

Building on Exx

[See live version](#)

EXX is the finest platform to build your decentralized Applications. EXX Network is the architecture for building web3 solutions aimed at speeding up and accelerating global web3 adoption. Start building on EXX and together, let's eliminate the blockers of adoption, and help more users delve into the world of web3 without fear or distrust.

This documentation is the guide that you need to explore the EXX blockchain. In it, you will find useful resources, links, tutorials and instructions that will help you navigate the network.

Developer Starter

Using Exx is easy. If you're an Ethereum Developer, you can develop with Exx. All the tools that you are familiar with are supported on the blockchain. Name them; Truffle, Remix, Web3js, Ethersjs etc.

All Exx Test network related details can be found in the [network docs](#)

- Setup [Metamask Wallet](#)
- Deploy your Contracts on Exx Network
 - [Using Remix](#)
 - [Using Truffle](#)
 - [Using Hardhat](#)
- Connecting to Exx [with RPC](#) by adding Exx Network on Metamask.

Already have a dApp?

- Migrate from Ethereum chain or any EVM compatible chain. Deploy all your smart contracts directly on the Exx Network. You don't have to worry about the underlying architecture, as long as it is EVM compatible!
- [Deploying your dApp on Exx](#)

Building a new dApp on Exx?

Start building!

- [Full Stack DApp: Tutorial Series](#)
- Getting to know your tools:
 - [Web3js](#), [Remix](#), [Truffle](#), [Metamask](#)
- [Writing your first DApp on Exx!](#) (updating - feranmi)

Learn the developer tools

- [CryptoZombies](#)
- [Full stack dapp tutorial series](#)
- [Infura Docs](#)
- [Truffle Suite Docs](#) (Recommended)
- [Truffle tutorials](#) (Recommended)
- [Parity Wiki](#)
- [Geth docs](#)
- [Remix](#)
- [OpenZeppelin Docs](#)
- [Ethernaut](#)
 - A game that teaches security
- [Capture the Ether](#)
4,8.
9 8 game that teaches security

Learn the Basics of Development

- [Full stack dapp tutorial series](#)
- [Truffle tutorials](#)
- [Dapp University](#)
- [ConsenSys Academy Developer Program On-Demand course](#)
- [What is Ethereum?](#)
- [Read Mastering Ethereum](#)
- [OpenZeppelin Learn Docs](#)

Getting Started

Connect to EXX Testnet on Metamask

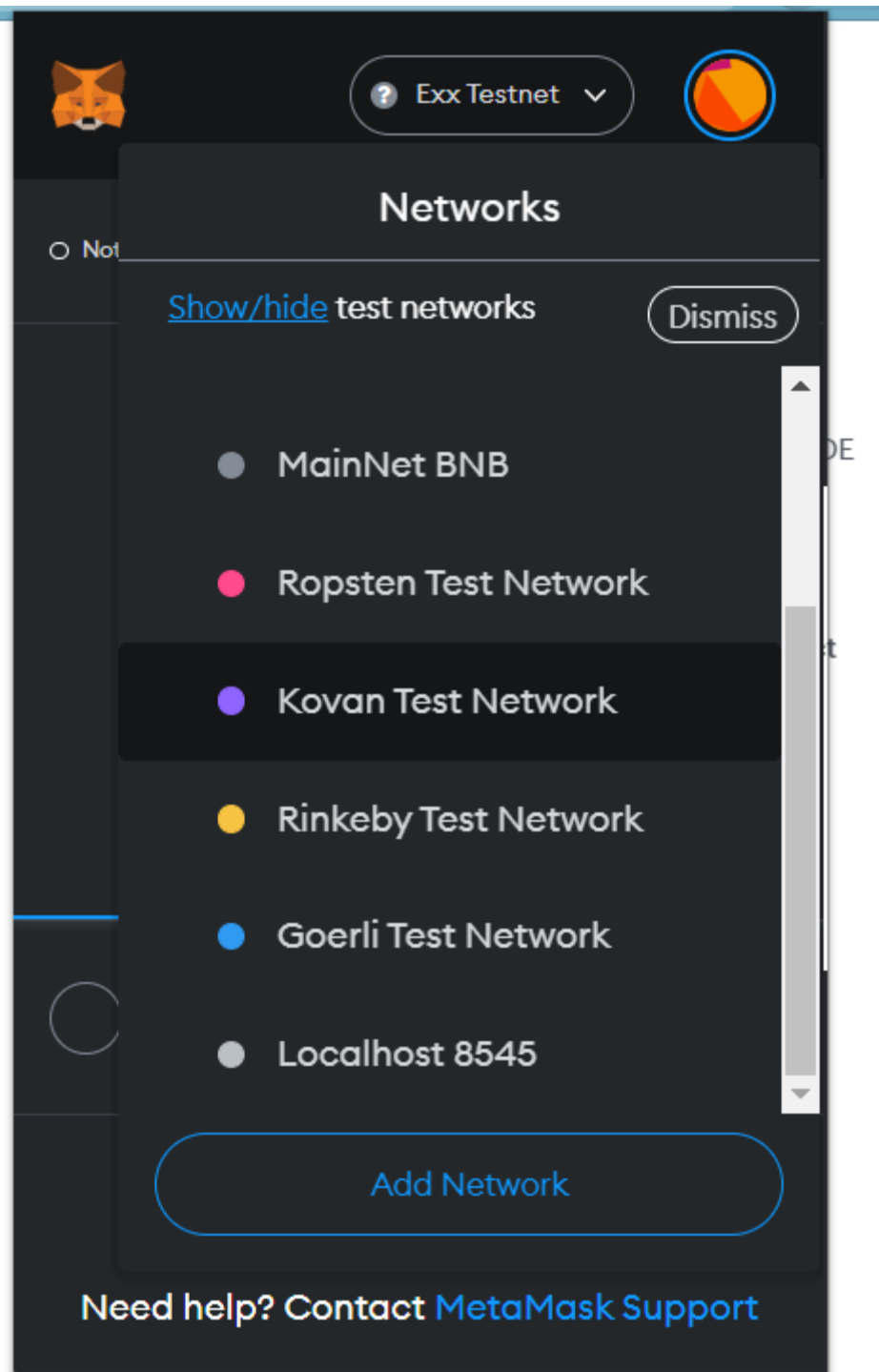
In order to view and operate on the EXX testnet, you will need to add our RPC and other credentials to your Metamask.

You can either add EXX network automatically on [EXXscan](#) or add manually.

In this tutorial, we will cover how to add manually.

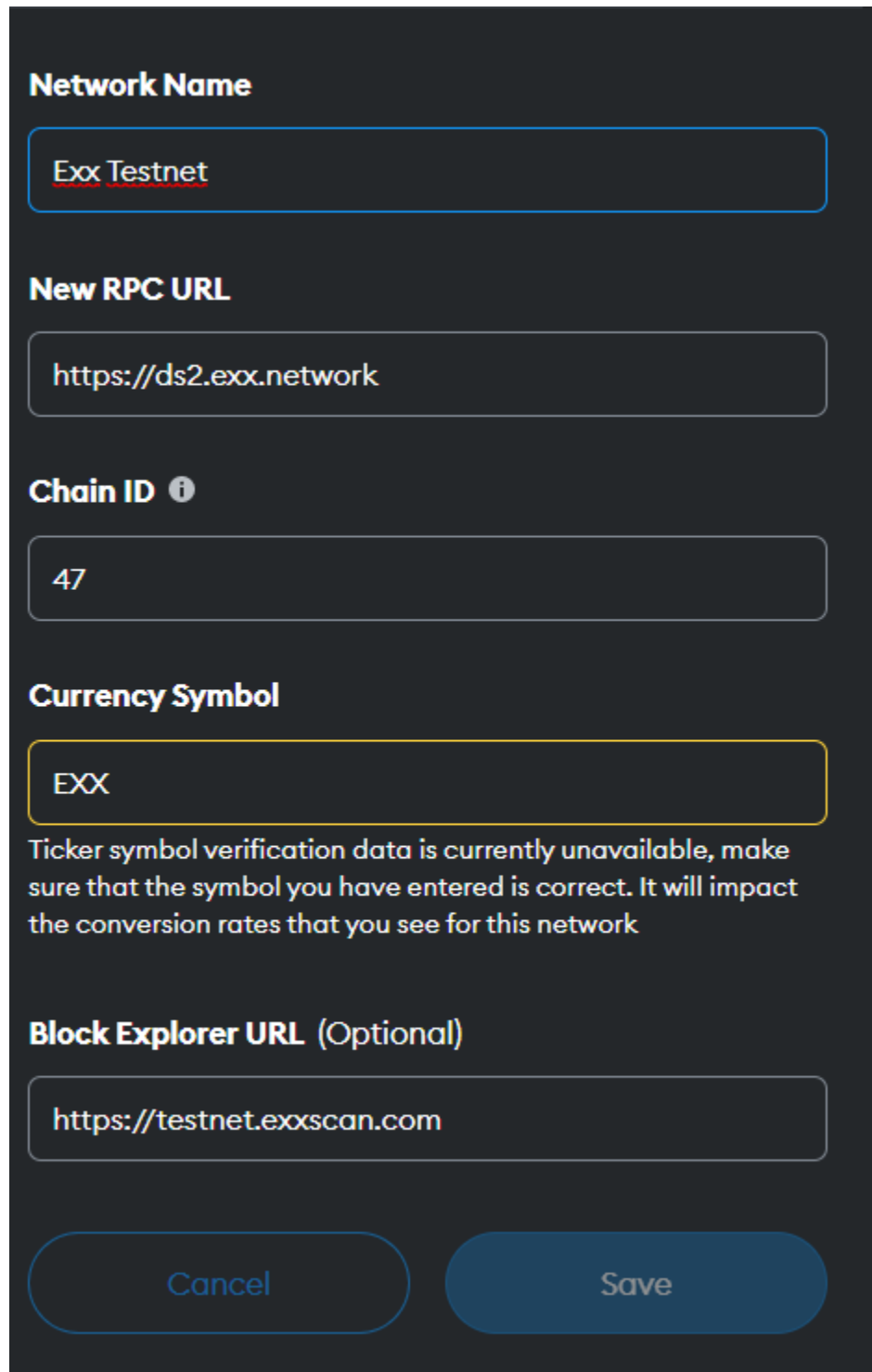
How to Add Manually

1. Open Metamask extension or the app on mobile.
2. Click on the Networks, then click on Add network or Custom RPC whatever your version shows.



3. Enter EXX Testnet as the network name
4. Enter <https://ds2.exx.network> for the RPC
5. Enter 47 for the chain ID
6. Enter EXX for the currency symbol

7. Enter <https://exxscan.com> for the Block explorer then save.



The screenshot shows a dark-themed configuration form for adding a new network. It contains several input fields and buttons. The 'Network Name' field is filled with 'Exx Testnet'. The 'New RPC URL' field is filled with 'https://ds2.exx.network'. The 'Chain ID' field is filled with '47'. The 'Currency Symbol' field is filled with 'EXX'. Below this field is a warning message. The 'Block Explorer URL (Optional)' field is filled with 'https://testnet.exxscan.com'. At the bottom are 'Cancel' and 'Save' buttons.

Network Name

Exx Testnet

New RPC URL

https://ds2.exx.network

Chain ID ⓘ

47

Currency Symbol

EXX

Ticker symbol verification data is currently unavailable, make sure that the symbol you have entered is correct. It will impact the conversion rates that you see for this network

Block Explorer URL (Optional)

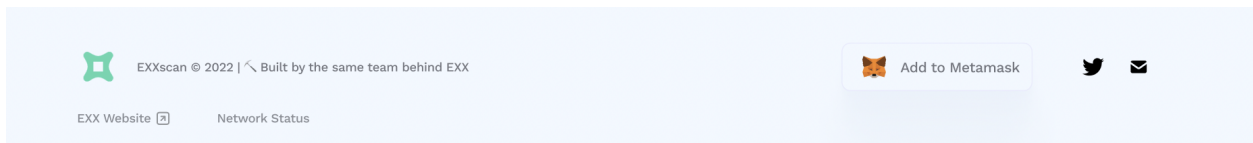
https://testnet.exxscan.com

Cancel Save

8. Save and start using EXX.

How to add on EXXscan

1. Proceed to the explorer <https://exxscan.com>
2. Scroll to the bottom and find the button showing Add to Metamask.



3. Click on Add to Metamask and then you will now be connected to the EXX testnet on your metamask wallet.

Using Remix

Overview

Using Truffle

Technical requirements of truffle requires that before you proceed, you need to install;

- [Node.js v8+ LTS and npm](#) (comes with Node)
- [Git](#)

Once installed, here is the command to install truffle

```
npm install -g truffle
```

To confirm that Truffle is properly installed, type the truffle version on a terminal. Should there be an error, see that your npm modules are added to your path.

New to Truffle? follow the [Getting Started](#) by truffle to set up the truffle environment.

Truffle-config

- Go to truffle-config.js
- Update the truffle-config with Exx-network-credentials.

N:B: it requires mnemonic to be passed in for Exx Provider. Mnemonic is the 12 word seed phrase for the account you'd like to deploy from. Create a new .secret file in the root directory and enter your 12 word mnemonic seed phrase to get started.

To get your seedwords from metamask wallet; go to Metamask Settings, from the menu click Security and Privacy, there, you will see a button that says reveal seed words.

Deploying on Exx Network

Run this command in root of the project directory:

```
$ truffle migrate --network exx
```

Contract will be deployed on Exx Testnet, it will look like this:

```
2_deploy_contracts.js
```

```
=====
```

```
Replacing 'MyContract'
```

```
-----
```

```
> transaction hash:
```

```
0x1c94d095a2f629521344885910e6a01076188fa815a310765679b05abc09a250
```



```
> Blocks: 5          Seconds: 5
> contract address:  0xbFa33D565Fcb81a9CE8e7a35B61b12B04220A8EB
> block number:      2371252
> block timestamp:    1578238698
> account:           0x9fB29AAc15b9A4B7F17c3385939b007540f4d791
> balance:           79.409358061899298312
> gas used:          1896986
> gas price:         0 gwei
> value sent:        0 ETH
> total cost:        0 ETH
```

Pausing for 2 confirmations...

> confirmation number: 5 (block: 2371262)
initialised!

```
> Saving migration to chain.
> Saving artifacts
```

> Total cost: 0 ETH

Summary

=====

```
> Total deployments:  2
> Final cost:         0 ETH
```

Remember your address, transaction_hash and other details provided would differ, Above is just to provide an idea of structure.

Congratulations! You have successfully deployed HelloWorld Smart Contract. Now you can interact with the Smart Contract.

You can check the deployment status here: <https://mumbai-explorer.matic.today/>

Using HardHat

Setting up the development environment

Technical requirements of HardHat requires that before you proceed, you need to install;

- Node.js v8+ LTS and npm (comes with Node)
- Git

Once installed, installing hardhat requires you to create an npm project by going to an empty folder, running npm in it, and following its instructions.

Once your project is ready, you should run

```
$ npm install --save-dev hardhat
```

Create a sample project to run npx hardhat in your project folder. Go through these steps to try out the sample task, compile test, and deploy the sample contract.

The sample project will request you to install hardhat-waffle and hardhat-ethers. You can learn more about it in this guide

hardhat-config#

- Go to hardhat.config.js
- Update the hardhat-config with matic-network-credentials.
- create .secret file in the root to store your private key

```
const fs = require('fs');
const privatekey = fs.readFileSync(".secret").toString().trim();
module.exports = {
  defaultNetwork: "matic",
  networks: {
    hardhat: {
    },
    matic: {
      url: "https://rpc-mumbai.maticvigil.com",
      accounts: [privateKey]
    }
  },
  solidity: {
```

```
version: "0.7.0",
settings: {
  optimizer: {
    enabled: true,
    runs: 200
  }
},
paths: {
  sources: "./contracts",
  tests: "./test",
  cache: "./cache",
  artifacts: "./artifacts"
},
mocha: {
  timeout: 20000
}
}
```

Deploying on Exx Network#

Run this command in root of the project directory:

```
$ npx hardhat run scripts/sample-script.js --network matic
```

Contract will be deployed on Exx Testnet, it look like this:

Compilation finished successfully

Greeter deployed to: 0xfafCAD549BAA6110c5Cc03976d9383AcE90bdBE

Remember your address would differ, Above is just to help with an idea of structure.

Congratulations! You have successfully deployed Greeter Smart Contract. Now you can interact with the Smart Contract.

You can check the deployment status here: <https://mumbai-explorer.matic.today/>

Exx Testnet

Exx Testnet is the test version of the Exx mainnet to be released. This is done to let developers and validators to experiment without having to use real assets.

Testnet coins are separate, are distinct from actual tokens/assets, and do not have any value whatsoever.

N:B: This documentation contains details for the RPC - HTTP, WS and Dagger endpoints. There is provision for setting up your personal full node, if that is what you prefer.

Network name	EXX Testnet
Chain ID	47
Gas token	EXX (testnet)
RPC	https://ds2.exx.network
Websocket	Coming soon
Block Explorer	https://testnet.exxscan.com

IMPORTANT

EXX network native token is EXX which will be used as gas fee.

Creating a MetaMask Wallet

Metamask is a free web3 browser extension that lets applications to read and interact with EVM compatible blockchains such as EXX.

To create a wallet, you need to install a metamask extension which can be found for any browser of your preference.

For this tutorial, we will be using the google chrome as an example

Adding Metamask Extension

1. Visit <https://metamask.io> or search for metamask extension using any browser of your choice. (Let us use chrome for the purpose of illustration).
2. Click install MetaMask as a chrome extension.
3. Click Add to Chrome.
4. Click Add Extension.

You have successfully installed metamask extension

Next step is to create an account

Step 2. Create an account

The next step is to create an account.

1. Click on the MetaMask icon in the upper right corner to open the extension.
2. To install the latest version of MetaMask, click Try it now.
3. Click Continue.
4. Create a strong password and click Create. ***Ensure you store your password somewhere safe***
5. Proceed by clicking Next, then accept Terms of Use.
6. Click Reveal secret words.
7. You will see a 12 word seed phrase. ***Save seed words as a file or copy them to a safe place and click Next.***

Reveal secret words and copy your secret backup phrase to a safe place

Random security tips: Write this phrase on a piece of paper and store it in a secure location. If you want even more security, write it down on multiple pieces of paper and store each in 2–3 different locations.

8. Verify your secret phrase by selecting the previously generated phrase. When done, click Confirm.

By “solving this puzzle” you are confirming that you know your secret phrase

Congratulations! You have successfully created your MetaMask account. A new Ethereum wallet address was automatically generated for you!

Configuring Exx on Metamask

This tutorial contains how you can add custom XRC20 tokens to any network on Metamask.

Switch the network on Metamask to point to the [EXX Testnet](#). (link to network connection)

On Metamask, this will be shown as EXX Testnet or whatever you have named it when adding.

Configuring Exx TST tokens to Metamask

Adding a Test token (XRC20) to your Metamask account on EXX testnet. You must have gotten the test tokens before you can see any amount in your wallet. You can claim from the faucet.

To display your tokens on your account on the Exx Network, you can click on the Add Tokens option in Metamask. It will take you to a screen.

Click on the Custom Token tab and copy-paste the address below in the Token Address field.

The Test token contract address is [0x3f152B63Ec5CA5831061B2DccFb29a874C317502](#).

N:B: The TEST token is an example XRC20 token contract that is used throughout the Exx Network developer docs for the purpose of illustration.

The other fields will auto-populate.

Click on Save and then click on Add Tokens.

The TEST token should now be displayed on your account on Metamask.

Assets

EXX Native token

EXX is the native token of EXX network, similar to ETH in Ethereum. EXX is used to interact with EXX Testnet and pay gas fees. XRC20 is the standard token type on the EXX network just as ERC20 is standard for the ethereum blockchain.

EVM Native Asset Call

An EVM Transaction is composed of the following :

- nonce Scalar value equal to the number of transactions sent by the sender.
- gasPrice Scalar value equal to the number of Wei (1 Wei = 10^{-18} AVAX) paid per unit of gas to execute this transaction.
- gasLimit Scalar value equal to the maximum amount of gas that should be used in executing this transaction.
- to The 20 byte address of the message call's recipient. If the transaction is creating a contract, to is left empty.
- value Scalar value of native asset (AVAX), in Wei (1 Wei = 10^{-18} AVAX), to be transferred to the message call's recipient or in the case of a contract creation, as an endowment to the newly created contract.
- v, r, s Values corresponding to the signature of the transaction.
- data Unlimited size byte array specifying the input data to a contract call or, if creating a contract, the EVM bytecode for the account initialization process.

EXX Testnet token

In order to interact EXX Testnet, you need to have the EXX native token. On testnet, it is free to claim these tokens at specified intervals.

To claim tokens: use the [faucet](#)

How to use the faucet

1. Connect to the faucet via <https://faucet.exx.network>

Peers	Blocks	Exx's	Funded
1	949656	19	38

2. Enter the wallet address you want faucet to send EXX to.
3. Click on Submit.
4. Click on Transfer to receive the tokens.

XRC20: Tokens on EXX

1. Summary

The XRC20 interface as proposed and implemented through the [EIP20](#)

2. Abstract

The following standard defines the implementation of APIs for token smart contracts. It is proposed by deriving the ERC20 protocol of Ethereum and provides the basic functionality to transfer tokens and allow tokens to be approved so they can be spent by another on-chain third party.

3. Motivation

A standard interface allows any tokens on EXX to be used by other applications including wallets, decentralized apps, decentralized exchanges etc. in a consistent way.

4. Specification

4.1 Token

NOTES:

- The following specifications use syntax from Solidity 0.5.16 (or above)
- Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned!

4.1.1 Methods

name

function name() public view returns (string)

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

symbol

function symbol() public view returns (string)

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

decimals

function decimals() public view returns (uint8)

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

totalSupply

function totalSupply() public view returns (uint256)

- Returns the total token supply.

balanceOf

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

- Returns the account balance of another account with address _owner.

getOwner

function getOwner() external view returns (address);

- Returns the xrc20 token owner.

transfer

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

- Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend.
- NOTE - Except transfer of 0 value is disabled in a contract, Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.

transferFrom

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

- Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event.
- The transferFrom method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.
- NOTE - Except transfer of 0 value is disabled in a contract, Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.

approve

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

- Allows _spender to withdraw from your account multiple times, up to the _value amount. If this function is called again it overwrites the current allowance with _value.
- NOTE - To prevent attack vectors like the one described [here](#), clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. **THOUGH** The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before.

allowance

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

- Returns the amount which _spender is still allowed to withdraw from _owner.

Events

Transfer

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

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

Approval

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

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

Wallets

Decentralized Storages

IPFS

One of the most appreciated features of web3 networks is that they store data. Exx can store data effectively but would cost too much

InterPlanetary File System is a distributed system for storing and accessing files, websites, applications, and data.

Using IPFS, you don't need to store entire files on EXX. You are only storing the hash of the IPFS on the Exx Network.

Filecoin

It is a web3 digital storage and data retrieval technique built on top of IPFS and supports storing data long-term via on-chain deals.

Storage Helpers (IPFS + Filecoin)

- [Estuary](#): Pinning service that stores and retrieves data on both IPFS and Filecoin networks via simple API calls ([video](#)).
- [NFT.storage](#): NFT storage service that stores and retrieves data relating to NFTs on IPFS and Filecoin ([video](#)).
- [Web3.storage](#): Data storage service that stores and retrieves data on IPFS and Filecoin ([video](#))
- [Textile Powergate](#): Highly configurable wrapper for IPFS+Filecoin inside a Docker container.
- [Fleek Space Daemon and Space SDK](#): For decentralized browser, mobile, or desktop development.

Mint NFTs

This tutorial will teach you to mint an NFT using the Exx blockchain and IPFS/Filecoin storage via NFT.Storage.

The tutorial will walk you through

- how to create and deploy a standardized smart contract,
- storing metadata and assets on IPFS, and Filecoin via the NFT.Storage API. and
- minting the NFT to your own wallet on Exx

Introduction

In this tutorial we aim to;

1. fulfill three unique features with our minting process:
 - *Scalability* of the minting process in terms of cost and throughput. Web3 needs to handle all minting requests and make minting cheap.
 - *NFT Durability* : assets can be long-lived, they must remain usable during their entire life.
 - *NFT Immutability* and the asset it represents to prevent unwanted exploration by malicious hackers, Thieves and Attacks.
2. Show you an overview of the NFT minting process, learn how to store a digital asset with NFT.Storage, as well as how to use this digital asset to mint your NFT on Exx.

Prerequisites

General knowledge about NFTs will give you background and context.

To test and run the code in this tutorial, you will need a working [Node.js installation](#).

You'll also need an Exx wallet on the Exx Testnet with a small amount of the EFT.

1. Download and install [Metamask](#).
2. Connect Metamask to Exx testnet and select it in the dropdown menu. We will use Exx testnet to mint our NFT. Remember it is free of charge
3. Receive MATIC token to your wallet by using the faucet
4. Select Exx testnet, then paste your wallet address from Metamask into the form
To mint NFT, we need to pay a little amount which is charged by miners for operations to add new transactions to the network
5. Copy your private key from Metamask by clicking on the three dots in the top right corner. Selecting 'Account details'.
6. On the bottom you can find a button to export your private key. Click it and enter your password when you receive the prompt. (You can copy and paste the private key in a text file for now. We will use it later in the course of this tutorial when interacting with the network)
7. You will need a text or code editor. We recommend you choose an editor with language support for both JavaScript and Solidity.

Preparation

Get an API key for NFT.storage

You need an API key to use NFT.Storage.

1. Head over to [NFT.Storage](#) to log in with your email address.
2. You will receive an email with a magic link that signs you in -- no password needed.
3. Once you are logged in, go to API Keys via the navigation bar. There, You will find a button to create a New Key. When prompted for an API key name, you can freely choose one or use "polygon + NFT.Storage".

Set up your workspace

Create a new empty folder that we can use as our workspace for this tutorial. Give it any name and store it in any computer file location of your choice. Open up a terminal and navigate to the newly created folder.

We will install the following Node.js dependencies:

- Hardhat and Hardhat-Ethers, a development environment for Ethereum (and Ethereum compatible blockchains like Polygon).
- OpenZeppelin, a collection of smart contracts featuring standardized NFT base contracts.
- NFT.Storage, a library to connect to the NFT.Storage API.
- Dotenv, a library to handle environment files for configuration (e.g., injecting private keys into the script).

Use the following command to install all dependencies at once:

```
npm install hardhat @openzeppelin/contracts nft.storage dotenv @nomiclabs/hardhat-ethers
```

Hardhat needs to be initialized in the current folder. In order to start the initialization, execute:
`npx hardhat`

When prompted, choose Create an empty hardhat.config.js. Your console output should look like this:

```
✓ What do you want to do? · Create an empty hardhat.config.js  
✨ Config file created ✨
```

We will do some modifications to the hardhat configuration file hardhat.config.js to support the Polygon Mumbai test network. Open the hardhat.config.js that was created in the last step.

Please note that we are loading your Polygon wallet private key from an environment file and that this environment file must be kept safe. You can even use other rpc link, as per requirement.

```
/**
 * @type import('hardhat/config').HardhatUserConfig
 */
require("@nomiclabs/hardhat-ethers");
require('dotenv').config();
const { PRIVATE_KEY } = process.env;
module.exports = {
  defaultNetwork: "PolygonMumbai",
  networks: {
    hardhat: {
    },
    PolygonMumbai : {
      url: "https://rpc-mumbai.maticvigil.com",
      accounts: [PRIVATE_KEY]
    }
  },
  solidity: {
    version: "0.8.12",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  },
}
```

Create a new file called .env which will hold your API key for NFT.Storage and your Polygon wallet. The content of the .env file should look like the listing below:

```
PRIVATE_KEY="Your Private Key"
NFT_STORAGE_API_KEY="Your Api Key"
```

Replace the placeholders with the API key you created during preparation and your Polygon wallet private key.

To keep our project organized, we'll create three new folders:

1. contracts, for the Exx contracts written in Solidity.
2. assets, containing the digital asset we will mint as an NFT.

3. scripts, as helpers to drive the preparation and minting process.

Execute the following command:

```
mkdir contracts assets scripts
```

Finally, we will add an image to the assets folder. The image is our artwork that we will upload to NFT.Storage and mint on Exx. We will name it MySampleNFT.png for now.

Minting your NFT

Storing asset data with NFT.Storage

We will use NFT.Storage to store our NFT and its metadata.

Create a script called store-asset.mjs below the scripts directory. The contents are listed below:

```
import { NFTStorage, File } from "nft.storage"
import fs from 'fs'
import dotenv from 'dotenv'
dotenv.config()

const { NFT_STORAGE_API_KEY } = process.env

async function storeAsset() {
  const client = new NFTStorage({ token: NFT_STORAGE_API_KEY })
  const metadata = await client.store({
    name: 'ExampleNFT',
    description: 'My ExampleNFT is an awesome artwork!',
    image: new File(
      [await fs.promises.readFile('assets/MyExampleNFT.png')],
      'MyExampleNFT.png',
      { type: 'image/png' }
    ),
  })
  console.log("Metadata stored on Filecoin and IPFS with URL:", metadata.url)
}

storeAsset()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

The main part of the script is the `storeAsset` function. It creates a new client connecting to `NFT.Storage` using the API key you created earlier. Next we introduce the metadata consisting of name, description, and the image. Note that we are reading the NFT asset directly from the file system from the `assets` directory. At the end of the function we will print the metadata URL as we will use it later when creating the NFT on Polygon.

After setting up the script, you can execute it by running:
`node scripts/store-asset.mjs`

Your output should look like the listing below, where `HASH` is the CID to the art you just stored. Metadata stored on Filecoin/IPFS at URL: `ipfs://HASH/metadata.json`

Creating your NFT on Exx

Create the smart contract for minting

First, we will create a smart contract that will be used to mint the NFT. We will write the smart contract in [Solidity](#).

Create a new file for our NFT smart contract called `SampleNFT.sol` inside the `contracts` directory. You can copy the code of the listing below:

```
// Contract based on https://docs.openzeppelin.com/contracts/4.x/erc721
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;
```

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

```
contract ExampleNFT is ERC721URIStorage, Ownable {
    using Counters for Counters.Counter;
    Counters.Counter private _tokenIds;
```

```
    constructor() ERC721("NFT", "ENFT") {}
```

```
    function mintNFT(address recipient, string memory tokenURI)
        public onlyOwner
        returns (uint256)
    {
        _tokenIds.increment();

        uint256 newItemId = _tokenIds.current();
        _mint(recipient, newItemId);
        _setTokenURI(newItemId, tokenURI);
    }
}
```

```
        return newItemId;
    }
}
```

To be a valid NFT, your smart contract must implement all the methods of the ERC-721 standard.

We use the implementation of the OpenZeppelin library, which already provides a set of basic functionalities and adheres to the standard.

At the top of our smart contract, we import three OpenZeppelin smart contract classes:

`\@openzeppelin/contracts/token/ERC721/ERC721.sol` contains the implementation of the basic methods of the ERC-721 standard, which our NFT smart contract will inherit. We use the `ERC721URIStorage`, which is an extension to store not just the assets but also metadata as a JSON file off-chain. Like the contract, this JSON file needs to adhere to ERC-721.

`\@openzeppelin/contracts/utils/Counters.sol` provides counters that can only be incremented or decremented by one. Our smart contract uses a counter to keep track of the total number of NFTs minted and to set the unique ID on our new NFT.

`\@openzeppelin/contracts/access/Ownable.sol` sets up access control on our smart contract, so only the owner of the smart contract (you) can mint NFTs.

After our import statements, we have our custom NFT smart contract, which contains a counter, a constructor, and a method to actually mint the NFT. Most of the hard work is done by the base contract inherited from OpenZeppelin, which implements most of the methods we require to create an NFT adhering to the ERC-721 standard.

The counter keeps track of the total number of NFTs minted, which is used in the minting method as a unique identifier for the NFT.

In the constructor, we pass in two string arguments for the name of the smart contract and the symbol (represented in wallets). You can change them to whatever you like.

Finally, we have our method `mintNFT` that allows us to actually mint the NFT. The method is set to `onlyOwner` to make sure it can only be executed by the owner of the smart contract.

`address recipient` specifies the address that will receive the NFT at first

`string memory tokenURI` is a URL that should resolve to a JSON document that describes the NFT's metadata. In our case it's already stored on `NFT.Storage`. We can use the returned IPFS link to the metadata JSON file during the execution of the method.

Inside the method, we increment the counter to receive a new unique identifier for our NFT. Then we call the methods provided by the base contract from OpenZeppelin to mint the NFT to the recipient with the newly created identifier and setting the URI of the metadata. The method returns the unique identifier after execution.

Deploy the smart contract to Exx

Create a new file titled `deploy-contract.mjs` within the `scripts` directory.

Copy the contents of the listing below into that file and save it.

```
async function deployContract() {
  const ExampleNFT = await ethers.getContractFactory("ExampleNFT")
  const exampleNFT = await ExampleNFT.deploy()
  await exampleNFT.deployed()
  // This solves the bug in Mumbai network where the contract address is not the real one
  const txHash = exampleNFT.deployTransaction.hash
  const txReceipt = await ethers.provider.waitForTransaction(txHash)
  const contractAddress = txReceipt.contractAddress
  console.log("Contract deployed to address:", contractAddress)
}

deployContract()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

Deploying the contract is done with the helper functions provided by the hardhat library. First, we get the smart contract we created in the previous step with the provided factory. Then we deploy it by calling the respective method and wait for the deployment to be completed. There are a few more lines below the described code to get the correct address in the testnet environment. Save the `mjs` file. Execute the script with the following command:

```
npx hardhat run scripts/deploy-contract.mjs --network PolygonMumbai
```

If everything is correct, you will see the following output:

```
Contract deployed to address: 0x{YOUR_CONTRACT_ADDRESS}
```

Note that you will need the printed contract address in the minting step. You can copy and paste it into a separate text file and save it for later. This is necessary so the minting script can call the minting method of that specific contract.

Minting the NFT on Exx

Minting the NFT now means calling the contract we just deployed to Exx.

Create a new file called mint-nft.mjs inside the scripts directory and copy this code from the listing below:

```
const CONTRACT_ADDRESS = "0x00"
const META_DATA_URL = "ipfs://XX"

async function mintNFT(contractAddress, metaDataURL) {
  const ExampleNFT = await ethers.getContractFactory("ExampleNFT")
  const [owner] = await ethers.getSigners()
  await ExampleNFT.attach(contractAddress).mintNFT(owner.address, metaDataURL)
  console.log("NFT minted to: ", owner.address)
}

mintNFT(CONTRACT_ADDRESS, META_DATA_URL)
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

Edit the first two lines to insert your contract address from the earlier deployment and the metadata URL that was returned when storing the asset with NFT.Storage. The rest of the script sets up the call to your smart contract with you as the to-be owner of the NFT and the pointer to the metadata stored on IPFS.

Next, run the script:

```
npx hardhat run scripts/mint-nft.mjs --network PolygonMumbai
```

You can expect to see the following output:

```
NFT minted to: 0x{YOUR_WALLET_ADDRESS}
```

Looking for the sample code from this tutorial? You can find it in the [polygon-nft.storage-demo link Github repo](#).

Heads Up!

You can always define complex logic that governs your NFT life cycle.

For more complex use cases, the successor standard [ERC-1155](#) is a good place to start. OpenZeppelin, the library we use in our tutorial offers a [contracts wizard](#) that helps create NFT contracts.

Community

The EXX community is a home of developers, designers, validators, investors and interesting people all around the world trying to contribute to the growth and mainstream adoption of Web3.

Community Channels