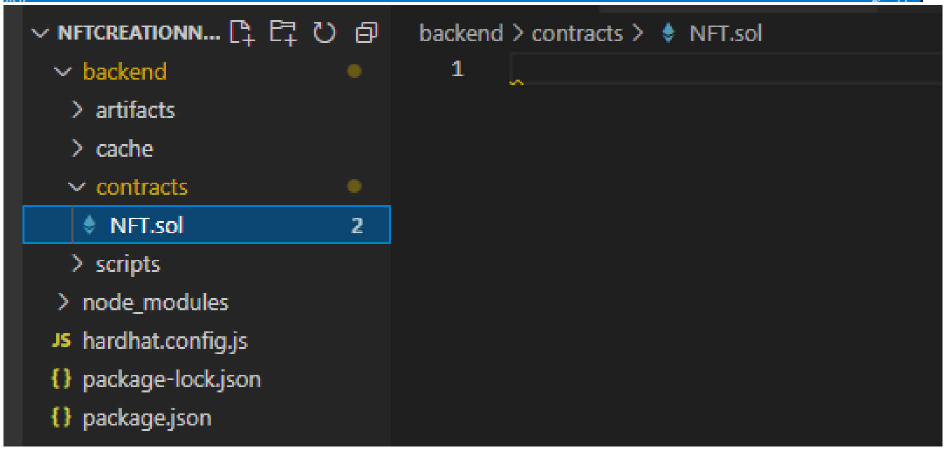
Visual Studio Marketplace v0.0.139 downloads 3861790 installs 658242 rating ★★★★½

Solidity is the language used in Ethereum to create smart contracts, this extension provides:

- Syntax highlighting
- Snippets
- Compilation of the current contract (Press `F1` Solidity : Compile Current Solidity Contract), or `F5`
- Compilation of all the contracts (Press `F1` Solidity : Compile all Solidity Contracts), or `Ctrl` + `F5` or `Cmd` + `F5`
- Code completion for all contracts / libraries in the current file an referenced imports

Marketplace Info

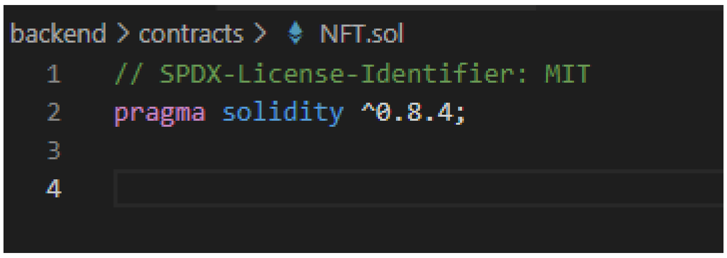
Released on	11/19/2015, 15:35:23
Last updated	4/13/2022, 15:22:45
Identifier	juanblanco.solidi



9. First thing we need to define is the license type in a comment as well as the version of solidity we are writing our code in.

   This is required because when we deploy our contract on to the blockchain it will be run in something called the Ethereum Virtual Machine (EVM) which, throughout time will come in many different versions.

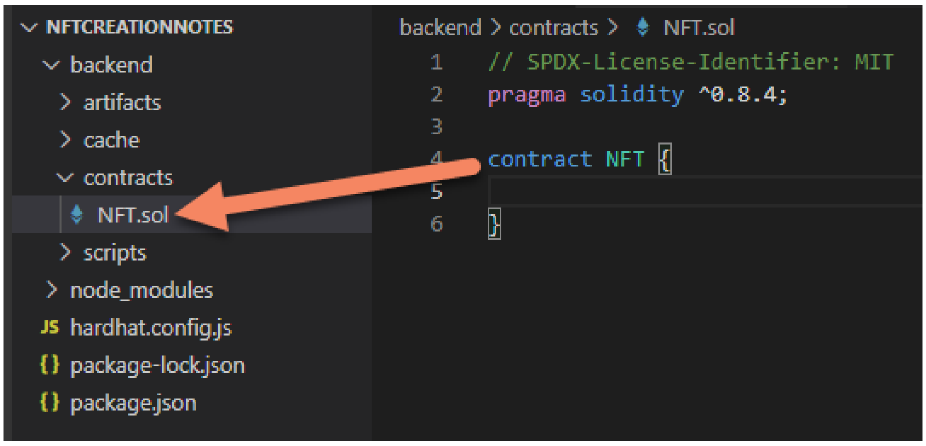
So, to make sure our contract code always works we define this at the top of the file with the "pragma" keyword:



backend > contracts > NFT.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;


```

10. All of the code inside a .sol file belong within a "contract" so next we define the contract and call it "NFT" the same as the file name:

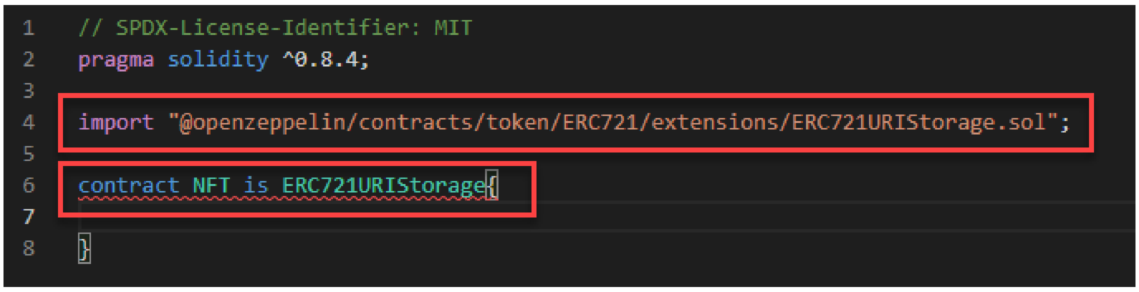


```
NFTCREATIONNOTES
  backend
    artifacts
    cache
    contracts
      NFT.sol
    scripts
  node_modules
  hardhat.config.js
  package-lock.json
  package.json
```

backend > contracts > NFT.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract NFT {

}
```

11. Next, instead of writing all the required ERC-721 standard functions ourselves we will just import them and have our NFT contract inherit all of them with the below additions to our code:



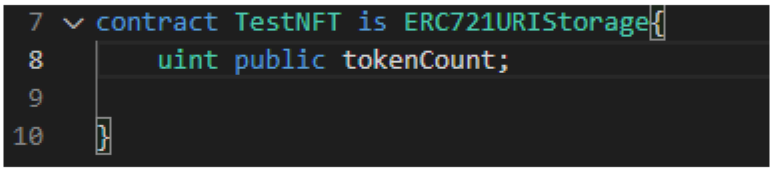
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";

contract NFT is ERC721URIStorage{

}
```

The "is" keyword is used to tell solidity what to inherit from.

12. Next, we will declare a state variable in the contract so that we can keep track of the number of tokens that have been minted from this contract and will also work as the token's ID when it gets minted. After deployment, this contract will be used to create all of the tokens going forward.

You access token information by first referencing this contract on the blockchain and then the ID of the token that is given to it during minting to get a URI which will then lead you off to the token's metadata.

The state variable will need to be accessible outside of this contract once it's deployed, it will also be an unsigned integer data type. So we create it like this:



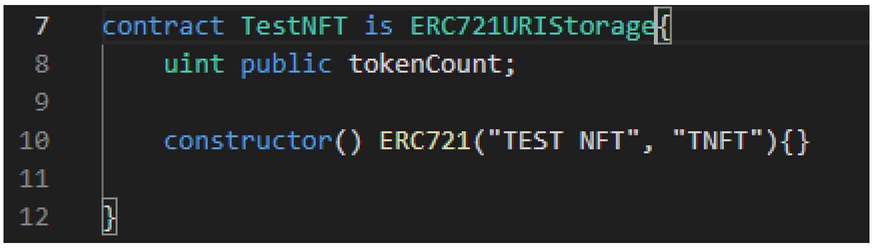
```
contract TestNFT is ERC721URIStorage{
    uint public tokenCount;

}
```

Solidity will automatically instantiate undefined variables with that data type's default value. For uint, this is 0.

13. Next, we have inherited all the functions from ERC721URIStorage contract so our contract is already ERC721 compliant, however for these to work we will need to run the constructor for the library. A constructor in solidity works the same as any other constructor, it only runs once when the contract is deployed to the block chain to set everything up.

    The ERC721 constructor needs us to define a token name and token symbol, all the NFTs created with this contract will be of this name and symbol but will just have a unique ID as well as a URI which will lead to more metadata about the NFT.

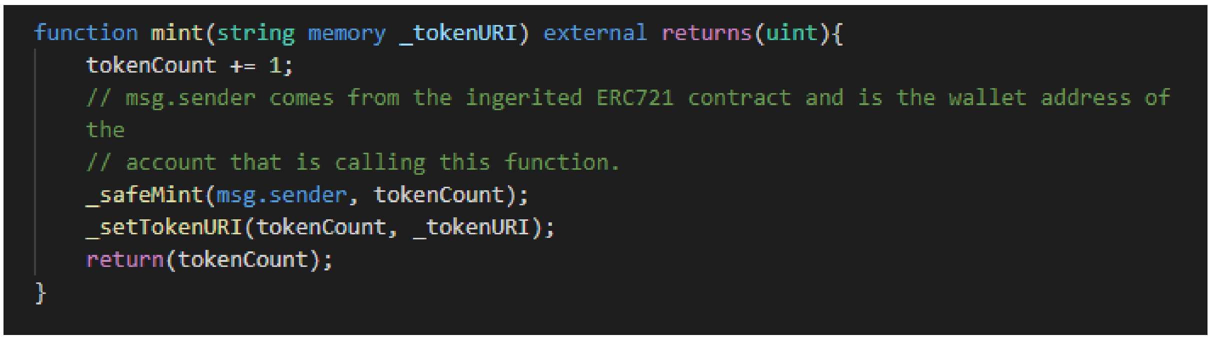
    We can run the constructor with the below syntax:



```
contract TestNFT is ERC721URIStorage{
    uint public tokenCount;

    constructor() ERC721("TEST NFT", "TNFT"){}

}
```

14. Now the constructor has run, all that is left is to create the externally executable minting function which will increment the tokenCount variable and use this as an ID for the creation of a new NFT token.

15. The function written below takes a URI as input and returns a uint which will be the ID of the NFT token. Inside the function, we simply increment tokenCount and then run the _safeMint() function which was inherited from the ERC721 library and pass in msg.sender which is the address of the person running the mint function and the ID of the token which is just gotten from the now incremented token count.



```
function mint(string memory _tokenURI) external returns(uint){
    tokenCount += 1;
    // msg.sender comes from the ingerited ERC721 contract and is the wallet address of
    the
    // account that is calling this function.
    _safeMint(msg.sender, tokenCount);
    _setTokenURI(tokenCount, _tokenURI);
    return(tokenCount);
}
```

next we need to set the token's URI which we get as an input from the function call then return the token ID (which is tokenCount). This tokenURI is where we have stored the token's metadata, when the mint function was called. So you would create all the relevant token content then host it somewhere else (like on IPFS explained later) and get a URI back then call this mint function to create the token with the URI of where the metadata is stored.

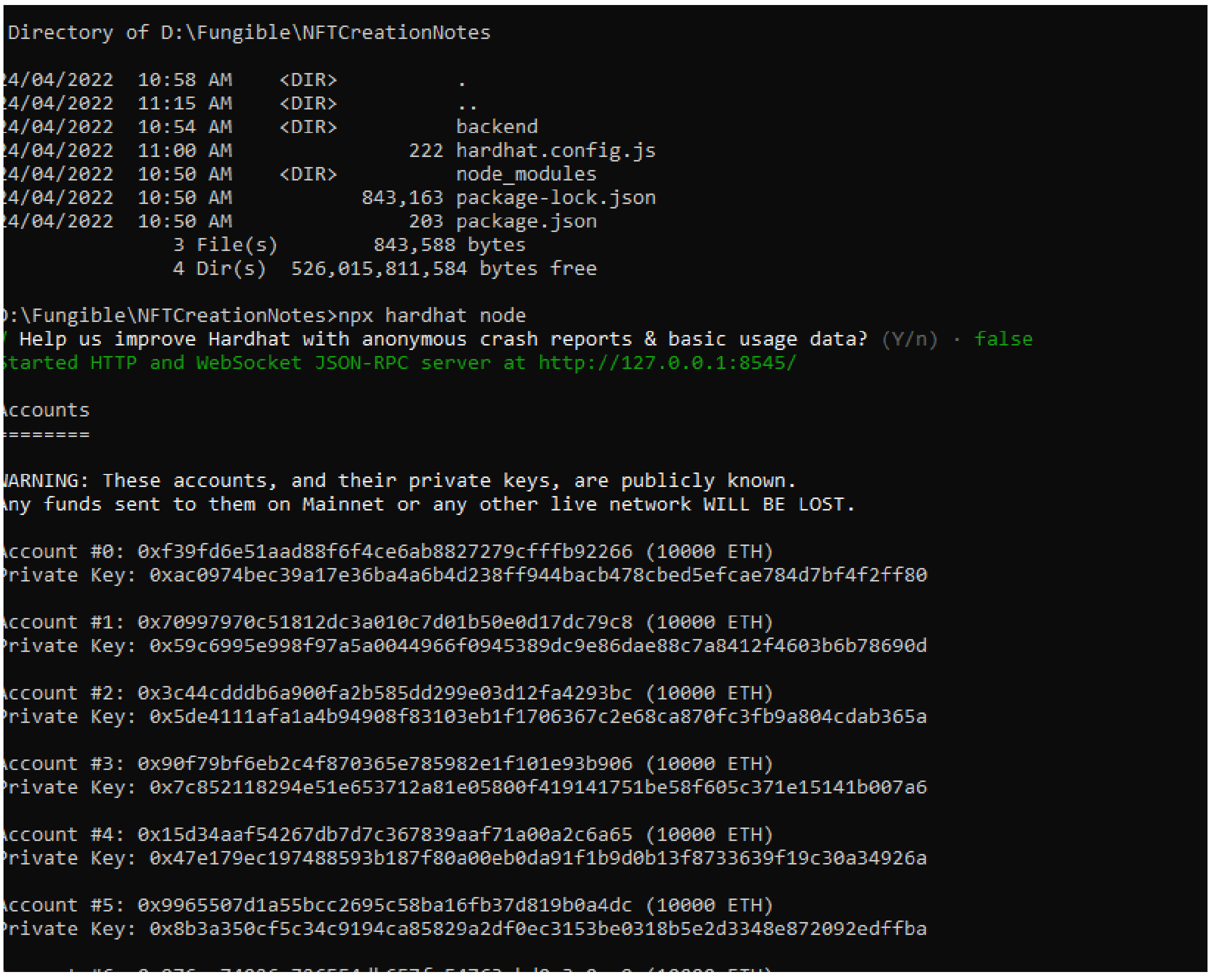
3.3 Deploying the contract to a local blockchain

You've now got the contract ready next we will need to deploy the contract to blockchain so we can call its functions, we will first deploy it to a local blockchain for testing. This is super important because once you deploy to a real blockchain it can never be updated ever again.

1. Before we can deploy our contract to a block chain, we first need a blockchain network to deploy to. We don't want to deploy to etherium because it'll cost you ETH. So instead, we can use hardhat to spin up a local block chain.

   Use the command below in your console window or from visual studio code to start up your local block chain:

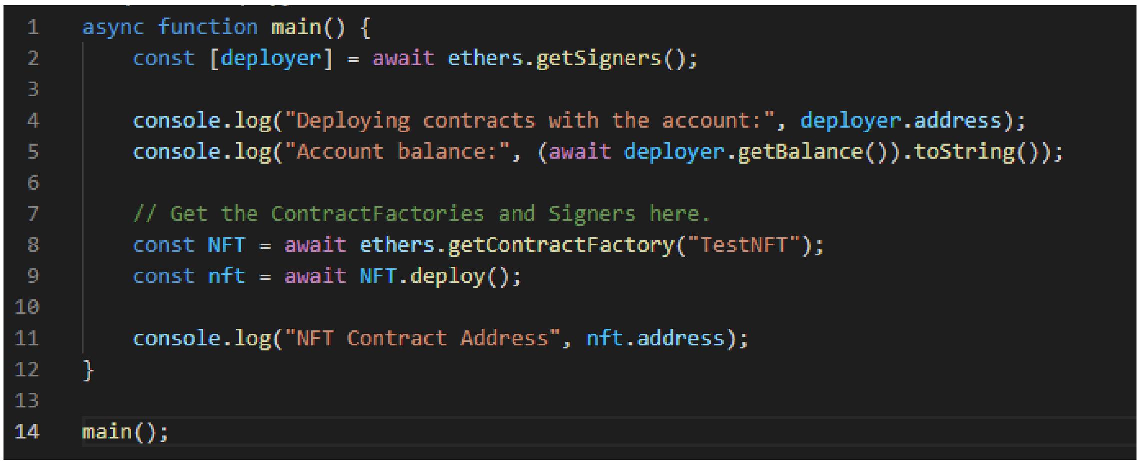
   npx hardhat node



```
 Directory of D:\Fungible\NFTCreationNotes

4/04/2022  10:58 AM    <DIR>          .
4/04/2022  11:15 AM    <DIR>          ..
4/04/2022  10:54 AM    <DIR>          backend
4/04/2022  11:00 AM               222 hardhat.config.js
4/04/2022  10:50 AM    <DIR>          node_modules
4/04/2022  10:50 AM           843,163 package-lock.json
4/04/2022  10:50 AM               203 package.json
              3 File(s)        843,588 bytes
              4 Dir(s)  526,015,811,584 bytes free

:\Fungible\NFTCreationNotes>npx hardhat node
 Help us improve Hardhat with anonymous crash reports & basic usage data? (Y/n) · false
tarted HTTP and WebSocket JSON-RPC server at http://127.0.0.1:8545/

ccounts
=======

ARNING: These accounts, and their private keys, are publicly known.
ny funds sent to them on Mainnet or any other live network WILL BE LOST.

ccount #0: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266 (10000 ETH)
rivate Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

ccount #1: 0x70997970c51812dc3a010c7d01b50e0d17dc79c8 (10000 ETH)
rivate Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

ccount #2: 0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc (10000 ETH)
rivate Key: 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

ccount #3: 0x90f79bf6eb2c4f870365e785982e1f101e93b906 (10000 ETH)
rivate Key: 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6

ccount #4: 0x15d34aaf54267db7d7c367839aaf71a00a2c6a65 (10000 ETH)
rivate Key: 0x47e179ec197488593b187f80a00eb0da91f1b9d0b13f8733639f19c30a34926a

ccount #5: 0x9965507d1a55bcc2695c58ba16fb37d819b0a4dc (10000 ETH)
rivate Key: 0x8b3a350cf5c34c9194ca85829a2df0ec3153be0318b5e2d3348e872092edffba
```

   you can see this has created a block chain all a whole bunch of accounts for us to use with 10000 fake ETH on them running on our local machine.

2. Now we have a blockchain to deploy to, lets setup the deployment script. Inside the scripts directory create a deploy.js while which we will eventually run with node. Enter the below into the deploy.js file:



```
async function main() {
    const [deployer] = await ethers.getSigners();

    console.log("Deploying contracts with the account:", deployer.address);
    console.log("Account balance:", (await deployer.getBalance()).toString());

    // Get the ContractFactories and Signers here.
    const NFT = await ethers.getContractFactory("TestNFT");
    const nft = await NFT.deploy();

    console.log("NFT Contract Address", nft.address);
}

main();
```

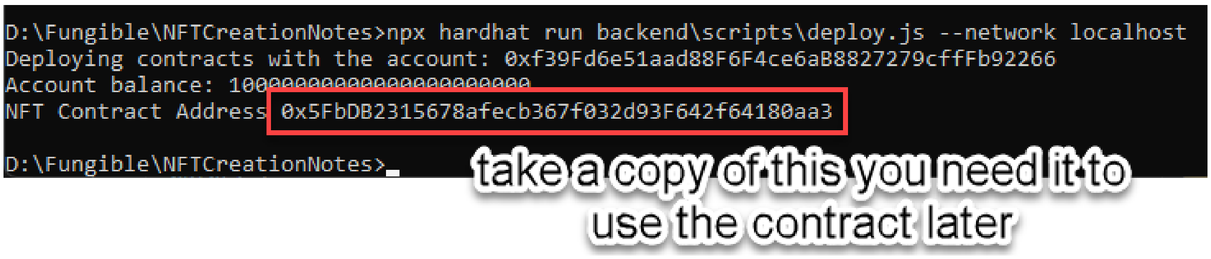
3. There are really only 3 lines to this main function, the [deployer] line which is where we use the ethers library to first create a “contract factory” object which we can use to create the contract object using the name of our contract (we called it NFT) then we run the deploy() on that object to de ploy it.

   This will return the address of the deployed contract which is all we need to get to it later on our front end application.

   Run this deployment script and deploy the NFT contract to the local blockchain with:

   npx hardhat run ./backend/scripts/deploy.js –network localhost

   note when you want to deploy to a real network refer to the polygon documentation for hardhat on how to do it but it’s not too different to the above. You just need to put a few settings in the hardhat.config.js file is all.



```
D:\Fungible\NFTCreationNotes>npx hardhat run backend\scripts\deploy.js --network localhost
Deploying contracts with the account: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266
Account balance: 10000000000000000000000
NFT Contract Address 0x5FbDB2315678afecb367f032d93F642f64180aa3

D:\Fungible\NFTCreationNotes>_
```

take a copy of this you need it to
use the contract later

On successful deployment we can see that our NFT contract has been deployed and it’s address is printed using that console.log() in the deploy script. You can also see the account balance before deploy. If you printed the account balance after deploy you will have seen that it’s gone down because it costs ETH to make any changes to the blockchain.

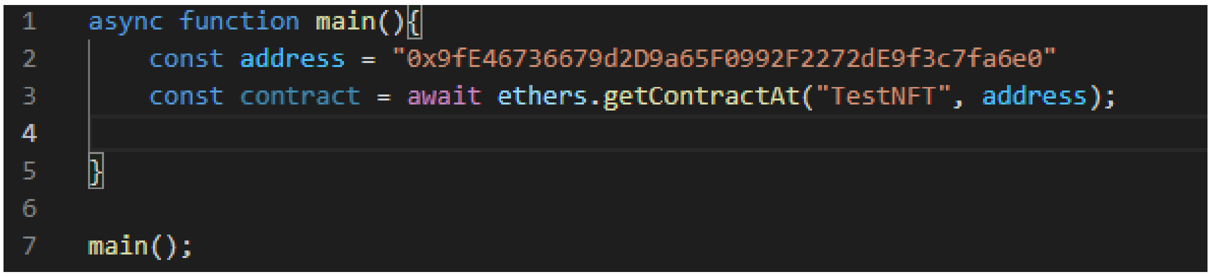
you’ve now created an NFT contract and deployed it to a local blockchain. All that is left is to use the mint function from the deployed contract to mint a token.

3.4 Minting a new NFT token using the deployed ERC721 NFT Contract

The contract has now been deployed which means we can now call the mint function to make new NFT tokens out of it. The contract will keep track of all the tokens it creates for now on but it can never be updated now that it is deployed. In this section new will use the new contract to mint new tokens:

1. Next, we will make a mint.js file in the scripts folder to mint an NFT and print out a few things to the screen so we can see how a front end application might be able to interact with our newly deployed contract. Create the mint.js file.

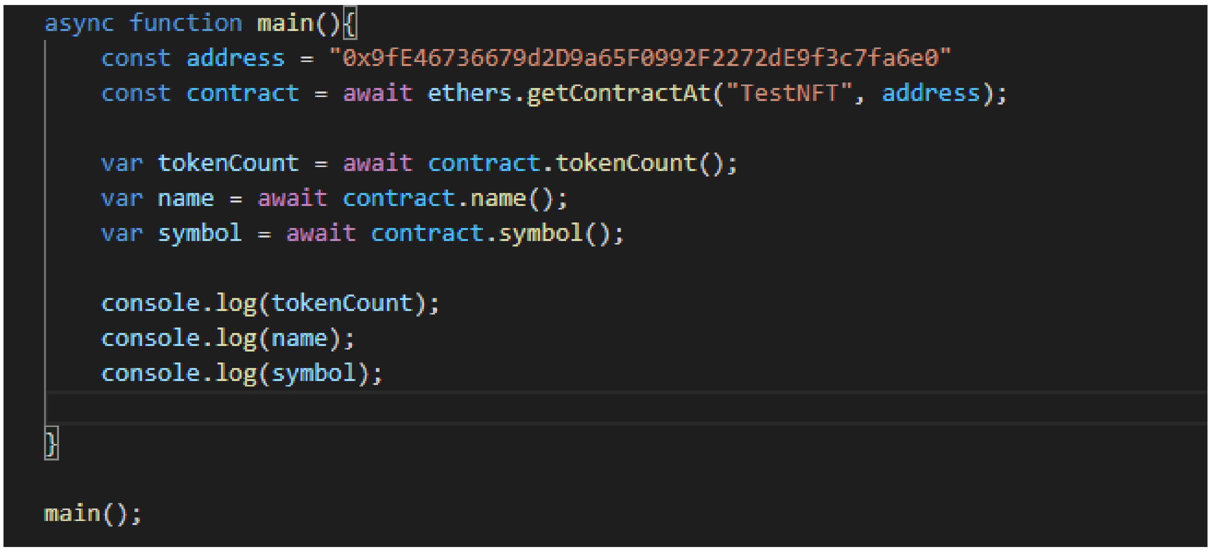
   First, we need to get the contract via it's name and address on the local blockchain using ethers.getContractAt() function:



```
async function main(){
    const address = "0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0"
    const contract = await ethers.getContractAt("TestNFT", address);

}

main();
```

   Any requests to the blockchain must be asynchronous because it is slow. We've now stored the contract object in the "contract" variable.

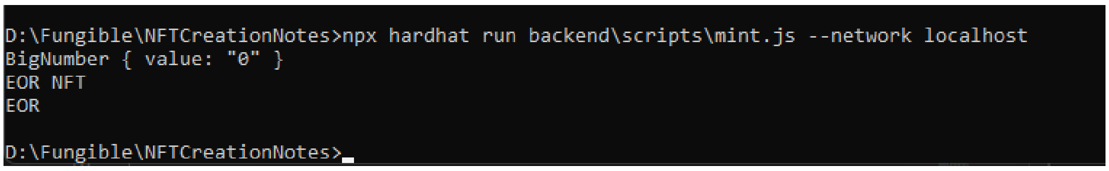
2. Next, lets try reading some of the data that is stored on the blockchain hosted contract, fill in the below into your mint.js file:



```
async function main(){
    const address = "0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0"
    const contract = await ethers.getContractAt("TestNFT", address);

    var tokenCount = await contract.tokenCount();
    var name = await contract.name();
    var symbol = await contract.symbol();

    console.log(tokenCount);
    console.log(name);
    console.log(symbol);

}

main();
```

3. Before we go on, lets run this so you can see that we can get data from the contract. Notice the tokenCount() function we have available to us to use to get the tokenCount value that we created a public state variable in the NFT.sol we wrote earlier.

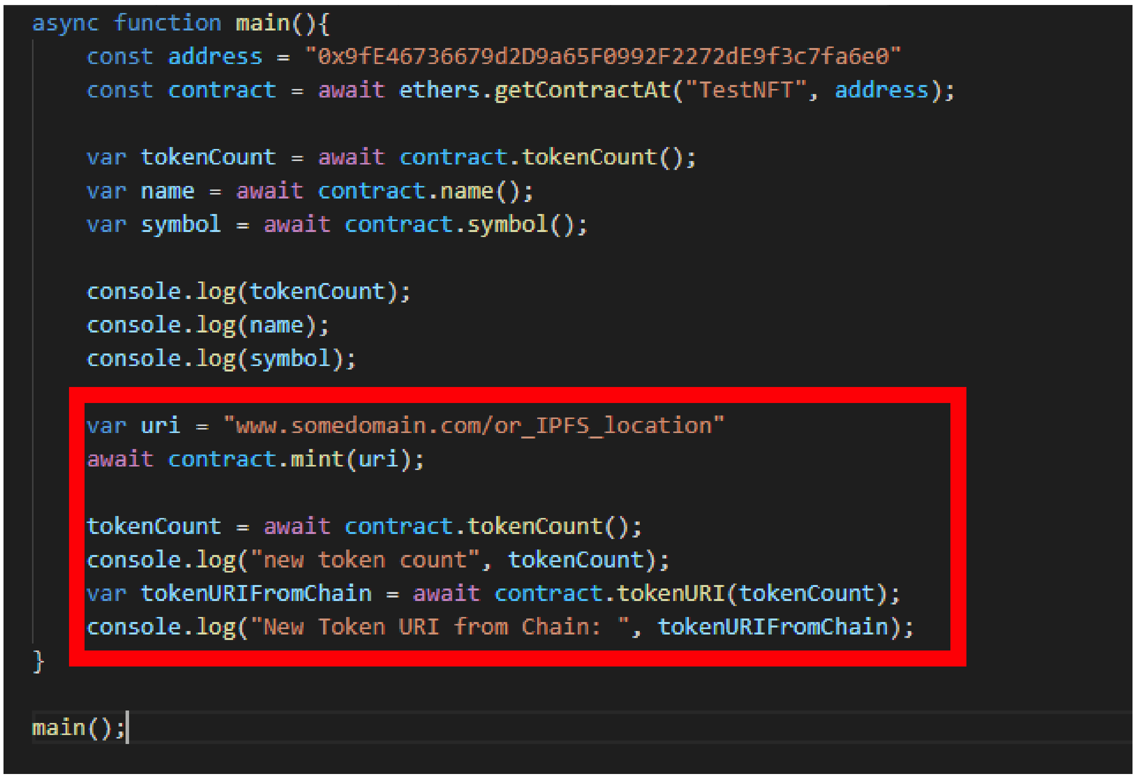
   Run mint.js file in the console with:



```
D:\Fungible\NFTCreationNotes>npx hardhat run backend\scripts\mint.js --network localhost
BigNumber { value: "0" }
EOR NFT
EOR

D:\Fungible\NFTCreationNotes>_
```

See how we have gotten those values back from our deployed blockchain? This means our co ntract was indeed successfully deployed and we can read it’s data.

4. Next, lets go ahead and mint a token. Put in a few extra lines into your mint.js file:

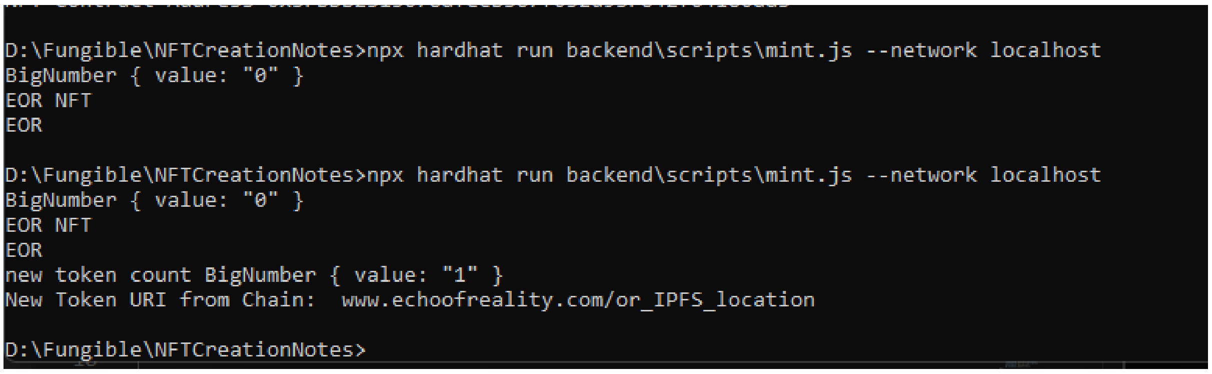


```
async function main(){
    const address = "0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0"
    const contract = await ethers.getContractAt("TestNFT", address);

    var tokenCount = await contract.tokenCount();
    var name = await contract.name();
    var symbol = await contract.symbol();

    console.log(tokenCount);
    console.log(name);
    console.log(symbol);

    var uri = "www.somedomain.com/or_IPFS_location"
    await contract.mint(uri);

    tokenCount = await contract.tokenCount();
    console.log("new token count", tokenCount);
    var tokenURIFromChain = await contract.tokenURI(tokenCount);
    console.log("New Token URI from Chain: ", tokenURIFromChain);
}

main();
```

Our NFTs will need to be minted with a URI, each minted NFT will have a unique URI which will correspond to a location on IPFS a decentralize storage location for metadata that is compatible with opensea. Setting up and interacting with IPFS is will be written in a different document. For now, just know that when we mint we need to give it a URI/URL that is related to the minted NFT.

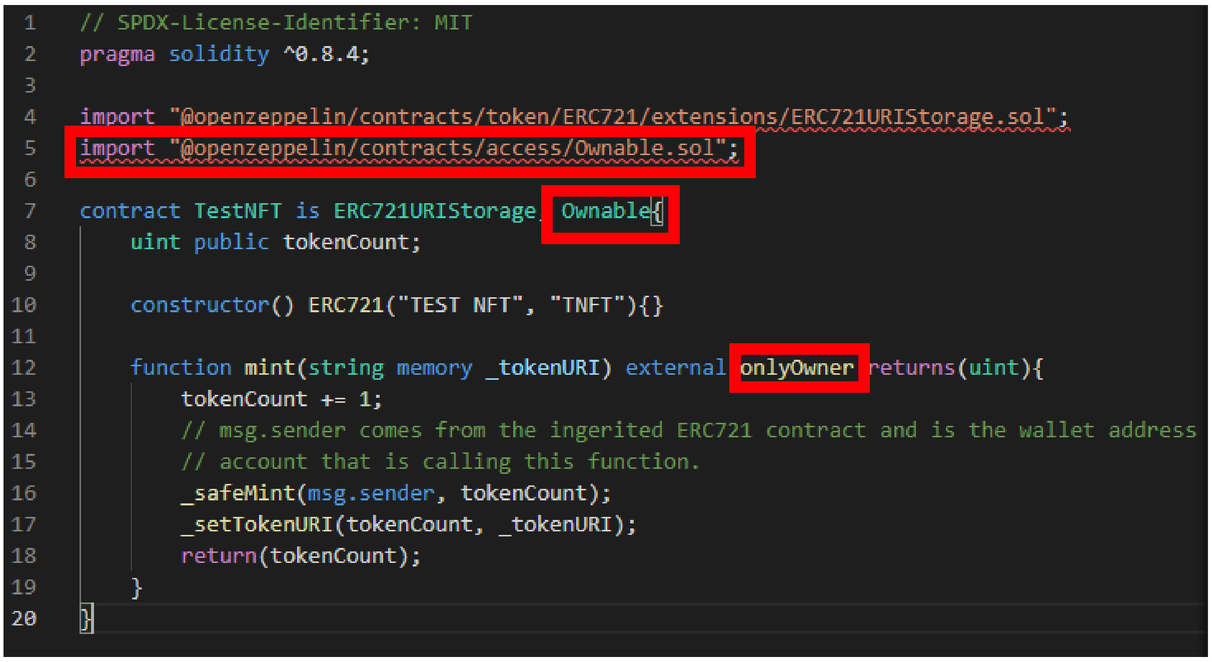
After minting, we go and retrieve the tokenCount from the contract again because we should have more than 0 now and we also assign with tokenCount to a variable because it is also the ID of the newly created token.

Finally, we test the token by asking for the tokenURI and passing in the token id which is “tokenCount” to ensure that we can get the token URI from the chain as we ll. Run the mint.js script from your console again:


```
D:\Fungible\NFTCreationNotes>npx hardhat run backend\scripts\mint.js --network localhost
BigNumber { value: "0" }
EOR NFT
EOR

D:\Fungible\NFTCreationNotes>npx hardhat run backend\scripts\mint.js --network localhost
BigNumber { value: "0" }
EOR NFT
EOR
new token count BigNumber { value: "1" }
New Token URI from Chain:  www.echoofreality.com/or_IPFS_location

D:\Fungible\NFTCreationNotes>
```

Congratulations you have just deployed your NFT contract and minted a token. One issue we still have to address at the moment is that anyone that has the address off the NFT contract will be able to mint tokens so to prevent this, we would normally setup our contract with access control as defined in the documentation here.

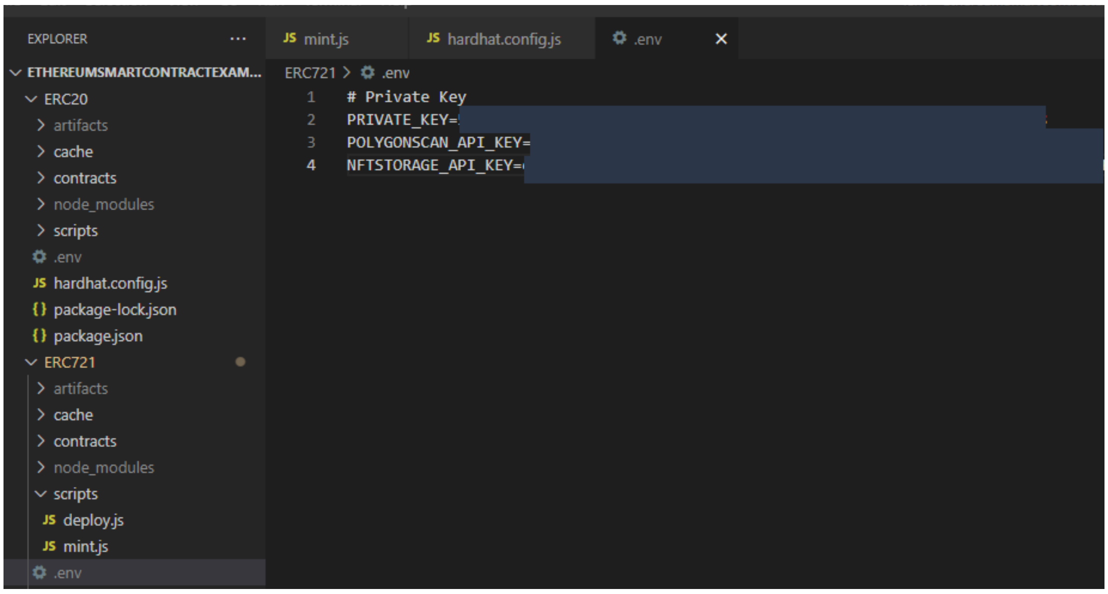
You can update your NFT.sol file with the below so that only the account that deployed the contract can run the mint function:


```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract TestNFT is ERC721URIStorage, Ownable{
    uint public tokenCount;

    constructor() ERC721("TEST NFT", "TNFT"){}

    function mint(string memory _tokenURI) external onlyOwner returns(uint){
        tokenCount += 1;
        // msg.sender comes from the ingerited ERC721 contract and is the wallet address
        // account that is calling this function.
        _safeMint(msg.sender, tokenCount);
        _setTokenURI(tokenCount, _tokenURI);
        return(tokenCount);
    }
}
```

3.5 Minting a new NFT token using the deployed ERC721 NFT Contract

The NFT token will only store a "tokenURI" value on the blockchain, but usually we will want to add an associated image to it or other metadata like "health", "armour", "strength" anything like that. Since we are doing it all decentralized we don't want to store that data on something like Amazon AWS or google cloud storage right? So instead, we can used IPFS (inter-planetary file storage) which is a decentralized file storage built on more blockchain tech.

1. Start my first going to https://nft.storage and make an account and API key. Then setup a .env file in the root of the project with your API key as a variable.



```
# Private Key
PRIVATE_KEY=[redacted]
POLYGONSCAN_API_KEY=[redacted]
NFTSTORAGE_API_KEY=[redacted]
```

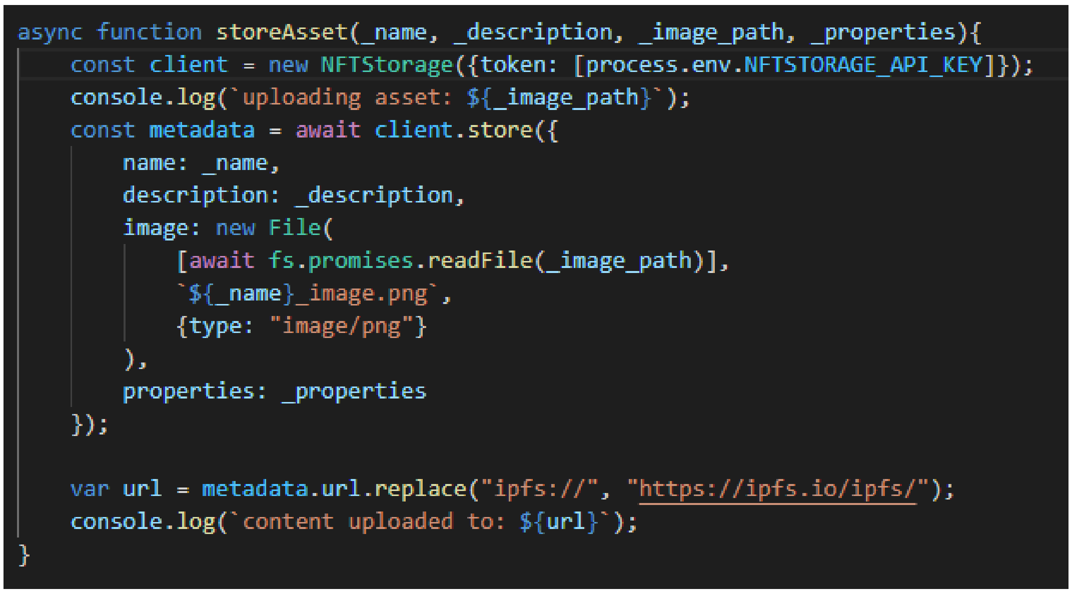
2. Next, install the required nft.storage libraries and while we are at it the "dotenv" library as well we need this to refer to the .env variables:
    a. npm install nft.storage
    b. npm install mime
    c. npm install dotenv

3. Prior to minting a new NFT we will first need to store the metadata and image for the NFT somewhere so now we will create the storage script for storing metadata and an image for our NFT.

   First, start by adding all the required libraries to the script, we will need nft storage the dotenv and a ref to FS to read the image file:

```
const { NFTStorage, File } = require("nft.storage");
require('dotenv').config();
var fs = require('fs');
```

4. Next we will make a function to store and upload the imagery along with some metadata properties:

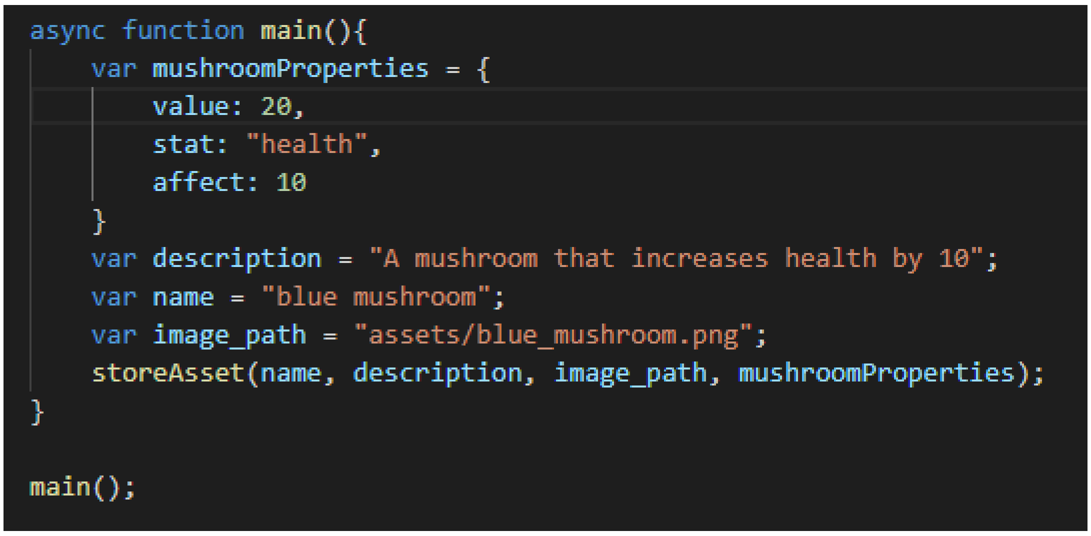


```
async function storeAsset(_name, _description, _image_path, _properties){
    const client = new NFTStorage({token: [process.env.NFTSTORAGE_API_KEY]});
    console.log(`uploading asset: ${_image_path}`);
    const metadata = await client.store({
        name: _name,
        description: _description,
        image: new File(
            [await fs.promises.readFile(_image_path)],
            `${_name}_image.png`,
            {type: "image/png"}
        ),
        properties: _properties
    });

    var url = metadata.url.replace("ipfs://", "https://ipfs.io/ipfs/");
    console.log(`content uploaded to: ${url}`);
}
```

We create the NFTStorage item by passing in our api key to the NFTStorage() constructor then use it to run it’s store() function which takes a name, description and the image file.

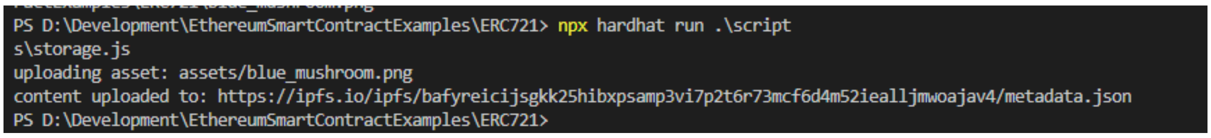
For the image file we use the FS library to read the image in and convert it into a nftstorage file object for the image property. Finally we define a properties item which will be a javascript object with custom values that can be passed in as a parameter to the function.

5. Alright now we just have to call the storeAsset() function inside a main function and give it the params we want:



```
async function main(){
    var mushroomProperties = {
        value: 20,
        stat: "health",
        affect: 10
    }
    var description = "A mushroom that increases health by 10";
    var name = "blue mushroom";
    var image_path = "assets/blue_mushroom.png";
    storeAsset(name, description, image_path, mushroomProperties);
}

main();
```

6. Now that everything is written up and ready to go we can run the script to store our blue_mushroom on IPFS



```
PS D:\Development\EthereumSmartContractExamples\ERC721> npx hardhat run .\script
s\storage.js
uploading asset: assets/blue_mushroom.png
content uploaded to: https://ipfs.io/ipfs/bafyreicijsgkk25hibxpsamp3vi7p2t6r73mcf6d4m52iealljmwoajav4/metadata.json
PS D:\Development\EthereumSmartContractExamples\ERC721>
```

IPFS will return the metadata URL which we print to the screen.

3.6 Deploying and running it on the polygon network

We've put together the NFT contract and we've got the code together to create the metadata for a new NFT now. The final step is to put all of this together and get this working on a real live blockchain we will be using Polygon.

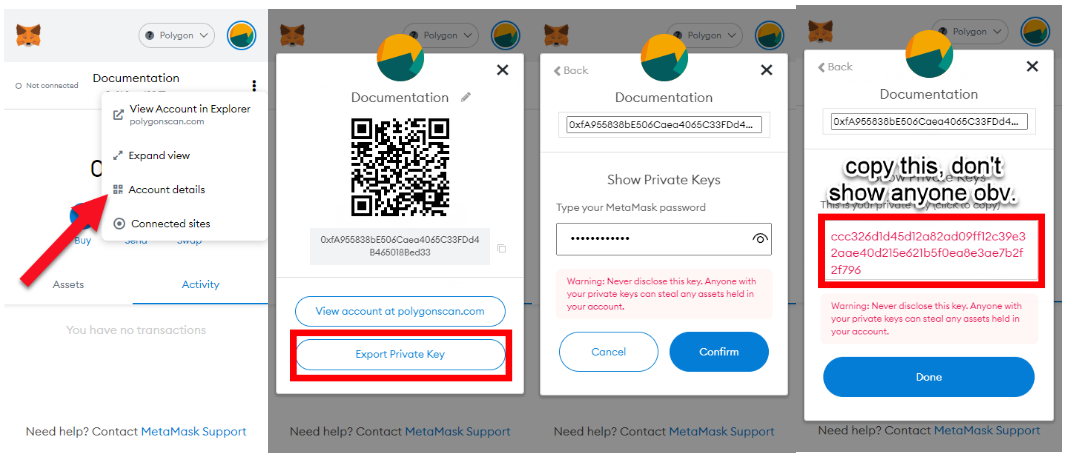
To start, make sure your active **metamask account** has been setup with a wallet that is connected to the **polygon network with at least 1 matic** in it to pay for gas during our deploy.

WARNING

If you are running through this document as a test and learning experience do not use any reference to "EOR" of "FUNGIBLE" in it's naming or any of its URLs because they will be on the chain forever and turn up in polygonscan if someone searches them.

Your .env file should have already been created when you went through the process for testing NFT storage api in the previous section but if now just make an NFT storage api and refer to the screenshots on how to put it into a .env (.env should be stored in the root of your project and install the dotenv library using npm). Inside that .env file you will have set the nft storage API, to deploy to polygon you will need your deploy wallet's private key as well as a polygonscan api key.

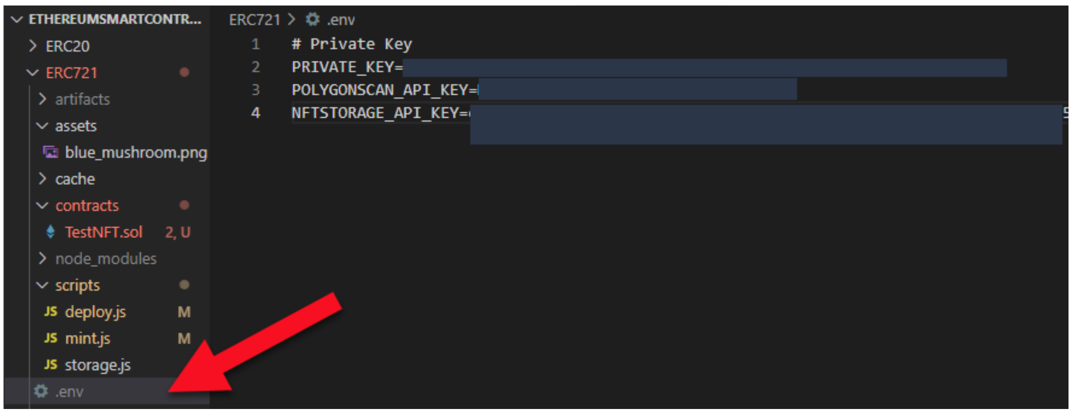
1. First, use metamask to get your account's private key:



Polygon
Not connected
Documentation
View Account in Explorer
polygonscan.com
Expand view
Account details
Connected sites
Assets
Activity
You have no transactions
Need help? Contact MetaMask Support
Documentation
0xfA955838bE506Caea4065C33FDd4B465018Bed33
View account at polygonscan.com
Export Private Key
Back
Show Private Keys
Type your MetaMask password
Warning: Never disclose this key. Anyone with your private keys can steal any assets held in your account.
Cancel
Confirm
copy this, don't show anyone obv.
ccc326d1d45d12a82ad09ff12c39e32aae40d215e621b5f0ea8e3ae7b2f2f796
Done

Don't worry this documentation account was created just for these screenshots.

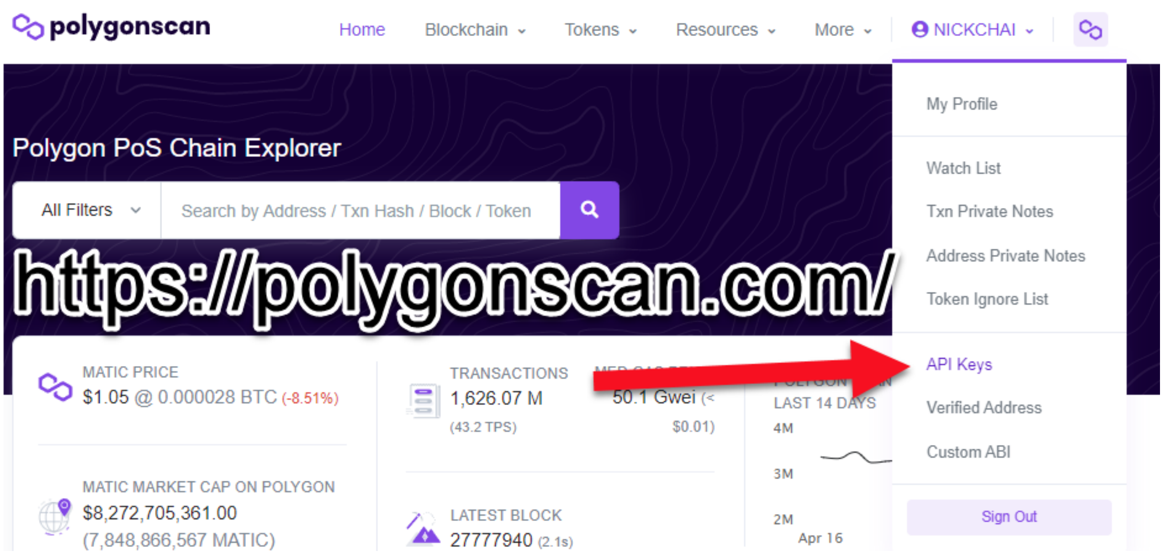
2. Copy your private key and create an entry into the .env file like below:



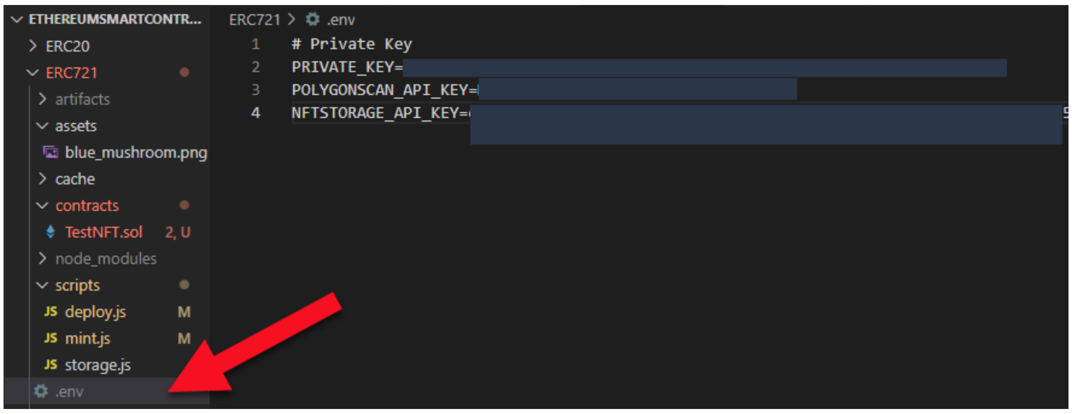
```
# Private Key
PRIVATE_KEY=[redacted]
POLYGONSCAN_API_KEY=[redacted]
NFTSTORAGE_API_KEY=[redacted]
```

3. Next, we will need a polygonscan api key so that we can run the verify process on our deployed contract to legitimize our claim on it on polygonscan. Using the API key to do it allows us to just run verify from hardhat which is much easier than doing it via the GUI.

4. Go to polygonscan and make an account then generate and API key for yourself to use.



Make your API key and add it to your .env file like this:

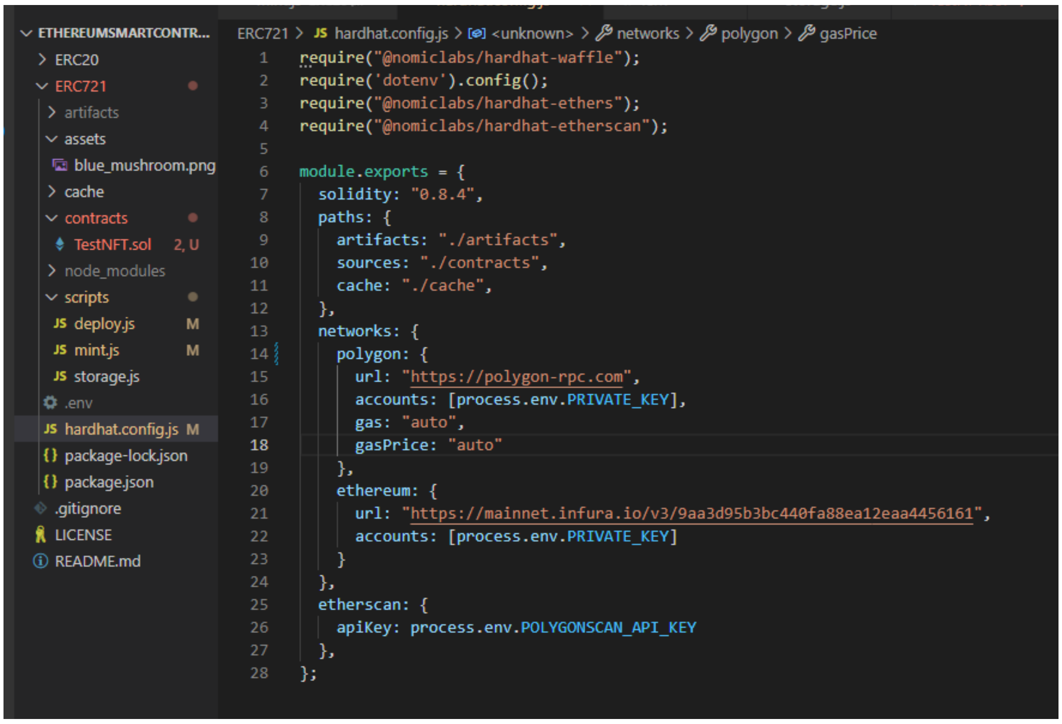


```
# Private Key
PRIVATE_KEY=[redacted]
POLYGONSCAN_API_KEY=[redacted]
NFTSTORAGE_API_KEY=[redacted]
```

5. Now our API keys are ready, we will next need to setup our hardhat config file so that it knows how to connect to the polygon network like this also install the hardhat-etherscan library into your project with:

   npm install @nomiclabs/hardhat-etherscan

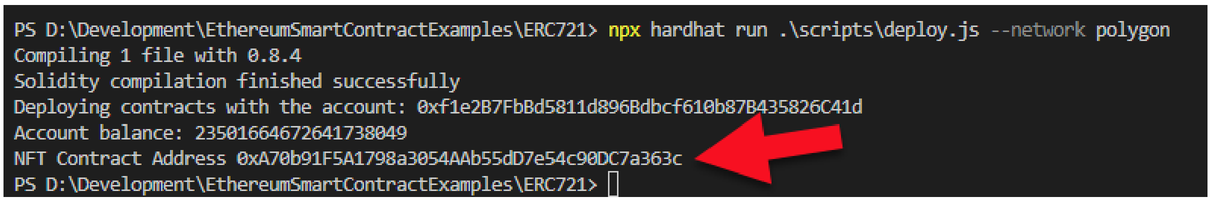
   Your hardhat.config.js should look like this:



```javascript
require("@nomiclabs/hardhat-waffle");
require('dotenv').config();
require("@nomiclabs/hardhat-ethers");
require("@nomiclabs/hardhat-etherscan");

module.exports = {
  solidity: "0.8.4",
  paths: {
    artifacts: "./artifacts",
    sources: "./contracts",
    cache: "./cache",
  },
  networks: {
    polygon: {
      url: "https://polygon-rpc.com",
      accounts: [process.env.PRIVATE_KEY],
      gas: "auto",
      gasPrice: "auto"
    },
    ethereum: {
      url: "https://mainnet.infura.io/v3/9aa3d95b3bc440fa88ea12eaa4456161",
      accounts: [process.env.PRIVATE_KEY]
    }
  },
  etherscan: {
    apiKey: process.env.POLYGONSCAN_API_KEY
  },
};
```

6. Alright we are now ready to run our deploy script to deploy our NFT contract to the polygon network. Using the console navigate to your project and execute the deploy.js script with the below command (the deploy script was explained earlier in this document refer to the relevant section if you need to):

   npx hardhat run .\scripts\deploy.js --network polygon



```
PS D:\Development\EthereumSmartContractExamples\ERC721> npx hardhat run .\scripts\deploy.js --network polygon
Compiling 1 file with 0.8.4
Solidity compilation finished successfully
Deploying contracts with the account: 0xf1e2B7FbBd5811d896Bdbcf610b87B435826C41d
Account balance: 23501664672641738049
NFT Contract Address 0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c
PS D:\Development\EthereumSmartContractExamples\ERC721>
```

7. Congratulations you have deployed your NFT contract to the polygon network. Our deploy script was written to output the contracts address after deployment so that we can use it later, make sure you copy that. For this document it is, if you follow this link you will see this one has already been deployed when this document was written:

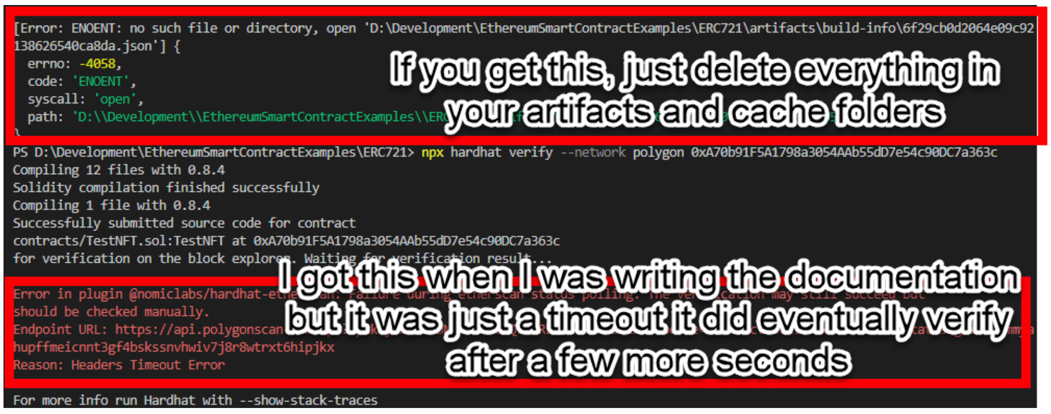
   0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c

8. Next, we will need to verify the contract on polygonscan we can do this easily now that we have setup the API key for polygonscan. This is an important step, don't skip it, it will allow us to do things like add a logo to the contract and links and descriptions to the project.

9. Run the below command in your console to execute the verify:

   npx hardhat verify –network polygon [your contracts address on polygon]

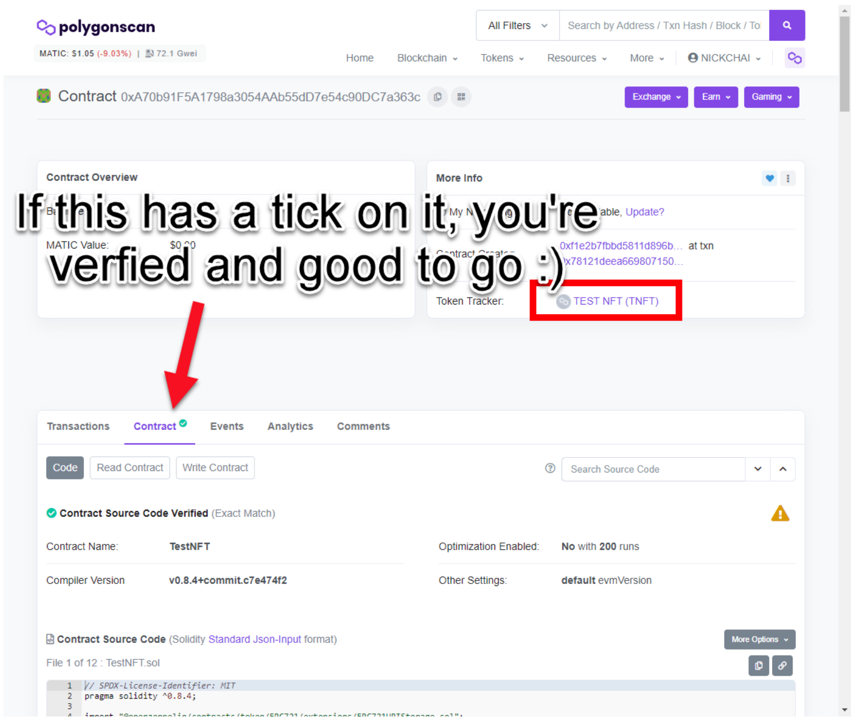
   (you may need to delete the files inside you artifacts and cache directories before you run this command if you are getting errors)



```
[Error: ENOENT: no such file or directory, open 'D:\Development\EthereumSmartContractExamples\ERC721\artifacts\build-info\6f29cb0d2064e09c92
138626540ca8da.json'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'D:\\Development\\EthereumSmartContractExamples\\ER
}
PS D:\Development\EthereumSmartContractExamples\ERC721> npx hardhat verify --network polygon 0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c
Compiling 12 files with 0.8.4
Solidity compilation finished successfully
Compiling 1 file with 0.8.4
Successfully submitted source code for contract
contracts/TestNFT.sol:TestNFT at 0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c
for verification on the block explorer. Waiting for verification result...

Error in plugin @nomiclabs/hardhat-ethe
should be checked manually.
Endpoint URL: https://api.polygonscan
hupffmeicnnt3gf4bskssnvhwiv7j8r8wtrxt6hipjkx
Reason: Headers Timeout Error

For more info run Hardhat with --show-stack-traces
```

If you get this, just delete everything in your artifacts and cache folders
I got this when I was writing the documentation but it was just a timeout it did eventually verify after a few more seconds

10. After executing the verify open your browser, go to polygonscan and find your contract by searching for it via it's address:



Contract 0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c

Contract Overview

MATIC Value: $0.00

More Info

Contract Creator: 0xf1e2b7fbbd5811d896b... at txn 0x78121deea669807150...

Token Tracker: TEST NFT (TNFT)

Contract Source Code Verified (Exact Match)

Contract Name: TestNFT

Optimization Enabled: No with 200 runs

Compiler Version v0.8.4+commit.c7e474f2

Other Settings: default evmVersion

Contract Source Code (Solidity Standard Json-Input format)

File 1 of 12 : TestNFT.sol

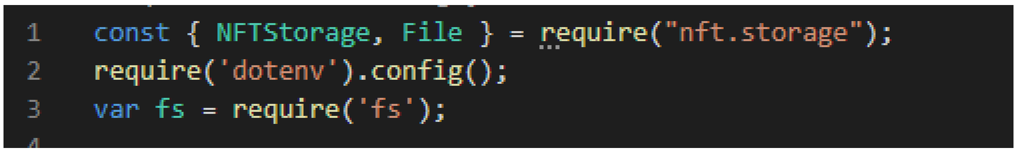
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;
```

Alright, contract is deployed, verified and good to go. It is usually a good idea to wait a little while for it to get a couple more confirmations on the blockchain before proceeding with using it to mint tokens.

11. Next, we will write a script that will first store an image into IPFS and get a URI for the stored data and then we will use that URI to mint a new NFT using our newly deployed contract.

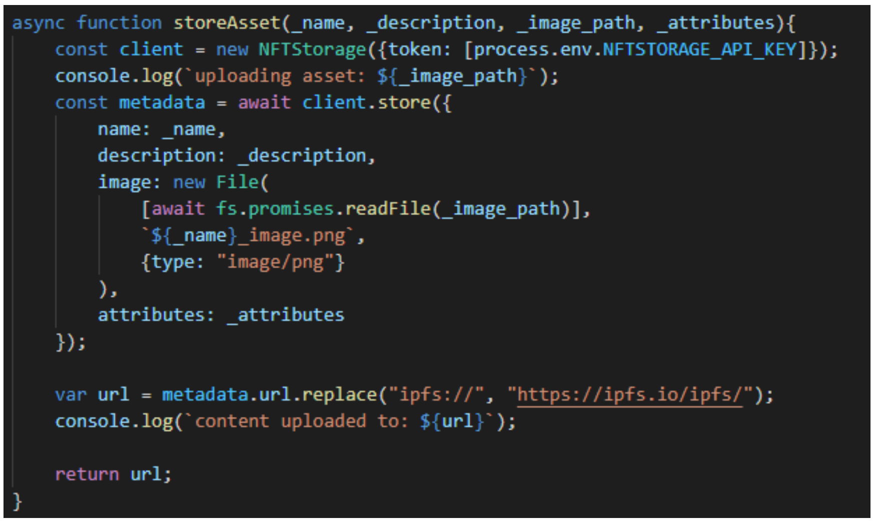
    Create a new file in the scripts directory called "mint-with-storage.js"

12. Inside this file define the libraries that we will need to store and asset via IPFS like this:



```
const { NFTStorage, File } = require("nft.storage");
require('dotenv').config();
var fs = require('fs');
```

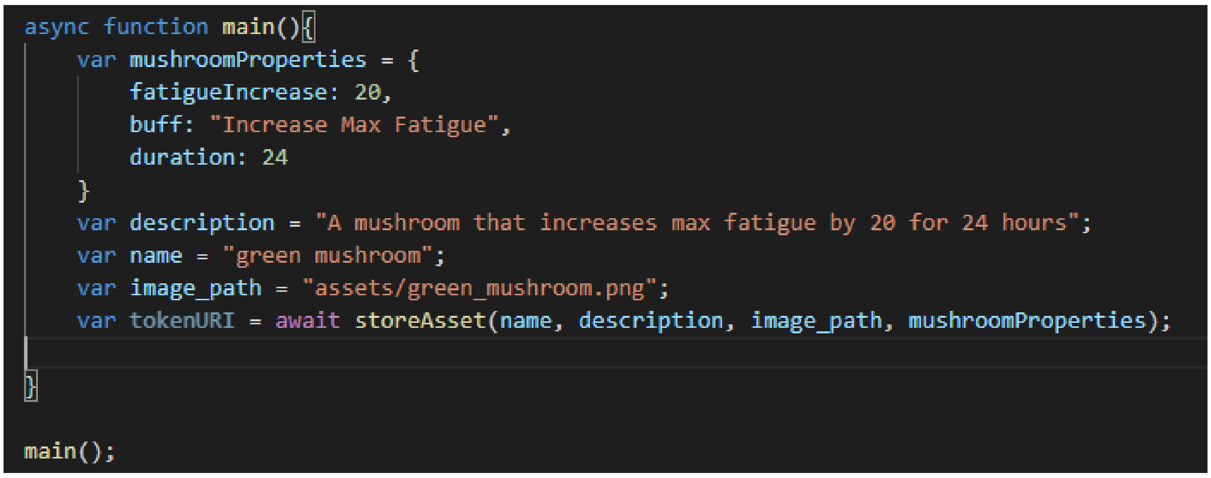
Next, define the storeAsset function the same way we had written it before in the previous section except this time, add a "return url" to the bottom of the script so that we can get the tokenURI value for the mint step.



```
async function storeAsset(_name, _description, _image_path, _attributes){
    const client = new NFTStorage({token: [process.env.NFTSTORAGE_API_KEY]});
    console.log(`uploading asset: ${_image_path}`);
    const metadata = await client.store({
        name: _name,
        description: _description,
        image: new File(
            [await fs.promises.readFile(_image_path)],
            `${_name}_image.png`,
            {type: "image/png"}
        ),
        attributes: _attributes
    });

    var url = metadata.url.replace("ipfs://", "https://ipfs.io/ipfs/");
    console.log(`content uploaded to: ${url}`);

    return url;
}
```

Also change the "properties" key to "attributes" as this is what opensea will expect.

13. Next create a main() function and setup the attributes and params to call the storeAsset() function like this:

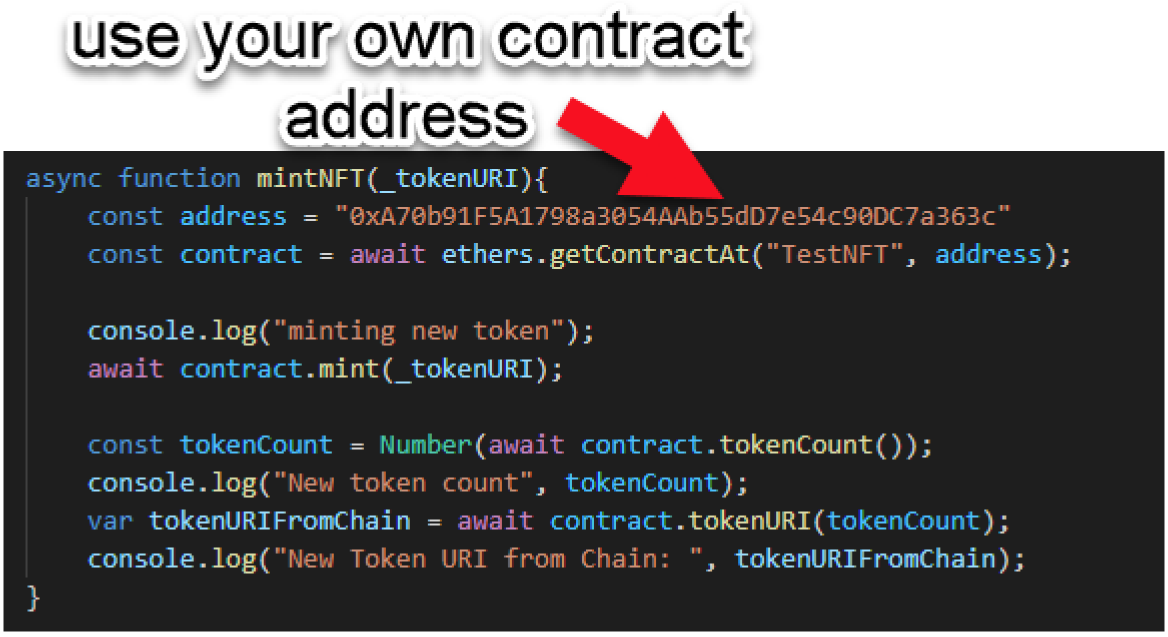


```
async function main(){
    var mushroomProperties = {
        fatigueIncrease: 20,
        buff: "Increase Max Fatigue",
        duration: 24
    }
    var description = "A mushroom that increases max fatigue by 20 for 24 hours";
    var name = "green mushroom";
    var image_path = "assets/green_mushroom.png";
    var tokenURI = await storeAsset(name, description, image_path, mushroomProperties);

}

main();
```

Make sure to also create the "main();" line on the bottom so it' runs the main function when we execute.

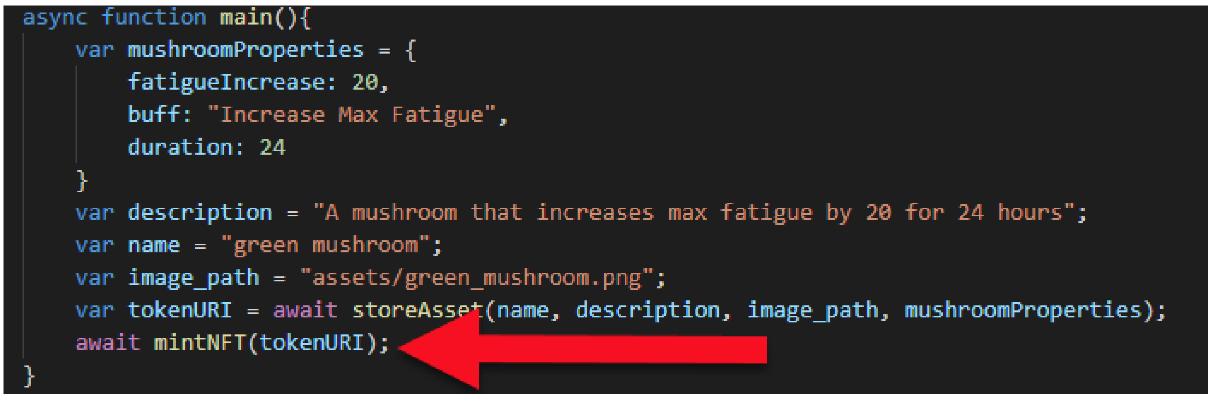
14. We've now created the code to create the NFT metadata and return a URI we will now use that URI to execute the "mint" function on our new contract to create an NFT. Create a new function called "mintNFT()" and put in the following code, similar as our "mint.js" file created previously but with less reading.

    Make sure you use your own contract address, otherwise you'll just be making an NFT from this documentation's contract.



```
async function mintNFT(_tokenURI){
    const address = "0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c"
    const contract = await ethers.getContractAt("TestNFT", address);

    console.log("minting new token");
    await contract.mint(_tokenURI);

    const tokenCount = Number(await contract.tokenCount());
    console.log("New token count", tokenCount);
    var tokenURIFromChain = await contract.tokenURI(tokenCount);
    console.log("New Token URI from Chain: ", tokenURIFromChain);
}
```

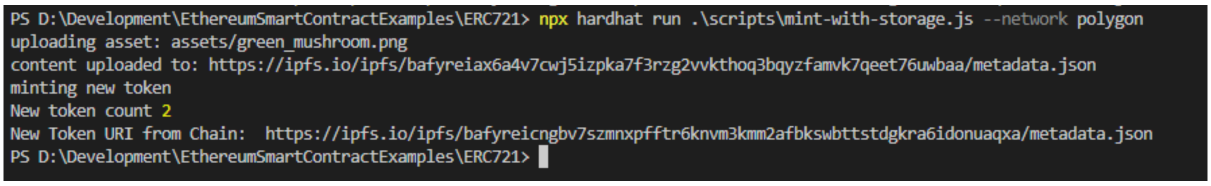
15. Next we will add the mintNFT() function call to our main and pass in the tokenURI we generate from the storeAsset() function:



```
async function main(){
    var mushroomProperties = {
        fatigueIncrease: 20,
        buff: "Increase Max Fatigue",
        duration: 24
    }
    var description = "A mushroom that increases max fatigue by 20 for 24 hours";
    var name = "green mushroom";
    var image_path = "assets/green_mushroom.png";
    var tokenURI = await storeAsset(name, description, image_path, mushroomProperties);
    await mintNFT(tokenURI);
}
```

16. Awesome, the code is in place now, all there is to do is run the mint-with-storage.js file with the below command:

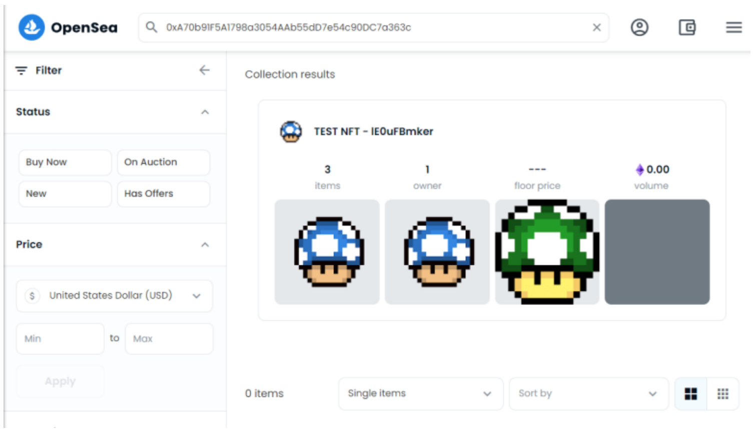
    npx hardhat run .\scripts\mint-with-storage.js –network polygon



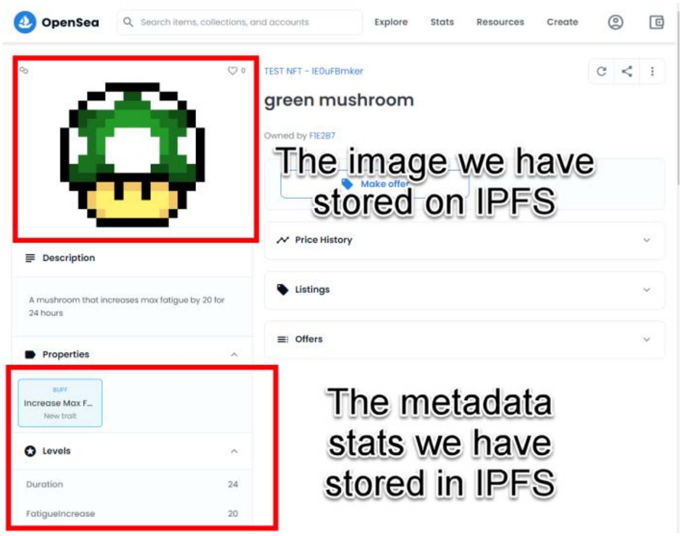
```
PS D:\Development\EthereumSmartContractExamples\ERC721> npx hardhat run .\scripts\mint-with-storage.js --network polygon
uploading asset: assets/green_mushroom.png
content uploaded to: https://ipfs.io/ipfs/bafyreiax6a4v7cwj5izpka7f3rzg2vvkthoq3bqyzfamvk7qeet76uwbaa/metadata.json
minting new token
New token count 2
New Token URI from Chain:  https://ipfs.io/ipfs/bafyreicngbv7szmnxpfftr6knvm3kmm2afbkswbttstdgkra6idonuaqxa/metadata.json
PS D:\Development\EthereumSmartContractExamples\ERC721> 
```

17. Great work! You have successfully deployed a new NFT contract onto polygon, created a metadata storage for a new token on IPFS and then minted the new token using the IPFS URI for the token metadata. Now it's time to have a look at your token on opensea.

18. Go to https://opensea.io and search for your contract's address for this documentation the contract address is: [0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c](0xA70b91F5A1798a3054AAb55dD7e54c90DC7a363c)



There will be a few NFTs on that contract already as they were created while writing this document.



4 Appendix

4.1 Gas transaction costs for deployments, minting and transfers

Transaction Type	Network	Currency	Cost (USD)	Cost (Matic)	Transaction URL	Description
Deploying ERC20 Token Contract	Polygon	MATIC	$0.22	0.1722	Polygon Transaction Hash (Txhash) Details | PolygonScan	deployed simple contract check git
Minting additional ERC20 Tokens	Polygon	MATIC	$0.01	0.0051	Polygon Transaction Hash (Txhash) Details | PolygonScan	minted 100 tokens
Transferring ERC20 Tokens	Polygon	MATIC	$0.02	0.0149	Polygon Transaction Hash (Txhash) Details | PolygonScan	transferred 10 tokens
Deploying ERC721 NFT Contract	Polygon	MATIC	$0.14	0.1368	Polygon Transaction Hash (Txhash) Details | PolygonScan	deployed an ownable URIStorage NFT contract
Minting new ERC721 NFT	Polygon	MATIC	$0.01	0.0083	Polygon Transaction Hash (Txhash) Details | PolygonScan	minting a new token using deployed contract with IPFS storage
Transferring ownership of NFT	Polygon	MATIC	$0.00	0.0036	Polygon Transaction Hash (Txhash) Details | PolygonScan	transfer an existing NFT from one wallet to another