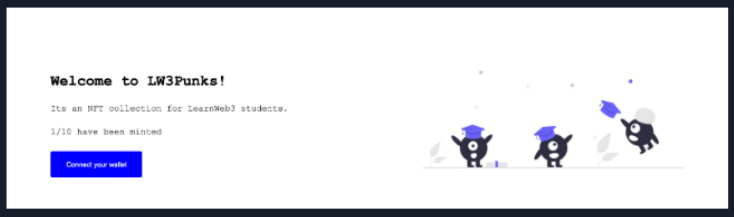
Lesson Type: Practical | Estimated Time: 4-6 hours | Current Score: 100%

# Create an NFT Collection with metadata stored on IPFS

Now its time for you to launch your own NFT collection and store its metadata on IPFS




## Requirements

- There should only exist 10 LearnWeb3 Punk NFT's and each one of them should be unique.
- User's should be able to mint only 1 NFT with one transaction.
- The metadata for the NFT's should be stored on IPFS
- There should be a website for your NFT Collection.
- The NFT contract should be deployed on Mumbai testnet

Lets start building 🚀

## Prerequisites

You should have completed the **IPFS Theory tutorial**

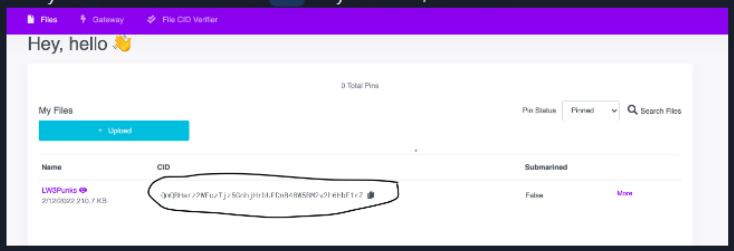
## Build

### IPFS

First, we need to upload our images and metadata on to IPFS. We'll use a service called **Pinata** which will help us upload and pin content on IPFS. Once you've signed up, go to the **Pinata Dashboard** and click on 'Upload' and then on 'Folder'.

Download **the LW3Punks folder** to your computer and then upload to it `Pinata`, name the folder `LW3Punks`

Now you should be able to see a `CID` for your folder, Awesome!




You can check that it actually got uploaded to IPFS is by opening this up: `https://ipfs.io/ipfs/your-nft-folder-cid` replace `your-nft-folder-cid` with the CID you recieved from pinata.

The images for your NFT's have now been uploaded to IPFS but just having images is not enough, each NFT should also have associated metadata

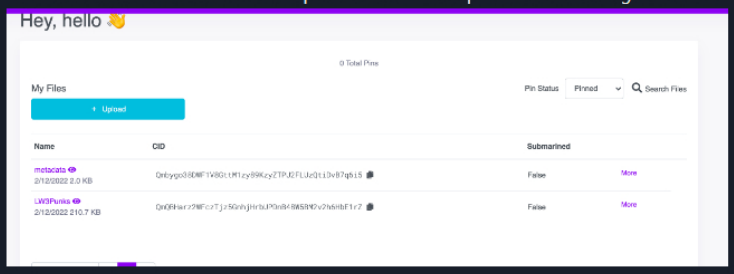
We will now upload metadata for each NFT to IPFS, each metadata file will be a `json` file. Example for metadata of NFT `1` has been given below:

```
{
  "name": "1",
  "description": "NFT Collection for LearnWeb3 Students",
  "image": "ipfs://QmQBHarz2WFczTjz5GnhjHrbUPDnB48W5BM2v2h6HbE1rZ/1.png"
}
```

Note how "image" has ipfs location in it instead of an `https` url. Also note that because you uploaded a folder, you will also need to specify which file within the folder has the correct image for the given NFT. Thus in our case the correct way to specify the location for an NFT image would be `ipfs://CID-OF-THE-LW3Punks-Folder/NFT-NAME.png`

We have pre-generated files for metadata for you, you can download them to your computer from **here**, upload these files to pinata and name the folder `metadata`

Now each NFT's metadata has been uploaded to IPFS and pinata should have generated a CID for your metadata folder




You can check that it actually got uploaded to IPFS is by opening this up: `https://ipfs.io/ipfs/your-metadata-folder-cid` replace `your-metadata-folder-cid` with the CID you recieved from pinata.

Copy this CID and store it on your notepad, you will need this futher down in the tutorial

## Contract

For the contract, we will use a simple NFT contract. We will also be using `Ownable.sol` from Openzeppelin which helps you manage the `Ownership` of a contract

By default, the owner of an Ownable contract is the account that deployed it, which is usually exactly what you want. Ownable also lets you:

- `transferOwnership` from the owner account to a new one, and
- `renounceOwnership` for the owner to relinquish this administrative privilege, a common pattern after an initial stage with centralized administration is over.

We would also be using an extension of ERC721 known as **ERC721 Enumerable**

ERC721 Enumerable is helps you to keep track of all the tokenIds in the contract and also the tokensIds held by an address for a given contract. Please have a look at the **functions** it implements before moving ahead

To build the smart contract we would be using **Hardhat**. Hardhat is an Ethereum development environment and framework designed for full stack development in Solidity. In simple words you can write your smart contract, deploy them, run tests, and debug your code.

To setup a Hardhat project, Open up a terminal and execute these commands

```
mkdir nft-ipfs
cd nft-ipfs
mkdir hardhat
cd hardhat
npm init --yes
npm install --save-dev hardhat
npx hardhat
```

Make sure you select `Create a Javascript Project` and then follow the instructions on your terminal.

In the same terminal now install `@openzeppelin/contracts` as we would be importing **Openzeppelin's ERC721Enumerable Contract** in our `LW3Punks` contract.

```
npm install @openzeppelin/contracts
```

Now lets create a new file inside the `nft-ipfs/hardhat/contracts` directory and call it `LW3Punks.sol`

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

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Strings.sol";

contract LW3Punks is ERC721Enumerable, Ownable {
    using Strings for uint256;
    /**
        * @dev _baseTokenURI for computing {tokenURI}. If set, the resulting URI for each
        * token will be the concatenation of the `baseURI` and the `tokenId`.
        */
    string _baseTokenURI;

    //  _price is the price of one LW3Punks NFT
    uint256 public _price = 0.01 ether;

    // _paused is used to pause the contract in case of an emergency
    bool public _paused;

    // max number of LW3Punks
    uint256 public maxTokenIds = 10;

    // total number of tokenIds minted
    uint256 public tokenIds;

    modifier onlyWhenNotPaused {
        require(!_paused, "Contract currently paused");
        _;
    }

    /**
        * @dev ERC721 constructor takes in a `name` and a `symbol` to the token collection.
        * name in our case is `LW3Punks` and symbol is `LW3P`.
        * Constructor for LW3P takes in the baseURI to set _baseTokenURI for the collection.
        */
    constructor (string memory baseURI) ERC721("LW3Punks", "LW3P") {
        _baseTokenURI = baseURI;
    }

    /**
    * @dev mint allows an user to mint 1 NFT per transaction.
    */
    function mint() public payable onlyWhenNotPaused {
        require(tokenIds < maxTokenIds, "Exceed maximum LW3Punks supply");
        require(msg.value >= _price, "Ether sent is not correct");
        tokenIds += 1;
        _safeMint(msg.sender, tokenIds);
    }

    /**
    * @dev _baseURI overides the Openzeppelin's ERC721 implementation which by default
    * returned an empty string for the baseURI
    */
    function _baseURI() internal view virtual override returns (string memory) {
        return _baseTokenURI;
    }

    /**
    * @dev tokenURI overides the Openzeppelin's ERC721 implementation for tokenURI function
    * This function returns the URI from where we can extract the metadata for a given tokenId
    */
    function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {
        require(_exists(tokenId), "ERC721Metadata: URI query for nonexistent token");

        string memory baseURI = _baseURI();
        // Here it checks if the length of the baseURI is greater than 0, if it is return the baseURI and attach
        // the tokenId and `.json` to it so that it knows the location of the metadata json file for a given
        // tokenId stored on IPFS
        // If baseURI is empty return an empty string
        return bytes(baseURI).length > 0 ? string(abi.encodePacked(baseURI, tokenId.toString(), ".json")) : "";
    }

    /**
```

```
    * @dev setPaused makes the contract paused or unpaused
        */
    function setPaused(bool val) public onlyOwner {
        _paused = val;
    }

    /**
    * @dev withdraw sends all the ether in the contract
    * to the owner of the contract
        */
    function withdraw() public onlyOwner  {
        address _owner = owner();
        uint256 amount = address(this).balance;
        (bool sent, ) =  _owner.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }

        // Function to receive Ether. msg.data must be empty
    receive() external payable {}

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}
}
```

Now let's install the `dotenv` package to be able to import the env file and use it in our config. Open up a terminal pointing at `hardhat` directory and execute this command

```
npm install dotenv
```

Create a `.env` file in the `hardhat` folder and add the following lines. Follow the instructions below:

Go to **Quicknode** and sign up for an account. If you already have an account, log in. Quicknode is a node provider that lets you connect to various different blockchains. We will be using it to deploy our contract through Hardhat. After creating an account, `Create an endpoint` on Quicknode, select `Polygon`, and then select the `Mumbai` network. Click `Continue` in the bottom right and then click on `Create Endpoint`. Copy the link given to you in `HTTP Provider` and add it to the `.env` file below for `QUICKNODE_HTTP_URL`.

> NOTE: If you previously set up a Mumbai endpoint, you can keep using that one. If you have any other endpoints from the past, you need to delete it to create a new one.

To get your private key, you need to export it from Metamask. Open Metamask, click on the three dots, click on `Account Details` and then `Export Private Key`. MAKE SURE YOU ARE USING A TEST ACCOUNT THAT DOES NOT HAVE MAINNET FUNDS FOR THIS. Add this Private Key below in your `.env` file for `PRIVATE_KEY` variable.

Also, make sure you have some Mumbai `MATIC` tokens to work with. If you don't know how to get them, follow **this guide by ThirdWeb**.

```
QUICKNODE_HTTP_URL="add-quicknode-http-provider-url-here"

PRIVATE_KEY="add-the-private-key-here"
```

Lets deploy the contract to `mumbai` network. Create a new file, or replace the existing default one, named `deploy.js` under the `scripts` folder. Remember to replace `YOUR-METADATA-CID` with the CID you saved to your notepad.

```
const { ethers } = require("hardhat");
require("dotenv").config({ path: ".env" });

async function main() {
  // URL from where we can extract the metadata for a LW3Punks
  const metadataURL = "ipfs://YOUR-METADATA-CID/";
  /*
  A ContractFactory in ethers.js is an abstraction used to deploy new smart contracts,
  so lw3PunksContract here is a factory for instances of our LW3Punks contract.
  */
  const lw3PunksContract = await ethers.getContractFactory("LW3Punks");

  // deploy the contract
  const deployedLW3PunksContract = await lw3PunksContract.deploy(metadataURL);

  await deployedLW3PunksContract.deployed();
```

```
  // print the address of the deployed contract
  console.log("LW3Punks Contract Address:", deployedLW3PunksContract.address);
}

// Call the main function and catch if there is any error
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

Now open the hardhat.config.js file, we would add the `mumbai` network here so that we can deploy our contract to mumbai. Replace all the lines in the `hardhat.config.js` file with the given below lines

```
require("@nomicfoundation/hardhat-toolbox");
require("dotenv").config({ path: ".env" });

const QUICKNODE_HTTP_URL = process.env.QUICKNODE_HTTP_URL;
const PRIVATE_KEY = process.env.PRIVATE_KEY;

module.exports = {
  solidity: "0.8.4",
  networks: {
    mumbai: {
      url: QUICKNODE_HTTP_URL,
      accounts: [PRIVATE_KEY],
    },
  },
};
```

Compile the contract, open up a terminal pointing at `hardhat` directory and execute this command

```
npx hardhat compile
```

To deploy, open up a terminal pointing at `hardhat` directory and execute this command

```
npx hardhat run scripts/deploy.js --network mumbai
```

Save the LW3Punks contract address that was printed on your terminal in your notepad, you would need it futher down in the tutorial.

## Website

To develop the website we would be using **React** and **Next Js**. React is a javascript framework which is used to make websites and Next Js is built on top of React.

First, You would need to create a new `next` app. Your folder structure should look something like

```
- nft-ipfs
      - hardhat
      - my-app
```

To create this `my-app`, in the terminal point to the `nft-ipfs` folder and type

```
npx create-next-app@latest
```

and press `enter` for all the questions

Now to run the app, execute these commands in the terminal

```
cd my-app
npm run dev
```

Now go to `http://localhost:3000`, your app should be running 🤘

Now lets install **Web3Modal library**. Web3Modal is an easy-to-use library to help developers add support for multiple providers in their apps with a simple customizable configuration. By default Web3Modal Library supports injected providers like (Metamask, Dapper, Gnosis Safe, Frame, Web3 Browsers, etc), You can also easily configure the library to support Portis, Fortmatic, Squarelink, Torus, Authereum, D'CENT Wallet and Arkane. Open up a terminal pointing at `my-app` directory and execute this command

```
npm install web3modal
```

In the same terminal also install `ethers.js`

```
npm install ethers
```

In your public folder, download this folder and all the images in it **the LW3Punks folder**. Make sure that the name of the downloaded folder is `LW3Punks`

Now go to `styles` folder and replace all the contents of `Home.modules.css` file with the following code, this would add some styling to your dapp:

```
.main {
  min-height: 90vh;
  display: flex;
  flex-direction: row;
  justify-content: center;
  align-items: center;
  font-family: "Courier New", Courier, monospace;
}

.footer {
  display: flex;
  padding: 2rem 0;
  border-top: 1px solid #eaeaea;
  justify-content: center;
  align-items: center;
}

.image {
  width: 70%;
  height: 50%;
  margin-left: 20%;
}

.title {
  font-size: 2rem;
  margin: 2rem 0;
}

.description {
  line-height: 1;
  margin: 2rem 0;
  font-size: 1.2rem;
}

.button {
  border-radius: 4px;
  background-color: blue;
  border: none;
  color: #ffffff;
  font-size: 15px;
  padding: 20px;
  width: 200px;
  cursor: pointer;
  margin-bottom: 2%;
}
@media (max-width: 1000px) {
  .main {
    width: 100%;
    flex-direction: column;
    justify-content: center;
    align-items: center;
  }
}
```

Open your `index.js` file under the `pages` folder and paste the following code, explanation of the code can be found in the comments.

```jsx
import { Contract, providers, utils } from "ethers";
import Head from "next/head";
import React, { useEffect, useRef, useState } from "react";
import Web3Modal from "web3modal";
import { abi, NFT_CONTRACT_ADDRESS } from "../constants";
import styles from "../styles/Home.module.css";

export default function Home() {
  // walletConnected keep track of whether the user's wallet is connected or not
  const [walletConnected, setWalletConnected] = useState(false);
  // loading is set to true when we are waiting for a transaction to get mined
  const [loading, setLoading] = useState(false);
  // tokenIdsMinted keeps track of the number of tokenIds that have been minted
  const [tokenIdsMinted, setTokenIdsMinted] = useState("0");
  // Create a reference to the Web3 Modal (used for connecting to Metamask) which persists as long as the page is open
  const web3ModalRef = useRef();

  /**
   * publicMint: Mint an NFT
   */
  const publicMint = async () => {
    try {
      console.log("Public mint");
      // We need a Signer here since this is a 'write' transaction.
      const signer = await getProviderOrSigner(true);
      // Create a new instance of the Contract with a Signer, which allows
      // update methods
      const nftContract = new Contract(NFT_CONTRACT_ADDRESS, abi, signer);
      // call the mint from the contract to mint the LW3Punks
      const tx = await nftContract.mint({
        // value signifies the cost of one LW3Punks which is "0.01" eth.
        // We are parsing `0.01` string to ether using the utils library from ethers.js
        value: utils.parseEther("0.01"),
      });
      setLoading(true);
      // wait for the transaction to get mined
      await tx.wait();
      setLoading(false);
      window.alert("You successfully minted a LW3Punk!");
    } catch (err) {
      console.error(err);
    }
  };

  /*
        connectWallet: Connects the MetaMask wallet
      */
  const connectWallet = async () => {
    try {
      // Get the provider from web3Modal, which in our case is MetaMask
      // When used for the first time, it prompts the user to connect their wallet
      await getProviderOrSigner();
      setWalletConnected(true);
    } catch (err) {
      console.error(err);
    }
  };

  /**
   * getTokenIdsMinted: gets the number of tokenIds that have been minted
   */
  const getTokenIdsMinted = async () => {
    try {
      // Get the provider from web3Modal, which in our case is MetaMask
      // No need for the Signer here, as we are only reading state from the blockchain
      const provider = await getProviderOrSigner();
      // We connect to the Contract using a Provider, so we will only
      // have read-only access to the Contract
      const nftContract = new Contract(NFT_CONTRACT_ADDRESS, abi, provider);
      // call the tokenIds from the contract
      const _tokenIds = await nftContract.tokenIds();
      console.log("tokenIds", _tokenIds);
      //_tokenIds is a `Big Number`. We need to convert the Big Number to a string
      setTokenIdsMinted(_tokenIds.toString());
    } catch (err) {
      console.error(err);
    }
  };

  /**
   * Returns a Provider or Signer object representing the Ethereum RPC with or without the
   * signing capabilities of metamask attached
   *
   * A `Provider` is needed to interact with the blockchain - reading transactions, reading balances, reading state,
   *
   * A `Signer` is a special type of Provider used in case a `write` transaction needs to be made to the blockchain,
```

```
   * A Signer is a special type of Provider used in case a `write` transaction needs to be made to the blockchain, w
   * needing to make a digital signature to authorize the transaction being sent. Metamask exposes a Signer API to al
   * request signatures from the user using Signer functions.
   *
   * @param {*} needSigner - True if you need the signer, default false otherwise
   */
  const getProviderOrSigner = async (needSigner = false) => {
    // Connect to Metamask
    // Since we store `web3Modal` as a reference, we need to access the `current` value to get access to the underlyi
    const provider = await web3ModalRef.current.connect();
    const web3Provider = new providers.Web3Provider(provider);

    // If user is not connected to the Mumbai network, let them know and throw an error
    const { chainId } = await web3Provider.getNetwork();
    if (chainId !== 80001) {
      window.alert("Change the network to Mumbai");
      throw new Error("Change network to Mumbai");
    }

    if (needSigner) {
      const signer = web3Provider.getSigner();
      return signer;
    }
    return web3Provider;
  };

  // useEffects are used to react to changes in state of the website
  // The array at the end of function call represents what state changes will trigger this effect
  // In this case, whenever the value of `walletConnected` changes - this effect will be called
  useEffect(() => {
    // if wallet is not connected, create a new instance of Web3Modal and connect the MetaMask wallet
    if (!walletConnected) {
      // Assign the Web3Modal class to the reference object by setting it's `current` value
      // The `current` value is persisted throughout as long as this page is open
      web3ModalRef.current = new Web3Modal({
        network: "mumbai",
        providerOptions: {},
        disableInjectedProvider: false,
      });

      connectWallet();

      getTokenIdsMinted();

      // set an interval to get the number of token Ids minted every 5 seconds
      setInterval(async function () {
        await getTokenIdsMinted();
      }, 5 * 1000);
    }
  }, [walletConnected]);

  /*
      renderButton: Returns a button based on the state of the dapp
    */
  const renderButton = () => {
    // If wallet is not connected, return a button which allows them to connect their wallet
    if (!walletConnected) {
      return (
        <button onClick={connectWallet} className={styles.button}>
          Connect your wallet
        </button>
      );
    }

    // If we are currently waiting for something, return a loading button
    if (loading) {
      return <button className={styles.button}>Loading...</button>;
    }

    return (
      <button className={styles.button} onClick={publicMint}>
        Public Mint 🚀
      </button>
    );
  };

  return (
    <div>
      <Head>
        <title>LW3Punks</title>
        <meta name="description" content="LW3Punks-Dapp" />
        <link rel="icon" href="/favicon.ico" />
      </Head>
      <div className={styles.main}>
        <div>
          <h1 className={styles.title}>Welcome to LW3Punks!</h1>
```

```
            <div className={styles.description}>
              Its an NFT collection for LearnWeb3 students.
            </div>
            <div className={styles.description}>
              {tokenIdsMinted}/10 have been minted
            </div>
            {renderButton()}
          </div>
          <div>
            <img className={styles.image} src="./LW3punks/1.png" />
          </div>
        </div>

        <footer className={styles.footer}>Made with &#10084; by LW3Punks</footer>
      </div>
    );
  }
```

Now create a new folder under the my-app folder and name it `constants`. In the constants folder create a file, `index.js` and paste the following code.

Replace `"address of your NFT contract"` with the address of the LW3Punks contract that you deployed and saved to your notepad.

Replace `---your abi---` with the abi of your LW3Punks Contract. To get the abi for your contract, go to your `hardhat/artifacts/contracts/LW3Punks.sol` folder and from your `LW3Punks.json` file get the array marked under the `"abi"` key.

```
export const NFT_CONTRACT_ADDRESS = "address of your NFT contract";
export const abi = "---your abi---";
```

Now in your terminal which is pointing to `my-app` folder, execute

```
npm run dev
```

Your LW3Punks NFT dapp should now work without errors 🚀

---

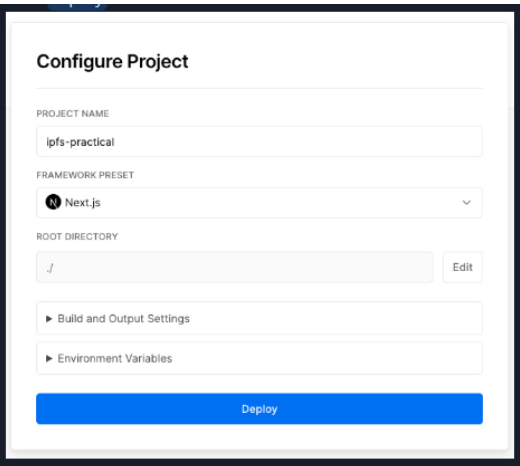
### Push to github

Make sure before proceeding you have **pushed all your code to github** :)

## Deploying your dApp

We will now deploy your dApp, so that everyone can see your website and you can share it with all of your LearnWeb3 DAO friends.

- Go to https://vercel.com/ and sign in with your GitHub
- Then click on `New Project` button and then select your IPFS-Practical repo
- When configuring your new project, Vercel will allow you to customize your `Root Directory`
- Click `Edit` next to `Root Directory` and set it to `my-app`
- Select the Framework as `Next.js`
- Click `Deploy`




- 

Now you can see your deployed website by going to your dashboard, selecting your project, and copying the `domain` from there! Save the `domain` on notepad, you would need it later.

- 

Share your website link with everyone on discord :) and spread happiness.

**Submit Practical**

Verify your smart contract address to pass the assessment for this level.

Ethereum Rinkeby | 0x... | Submit