By Jeff Delaney
Posted Jan 17, 2022

#web3 #solidity #react SOURCE CODE

Over the last few months, you've likely heard there term Web3 thrown around in the media. It generally refers to decentralized webapps that use Ethereum smart contracts to replace traditional web servers. Sounds pretty cool, let's build one!

The following tutorial demonstrates the entire process of building a smart contact, then interacting with it on the web using Ethers.js and React. The app can mint non-fungible tokens (NFTs) when a user transfers Ether from a wallet like MetaMask. There are many different technologies involved, but the core idea is to access the API of a smart contract from a frontend web app.

INTITIAL SETUP

Generate Art

To list an NFT collection, you'll need to first generate some art and some JSON metadata to go with it. There are many ways to go about it.

- My custom Art Generator Source Code
- Hashlips Art Engine
- No-code NFT Generator

To follow this tutorial, you **don't need to generate any art**. Feel free to use any random image or file.

Upload Art to IPFS

NFTs do not actually store images on the blockchain. Instead, they store a hash of the image. This hash is called the NFT's content ID (CID) and is typically hosted on IPFS. Once content is uploaded, it cannot be modified without changing the CID.

I would recommend using a tool like Pinata to simplify the process of uploading your art on IPFS.

Notice the unique Content ID in the IPFS folder

Setup Hardhat

Hardhat is a development toolchain that helps configure and deploy smart contracts. Get started by generating a React app (with Vite), then install the dependencies listed below.

command line

```
npm init vite myapp
cd myapp

npx hardhat
npm install --save-dev @nomiclabs/hardhat-waffle ethereum-waffle chai @nomiclabs/hardhat-ethers ethers @openzeppelin/contracts

npm run dev
```

In the Hardhat config, update the compilation path for artifacts so they can be easily recognized by React.

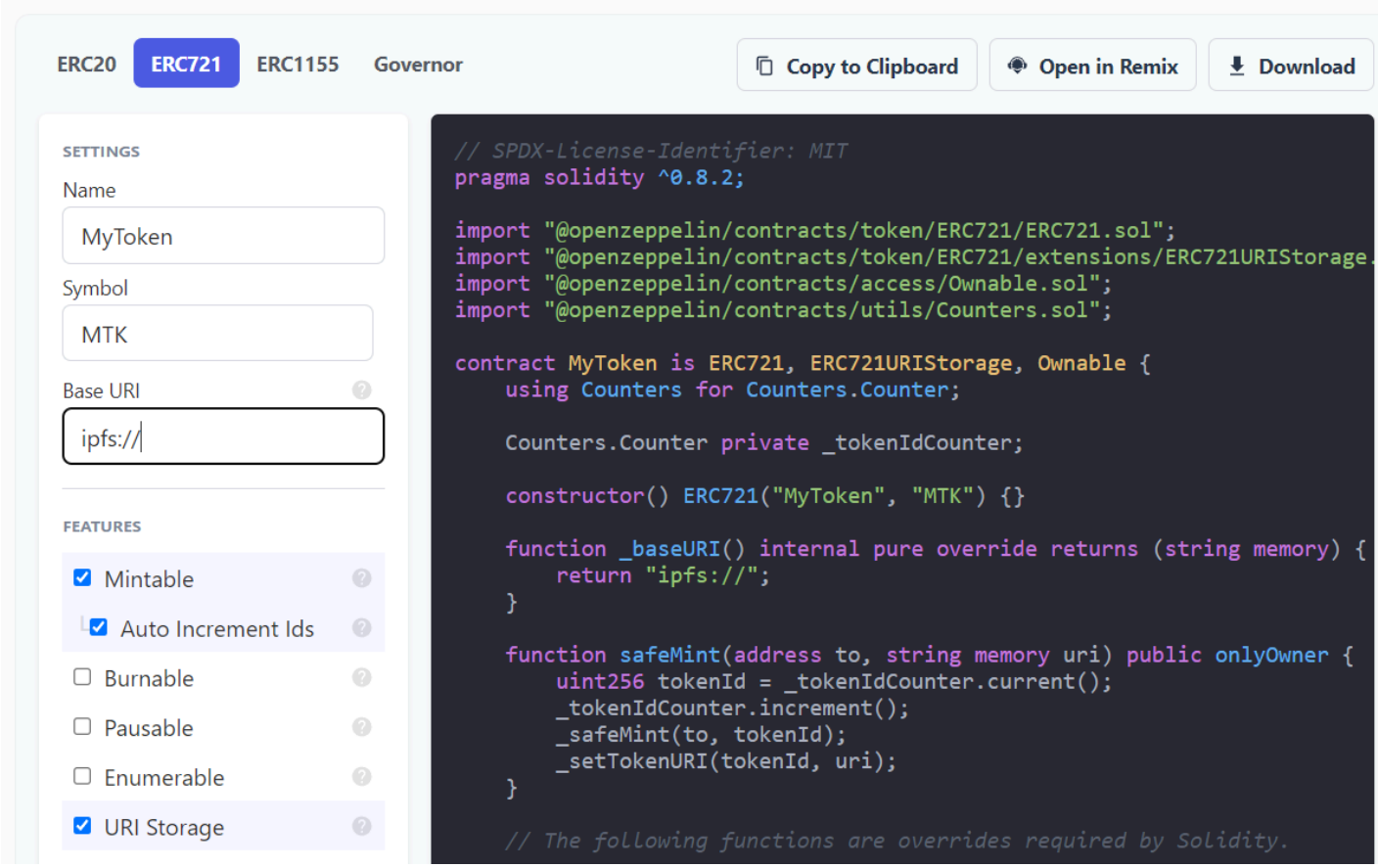
hardhat.config.js

```
module.exports = {
  solidity: "0.8.4",
  paths: {
    artifacts: './src/artifacts',
  },
};
```

SMART CONTRACT

Base ERC-721 Contract

Smart contracts have been standardized into a predictible API. When it comes to NFTs, the most common choice is ERC-721 and we can use a tool called OpenZeppelin to generate the initial boilerplate code.



Use the OpenZeppelin wizard to create a base contract

Now take the base contract and copy it into a Solidity file in the `contracts` directory.

contracts/MyNFT.sol

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

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

contract FiredGuys is ERC721, ERC721URIStorage, Ownable {
    using Counters for Counters.Counter;

    Counters.Counter private _tokenIdCounter;

    constructor() ERC721("FiredGuys", "MTK") {}

    function _baseURI() internal pure override returns (string memory) {
        return "ipfs://";
    }

    function safeMint(address to, string memory uri) public onlyOwner {
        uint256 tokenId = _tokenIdCounter.current();
        _tokenIdCounter.increment();
        _safeMint(to, tokenId);
        _setTokenURI(tokenId, uri);
    }

    // The following functions are overrides required by Solidity.

    function _burn(uint256 tokenId) internal override(ERC721, ERC721URIStorage) {
        super._burn(tokenId);
    }
```

```
    function tokenURI(uint256 tokenId)
        public
        view
        override(ERC721, ERC721URIStorage)
        returns (string memory)
    {
        return super.tokenURI(tokenId);
    }
}
```

Ensure URIs are Unique

First, create a mapping to ensure each token has a unique URI. Second, define a public function that can determine if a URI is already owned.

contracts/MyNFT.sol

```
contract FiredGuys is ERC721, ERC721URIStorage, Ownable {

    mapping(string => uint8) existingURIs;

  // ...

    function isContentOwned(string memory uri) public view returns (bool) {
        return existingURIs[uri] == 1;
    }
}
```

Pay to Mint

Let's add an additional method to the contract that handles the minting of a new token. It is a `payable` method, which means Ether (or other tokens like MATIC) can be sent from the end-user to the contract.

The method uses `require` to validate that (1) the URI is not already taken, and (2) the minimum amount of Ether has been sent. When the user calls this method, their wallet will prompt them for permission to transfer funds and execute the transation. In return, they will be given a new token linked to the metadata URI on IPFS.

contracts/MyNFT.sol

```
contract FiredGuys is ERC721, ERC721URIStorage, Ownable {

    // ...

    function payToMint(
        address recipient,
        string memory metadataURI
    ) public payable returns (uint256) {
        require(existingURIs[metadataURI] != 1, 'NFT already minted!');
        require (msg.value >= 0.05 ether, 'Need to pay up!');

        uint256 newItemId = _tokenIdCounter.current();
        _tokenIdCounter.increment();
        existingURIs[metadataURI] = 1;

        _mint(recipient, newItemId);
        _setTokenURI(newItemId, metadataURI);

        return newItemId;
    }

}
```

Deploy Contract

Before we can build an app, we need to deploy the contract. Update the sample script with your contract details.

JS scripts/sample-script.js

```
const hre = require("hardhat");

async function main() {

  const FiredGuys = await hre.ethers.getContractFactory("FiredGuys");
  const firedGuys = await FiredGuys.deploy();

  await firedGuys.deployed();

  console.log("My NFT deployed to:", firedGuys.address);
}


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

Use hardhat to run a blockchain network on localhost, then compile and deploy it from the terminal.

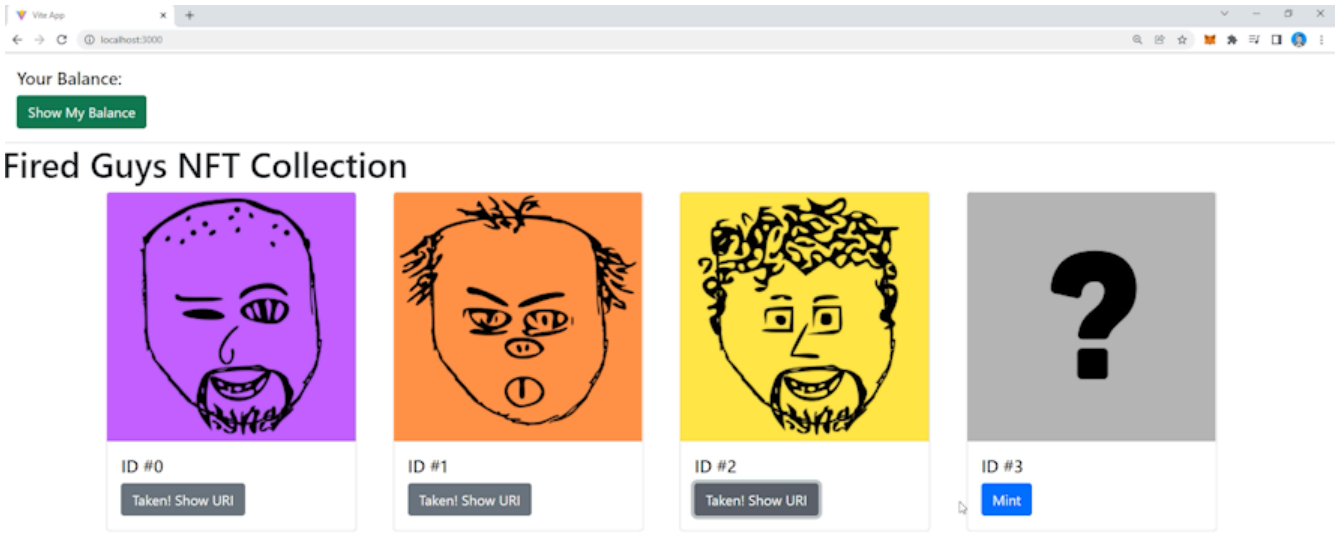
command line

```
# terminal 1
npx hardhat node

# terminal 2
npx hardhat compile
npx hardhat run scripts/sample-script.js --network localhost
```

Make a note of deployed contract address

WEB3 FRONTEND



Web3 NFT demo app

Check for Wallet Plugin

Before using the app, the user must have MetaMask installed. Create a component to prompt the user to install MetaMask.

Install.jsx

```
const Install = () => {
    return (
      <div>
        <h3>Follow the link to install 👇</h3>
        <a href="https://metamask.io/download.html">Meta Mask</a>
      </div>
    );
  };

export default Install;
```

If the plugin is installed, then render the `Home` screen.

App.jsx

```
import Install from './components/Install';
import Home from './components/Home';

function App() {

  if (window.ethereum) {
    return <Home />;
  } else {
    return <Install />
  }
}

export default App;
```

Get the Wallet Balance

The app uses ethers.js to interact with the user's wallet and the blockchain. The `getBalance` function returns the balance of the user's wallet.

WalletBalance.jsx

```
import { useState } from 'react';
import { ethers } from 'ethers';
```

```
function WalletBalance() {

    const [balance, setBalance] = useState();

    const getBalance = async () => {
        const [account] = await window.ethereum.request({ method: 'eth_requestAccounts' });
        const provider = new ethers.providers.Web3Provider(window.ethereum);
        const balance = await provider.getBalance(account);
        setBalance(ethers.utils.formatEther(balance));
    };

    return (
      <div>
          <h5>Your Balance: {balance}</h5>
          <button onClick={() => getBalance()}>Show My Balance</button>
      </div>
    );
  };

  export default WalletBalance;
```

Loop through through Existing NFTs

In the home screen, we use ethers.js to make a reference to the deployed contract. We request the total number of minted tokens, then create a loop to render a child component for each one.

Home.jsx

```
import WalletBalance from './WalletBalance';
import { useEffect, useState } from 'react';

import { ethers } from 'ethers';
import FiredGuys from '../artifacts/contracts/MyNFT.sol/MyNFT.json';

const contractAddress = 'YOUR_DEPLOYED_CONTRACT_ADDRESS';

const provider = new ethers.providers.Web3Provider(window.ethereum);

// get the end user
const signer = provider.getSigner();

// get the smart contract
const contract = new ethers.Contract(contractAddress, FiredGuys.abi, signer);


function Home() {

  const [totalMinted, setTotalMinted] = useState(0);
  useEffect(() => {
    getCount();
  }, []);

  const getCount = async () => {
    const count = await contract.count();
    console.log(parseInt(count));
    setTotalMinted(parseInt(count));
  };

  return (
    <div>
        <WalletBalance />

        {Array(totalMinted + 1)
        .fill(0)
        .map((_, i) => (
            <div key={i}>
            <NFTImage tokenId={i} getCount={getCount} />
            </div>
        ))}
```

```
    </div>
  );
}
```

Mint a new Token

Finally, we can implement a method on each NFT image to mint a new token. It first makes a reference to the metadata URI. When the mint button is clicked it connects the user's wallet to the smart contact on the blockchain, then mints a new token using the `payToMint` method we defined in the Solidity code.

```
function NFTImage({ tokenId, getCount }) {
  const contentId = 'PINATA_CONTENT_ID';
  const metadataURI = `${contentId}/${tokenId}.json`;
  const imageURI = `https://gateway.pinata.cloud/ipfs/${contentId}/${tokenId}.png`;

  const [isMinted, setIsMinted] = useState(false);
  useEffect(() => {
    getMintedStatus();
  }, [isMinted]);

  const getMintedStatus = async () => {
    const result = await contract.isContentOwned(metadataURI);
    console.log(result)
    setIsMinted(result);
  };

  const mintToken = async () => {
    const connection = contract.connect(signer);
    const addr = connection.address;
    const result = await contract.payToMint(addr, metadataURI, {
      value: ethers.utils.parseEther('0.05'),
    });

    await result.wait();
    getMintedStatus();
    getCount();
  };

  async function getURI() {
    const uri = await contract.tokenURI(tokenId);
    alert(uri);
  }
  return (
    <div>
      <img src={isMinted ? imageURI : 'img/placeholder.png'}></img>
        <h5>ID #{tokenId}</h5>
        {!isMinted ? (
          <button onClick={mintToken}>
            Mint
          </button>
        ) : (
          <button onClick={getURI}>
            Taken! Show URI
          </button>
        )}
    </div>
  );
}

export default Home;
```

Find an issue with this page? Fix it on GitHub

Need help? Email hello@fireship.io

HELPFUL LINKS

Courses | Labs | Snippets | Tags | Contrib | Privacy | Terms