Dashboard
Intro to React and Next.js
What is Gas, and why is it needed?
What is mining, and why is it done?
How does Proof of Work work?
Digging into the Ethereum Virtual Machine
Advanced Solidity syntax and concepts
Providers, Signers, ABIs, and Approval Flows
Build a full whitelist dApp
Build a full NFT collection
Launch your own Initial Coin Offering (ICO)
Build your own fully on-chain DAO to invest in NFTs
Intro and deep dive into decentralized exchanges
Build your own Decentralized Exchange like Uniswap

Lesson Type: Practical | Estimated Time: 6-10 hours | Current Score: 0%

Build an NFT Collection

Now its time for you to launch your own NFT collection - `Crypto Devs`.

Requirements

- There should only exist 20 Crypto Dev NFT's and each one of them should be unique.
- User's should be able to mint only 1 NFT with one transaction.
- Whitelisted users, should have a 5 min presale period before the actual sale where they are guaranteed 1 NFT per transaction.
- There should be a website for your NFT Collection.

Lets start building 🚀

Prerequisites

- You should have completed the `Whitelist dApp` tutorial from earlier.

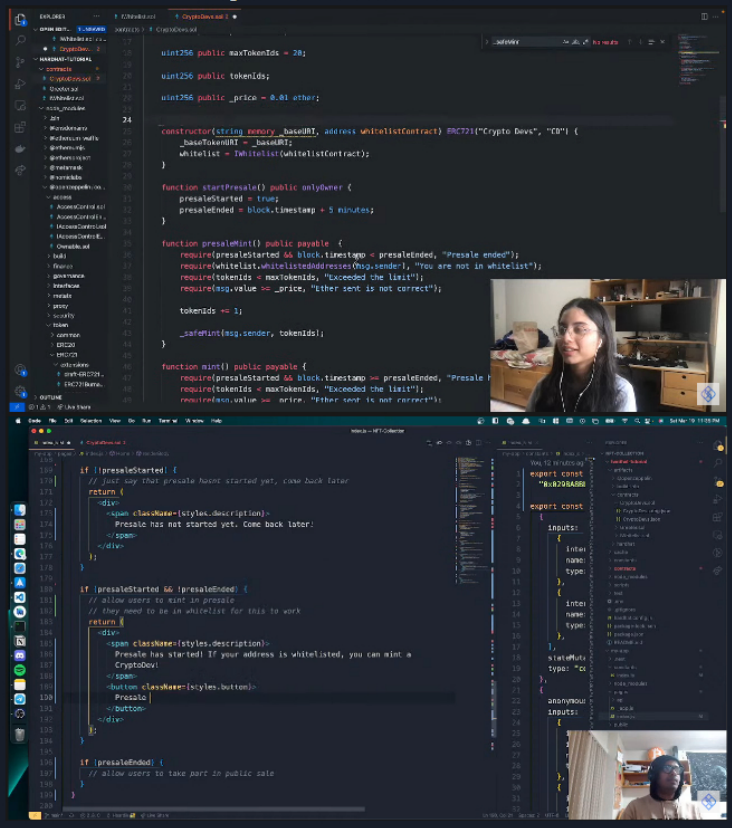
Theory

- What is a Non-Fungible Token? Fungible means to be the same or interchangeable eg. Eth is fungible. With this in mind, NFTs are unique; each one is different. Every single token has unique characteristics and values. They are all distinguishable from one another and are not interchangeable eg Unique Art
- What is ERC-721? ERC-721 is an open standard that describes how to build Non-Fungible tokens on EVM (Ethereum Virtual Machine) compatible blockchains; it is a standard interface for Non-Fungible tokens; it has a set of rules which make it easy to work with NFTs. Before moving ahead have a look at all the functions supported by **ERC721**

Build

Prefer a Video?

If you would rather learn from a video, we have a recording available of this tutorial on our YouTube. Watch the video by clicking on the screenshot below, or go ahead and read the tutorial!



Smart Contract

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

- ERC721 Enumerable 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-Collection
cd NFT-Collection
mkdir hardhat-tutorial
cd hardhat-tutorial
npm init --yes
npm install --save-dev hardhat
```

In the same directory where you installed Hardhat run:

```
npx hardhat
```

Make sure you select `Create a Javascript Project` and then follow the steps in the terminal to complete your Hardhat setup.

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

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

We will need to call the `Whitelist Contract` that you deployed for your previous level to check for addresses that were whitelisted and give them presale access. As we only need to call `mapping(address => bool) public whitelistedAddresses;` We can create an interface for `Whitelist contract` with a function only for this mapping, this way we would save `gas` as we would not need to inherit and deploy the entire `Whitelist Contract` but only a part of it.

Create a new file inside the `contracts` directory and call it `IWhitelist.sol`.

> Note: Solidity files which only include interfaces are often prefixed with `I` to signify they're only an interface.

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

interface IWhitelist {
    function whitelistedAddresses(address) external view returns (bool);
}
```

Now lets create a new file inside the `contracts` directory and call it `CryptoDevs.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 "./IWhitelist.sol";

contract CryptoDevs is ERC721Enumerable, Ownable {
    /**
      * @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 Crypto Dev 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 CryptoDevs
    uint256 public maxTokenIds = 20;

    // total number of tokenIds minted
    uint256 public tokenIds;

    // Whitelist contract instance
    IWhitelist whitelist;

    // boolean to keep track of whether presale started or not
    bool public presaleStarted;

    // timestamp for when presale would end
    uint256 public presaleEnded;

    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 `Crypto Devs` and symbol is `CD`.
      * Constructor for Crypto Devs takes in the baseURI to set _baseTokenURI for the collection.
      * It also initializes an instance of whitelist interface.
      */
    constructor (string memory baseURI, address whitelistContract) ERC721("Crypto Devs", "CD") {
        _baseTokenURI = baseURI;
```

```
        _baseTokenURI = baseURI;
        whitelist = IWhitelist(whitelistContract);
    }

    /**
    * @dev startPresale starts a presale for the whitelisted addresses
      */
    function startPresale() public onlyOwner {
        presaleStarted = true;
        // Set presaleEnded time as current timestamp + 5 minutes
        // Solidity has cool syntax for timestamps (seconds, minutes, hours, days, years)
        presaleEnded = block.timestamp + 5 minutes;
    }

    /**
      * @dev presaleMint allows a user to mint one NFT per transaction during the presale.
      */
    function presaleMint() public payable onlyWhenNotPaused {
        require(presaleStarted && block.timestamp < presaleEnded, "Presale is not running");
        require(whitelist.whitelistedAddresses(msg.sender), "You are not whitelisted");
        require(tokenIds < maxTokenIds, "Exceeded maximum Crypto Devs supply");
        require(msg.value >= _price, "Ether sent is not correct");
        tokenIds += 1;
        //_safeMint is a safer version of the _mint function as it ensures that
        // if the address being minted to is a contract, then it knows how to deal with ERC721 tokens
        // If the address being minted to is not a contract, it works the same way as _mint
        _safeMint(msg.sender, tokenIds);
    }

    /**
    * @dev mint allows a user to mint 1 NFT per transaction after the presale has ended.
    */
    function mint() public payable onlyWhenNotPaused {
        require(presaleStarted && block.timestamp >=  presaleEnded, "Presale has not ended yet");
        require(tokenIds < maxTokenIds, "Exceed maximum Crypto Devs 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 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 `dotenv` package to be able to import the env file and use it in our config. Open up a terminal pointing at `hardhat-tutorial` directory and execute this command

```
npm install dotenv
```

Now create a `.env` file in the `hardhat-tutorial` 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 `Ethereum`, and then select the `Goerli` 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 Goerli Endpoint on Quicknode during the Freshman Track, you can use the same URL as before. No need to delete it and set up 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.

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

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

Lets deploy the contract to the `goerli` network. Create a new file, or replace the default file, named `deploy.js` under the `scripts` folder

Let's write some code to deploy the contract in `deploy.js` file.

```
const { ethers } = require("hardhat");
require("dotenv").config({ path: ".env" });
const { WHITELIST_CONTRACT_ADDRESS, METADATA_URL } = require("../constants");

async function main() {
  // Address of the whitelist contract that you deployed in the previous module
  const whitelistContract = WHITELIST_CONTRACT_ADDRESS;
  // URL from where we can extract the metadata for a Crypto Dev NFT
  const metadataURL = METADATA_URL;
  /*
  A ContractFactory in ethers.js is an abstraction used to deploy new smart contracts,
  so cryptoDevsContract here is a factory for instances of our CryptoDevs contract.
  */
  const cryptoDevsContract = await ethers.getContractFactory("CryptoDevs");

  // deploy the contract
  const deployedCryptoDevsContract = await cryptoDevsContract.deploy(
    metadataURL,
    whitelistContract
  );

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

As you can see, `deploy.js` requires some constants. Lets create a folder named `constants` under the `hardhat-tutorial` folder. Create an `index.js` file inside the `constants` folder and add the following lines to the file. Replace "address-of-the-whitelist-contract" with the address of the whitelist contract that you deployed in the previous tutorial. For Metadata_URL, just copy the sample one that has been provided. We would replace this further down in the tutorial.

```
// Address of the Whitelist Contract that you deployed
const WHITELIST_CONTRACT_ADDRESS = "address-of-the-whitelist-contract";
// URL to extract Metadata for a Crypto Dev NFT
const METADATA_URL = "https://nft-collection-sneh1999.vercel.app/api/";

module.exports = { WHITELIST_CONTRACT_ADDRESS, METADATA_URL };
```

Now open the `hardhat.config.js` file, we'll set-up the `goerli` network here so that we can deploy our contract to the Goerli network. 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: {
    goerli: {
      url: QUICKNODE_HTTP_URL,
      accounts: [PRIVATE_KEY],
    },
  },
};
```

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

```
npx hardhat compile
```

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

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

Save the Crypto Devs 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-Collection
       - hardhat-tutorial
       - my-app
```

To create this `my-app`, in the terminal point to NFT-Collection 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(https://github.com/Web3Modal/web3modal). 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 (**Download Link**). Make sure that the name of the downloaded folder is `cryptodevs`

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

```
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);
  // presaleStarted keeps track of whether the presale has started or not
  const [presaleStarted, setPresaleStarted] = useState(false);
  // presaleEnded keeps track of whether the presale ended
  const [presaleEnded, setPresaleEnded] = useState(false);
  // loading is set to true when we are waiting for a transaction to get mined
  const [loading, setLoading] = useState(false);
  // checks if the currently connected MetaMask wallet is the owner of the contract
  const [isOwner, setIsOwner] = 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();

/**
 * presaleMint: Mint an NFT during the presale
 */
const presaleMint = async () => {
  try {
    // 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 presaleMint from the contract, only whitelisted addresses would be able to mint
    const tx = await nftContract.presaleMint({
      // value signifies the cost of one crypto dev 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 Crypto Dev!");
  } catch (err) {
    console.error(err);
  }
};

/**
 * publicMint: Mint an NFT after the presale
 */
const publicMint = async () => {
  try {
    // 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 Crypto Dev
    const tx = await nftContract.mint({
      // value signifies the cost of one crypto dev 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 Crypto Dev!");
  } 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);
  }
};

/**
 * startPresale: starts the presale for the NFT Collection
 */
const startPresale = async () => {
  try {
    // 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 startPresale from the contract
    const tx = await nftContract.startPresale();
    setLoading(true);
    // wait for the transaction to get mined
```

```
    await tx.wait();
    setLoading(false);
    // set the presale started to true
    await checkIfPresaleStarted();
  } catch (err) {
    console.error(err);
  }
};

/**
 * checkIfPresaleStarted: checks if the presale has started by quering the `presaleStarted`
 * variable in the contract
 */
const checkIfPresaleStarted = 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 presaleStarted from the contract
    const _presaleStarted = await nftContract.presaleStarted();
    if (!_presaleStarted) {
      await getOwner();
    }
    setPresaleStarted(_presaleStarted);
    return _presaleStarted;
  } catch (err) {
    console.error(err);
    return false;
  }
};

/**
 * checkIfPresaleEnded: checks if the presale has ended by quering the `presaleEnded`
 * variable in the contract
 */
const checkIfPresaleEnded = 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 presaleEnded from the contract
    const _presaleEnded = await nftContract.presaleEnded();
    // _presaleEnded is a Big Number, so we are using the lt(less than function) instead of `<`
    // Date.now()/1000 returns the current time in seconds
    // We compare if the _presaleEnded timestamp is less than the current time
    // which means presale has ended
    const hasEnded = _presaleEnded.lt(Math.floor(Date.now() / 1000));
    if (hasEnded) {
      setPresaleEnded(true);
    } else {
      setPresaleEnded(false);
    }
    return hasEnded;
  } catch (err) {
    console.error(err);
    return false;
  }
};

/**
 * getOwner: calls the contract to retrieve the owner
 */
const getOwner = 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 owner function from the contract
    const _owner = await nftContract.owner();
    // We will get the signer now to extract the address of the currently connected MetaMask account
    const signer = await getProviderOrSigner(true);
    // Get the address associated to the signer which is connected to  MetaMask
    const address = await signer.getAddress();
    if (address.toLowerCase() === _owner.toLowerCase()) {
      setIsOwner(true);
    }
  } catch (err) {
```

```
    console.error(err.message);
  }
};

/**
 * 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();
    //_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,
 * 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 Goerli network, let them know and throw an error
  const { chainId } = await web3Provider.getNetwork();
  if (chainId !== 5) {
    window.alert("Change the network to Goerli");
    throw new Error("Change network to Goerli");
  }

  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: "goerli",
      providerOptions: {},
      disableInjectedProvider: false,
    });
    connectWallet();

    // Check if presale has started and ended
    const _presaleStarted = checkIfPresaleStarted();
    if (_presaleStarted) {
      checkIfPresaleEnded();
    }

    getTokenIdsMinted();

    // Set an interval which gets called every 5 seconds to check presale has ended
    const presaleEndedInterval = setInterval(async function () {
      const _presaleStarted = await checkIfPresaleStarted();
      if (_presaleStarted) {
        const _presaleEnded = await checkIfPresaleEnded();
        if (_presaleEnded) {
```

```
            clearInterval(presaleEndedInterval);
          }
        }
      }, 5 * 1000);

      // 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 wllet
    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>;
    }

    // If connected user is the owner, and presale hasnt started yet, allow them to start the presale
    if (isOwner && !presaleStarted) {
      return (
        <button className={styles.button} onClick={startPresale}>
          Start Presale!
        </button>
      );
    }

    // If connected user is not the owner but presale hasn't started yet, tell them that
    if (!presaleStarted) {
      return (
        <div>
          <div className={styles.description}>Presale hasnt started!</div>
        </div>
      );
    }

    // If presale started, but hasn't ended yet, allow for minting during the presale period
    if (presaleStarted && !presaleEnded) {
      return (
        <div>
          <div className={styles.description}>
            Presale has started!!! If your address is whitelisted, Mint a Crypto
            Dev 🥳
          </div>
          <button className={styles.button} onClick={presaleMint}>
            Presale Mint 🚀
          </button>
        </div>
      );
    }

    // If presale started and has ended, its time for public minting
    if (presaleStarted && presaleEnded) {
      return (
        <button className={styles.button} onClick={publicMint}>
          Public Mint 🚀
        </button>
      );
    }
  };

  return (
    <div>
      <Head>
        <title>Crypto Devs</title>
        <meta name="description" content="Whitelist-Dapp" />
        <link rel="icon" href="/favicon.ico" />
      </Head>
      <div className={styles.main}>
        <div>
          <h1 className={styles.title}>Welcome to Crypto Devs!</h1>
          <div className={styles.description}>
            Its an NFT collection for developers in Crypto.
          </div>
```

```
          <div className={styles.description}>
            {tokenIdsMinted}/20 have been minted
          </div>
          {renderButton()}
        </div>
        <div>
          <img className={styles.image} src="./cryptodevs/0.svg" />
        </div>
      </div>

      <footer className={styles.footer}>
        Made with &#10084; by Crypto Devs
      </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 `"addres of your NFT contract"` with the address of the CryptoDevs contract that you deployed and saved to your notepad. Replace `---your abi---` with the abi of your CryptoDevs Contract. To get the abi for your contract, go to your `hardhat-tutorial/artifacts/contracts/CryptoDevs.sol` folder and from your `CryptoDevs.json` file get the array marked under the `"abi"` key.

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

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

```
npm run dev
```

Your Crypto Devs NFT dapp should now work without errors 🚀

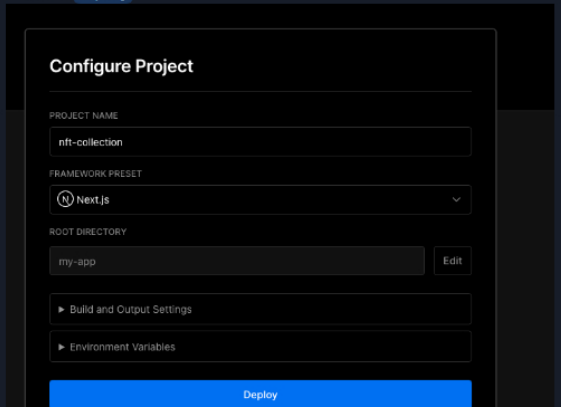
Push to github

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 NFT-Collection 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.

View your Collection on Opensea

Now lets make your collection is available on Opensea

To make the collection available on Opensea, we would need to create a metadata endpoint. This endpoint would return the metadata for an NFT given its `tokenId`.

Open your `my-app` folder and under `pages/api` folder, create a new file named `[tokenId].js` (Make sure the name has the brackets as well). Adding the brackets helps create dynamic routes in **next js**

Add the following lines to `[tokenId].js` file. Read about adding API routes in `next js` **here**

```
export default function handler(req, res) {
  // get the tokenId from the query params
  const tokenId = req.query.tokenId;
  // As all the images are uploaded on github, we can extract the images from github directly.
  const image_url =
    "https://raw.githubusercontent.com/LearnWeb3DAO/NFT-Collection/main/my-app/public/cryptodevs/";
  // The api is sending back metadata for a Crypto Dev
  // To make our collection compatible with Opensea, we need to follow some Metadata standards
  // when sending back the response from the api
  // More info can be found here: https://docs.opensea.io/docs/metadata-standards
  res.status(200).json({
    name: "Crypto Dev #" + tokenId,
    description: "Crypto Dev is a collection of developers in crypto",
    image: image_url + tokenId + ".svg",
  });
}
```

Now you have an API route that OpenSea, and other websites, can call to retrieve the metadata for the NFT. Those websites first call `tokenURI` on the smart contract to get the link of where the NFT metadata is stored. `tokenURI` will give them your API route we just created. Then, they can call this API route to get the name, description, and image for the NFT.

Lets deploy a new version of the `Crypto Devs` contract with this new API route as your `METADATA_URL`

Open your `hardhat-tutorial/constants` folder and inside your `index.js` file, replace "https://nft-collection-sneh1999.vercel.app/api/" with the domain which you saved to notepad and add "/api/" to its end.

Save the file and open up a new terminal pointing to `hardhat-tutorial` folder and deploy a new contract

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

Save the new NFT contract address to a notepad as we will need it later. Open up "my-app/constants" folder and inside the `index.js` file replace the old NFT contract address with the new one

Push all the code to github and wait for Vercel to deploy the new code. After vercel has deployed your code, open up your website and mint an NFT. After your transaction gets successful, in your browser open up this link by replacing `your-nft-contract-address` with the address of your NFT contract (https://testnets.opensea.io/assets/goerli/your-nft-contract-address/1)

Your NFT is now available on Opensea 🚀 🥳 Share your Opensea 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

LearnWeb3

Product

- Dashboard
- Courses
- Community
- Blog

Company

- About
- Work With Us
- We're Hiring
- Buy us a coffee

Stay up to date

Your email address