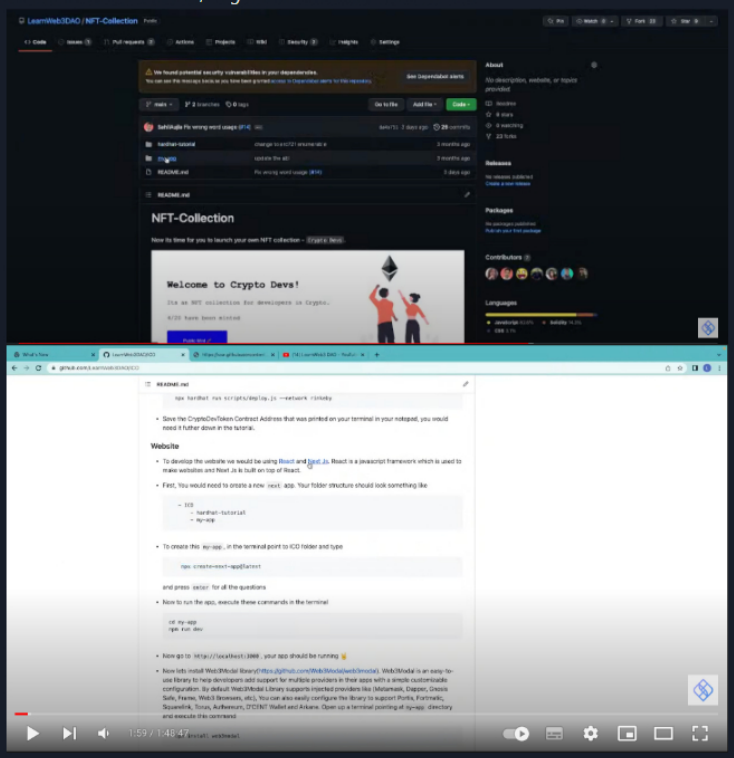
Lesson Type: Practical | Estimated Time: 6-10 hours | Current Score: 0%

Launch your own Initial Coin Offering

Now it's time for you to launch a token for `Crypto Devs`. Let's call the token Crypto Dev Token. We'll give this token out for free to all our NFT holders, and let other people buy them for ETH.

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!



Note: The video tutorial is based on Rinkeby testnet, but the text tutorial is based on Goerli for future support. Please use the Goerli network.

Build

Requirements

- There should be a max of `10,000 CD` tokens.
- Every `Crypto Dev` NFT holder should get 10 tokens for free but they would have to pay the gas fees.
- The price of one CD at the time of ICO should be `0.001 ether`.
- There should be a website that users can visit for the ICO.

Let's start building 🚀

Prerequisites

- You must have completed the `NFT Collection` tutorial from earlier.

Theory

- What is an ERC20?
  - ERC-20 is a technical standard; it is used for all smart contracts on the Ethereum blockchain for token implementation and provides a list of rules that all Ethereum-based tokens must follow.
  - Please look at all the ERC20 **functions** before moving ahead.

Build

Smart Contract

To build the smart contract we will 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 contracts, deploy them, run tests, and debug your code.

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

```
mkdir ICO
cd ICO
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, install `@openzeppelin/contracts` as we will be importing **Openzeppelin's ERC20 Contract** and **Openzeppelin's Ownable Contract** in our `CryptoDevToken` contract.

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

We need to call the `CryptoDevs Contract` that you deployed for the previous level to check for owners of CryptoDev NFT's. As we only need to call `tokenOfOwnerByIndex` and `balanceOf` methods, we can create an interface for `CryptoDevs contract` with only these two functions. This way we save `gas` as we do not need to inherit and deploy the entire `CryptoDevs Contract`, but only a part of it.

Create a new file inside the `contracts` directory and call it `ICryptoDevs.sol`. Add the following lines:

Create a new file inside the `contracts` directory and call it `ICryptoDevs.sol`. Add the following lines:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface ICryptoDevs {
    /**
     * @dev Returns a token ID owned by `owner` at a given `index` of its token list.
     * Use along with {balanceOf} to enumerate all of ``owner``'s tokens.
     */
    function tokenOfOwnerByIndex(address owner, uint256 index)
        external
        view
        returns (uint256 tokenId);

    /**
     * @dev Returns the number of tokens in ``owner``'s account.
     */
    function balanceOf(address owner) external view returns (uint256 balance);
}
```

Create another file inside the `contracts` directory and call it `CryptoDevToken.sol`. Add the following lines:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "./ICryptoDevs.sol";

contract CryptoDevToken is ERC20, Ownable {
    // Price of one Crypto Dev token
    uint256 public constant tokenPrice = 0.001 ether;
    // Each NFT would give the user 10 tokens
    // It needs to be represented as 10 * (10 ** 18) as ERC20 tokens are represented by the smallest denomination po
    // By default, ERC20 tokens have the smallest denomination of 10^(-18). This means, having a balance of (1)
    // is actually equal to (10 ^ -18) tokens.
    // Owning 1 full token is equivalent to owning (10^18) tokens when you account for the decimal places.
    // More information on this can be found in the Freshman Track Cryptocurrency tutorial.
    uint256 public constant tokensPerNFT = 10 * 10**18;
    // the max total supply is 10000 for Crypto Dev Tokens
    uint256 public constant maxTotalSupply = 10000 * 10**18;
    // CryptoDevsNFT contract instance
    ICryptoDevs CryptoDevsNFT;
    // Mapping to keep track of which tokenIds have been claimed
    mapping(uint256 => bool) public tokenIdsClaimed;

    constructor(address _cryptoDevsContract) ERC20("Crypto Dev Token", "CD") {
        CryptoDevsNFT = ICryptoDevs(_cryptoDevsContract);
    }

    /**
     * @dev Mints `amount` number of CryptoDevTokens
     * Requirements:
     * - `msg.value` should be equal or greater than the tokenPrice * amount
     */
    function mint(uint256 amount) public payable {
        // the value of ether that should be equal or greater than tokenPrice * amount;
        uint256 _requiredAmount = tokenPrice * amount;
        require(msg.value >= _requiredAmount, "Ether sent is incorrect");
        // total tokens + amount <= 10000, otherwise revert the transaction
        uint256 amountWithDecimals = amount * 10**18;
        require(
            (totalSupply() + amountWithDecimals) <= maxTotalSupply,
            "Exceeds the max total supply available."
        );
        // call the internal function from Openzeppelin's ERC20 contract
        _mint(msg.sender, amountWithDecimals);
    }

    /**
     * @dev Mints tokens based on the number of NFT's held by the sender
     * Requirements:
     * balance of Crypto Dev NFT's owned by the sender should be greater than 0
     * Tokens should have not been claimed for all the NFTs owned by the sender
     */
    function claim() public {
        address sender = msg.sender;
        // Get the number of CryptoDev NFT's held by a given sender address
        uint256 balance = CryptoDevsNFT.balanceOf(sender);
        // If the balance is zero, revert the transaction
        require(balance > 0, "You dont own any Crypto Dev NFT's");
        // amount keeps track of number of unclaimed tokenIds
```

```
        uint256 amount = 0;
        // Loop over the balance and get the token ID owned by `sender` at a given `index` of its token list.
        for (uint256 i = 0; i < balance; i++) {
            uint256 tokenId = CryptoDevsNFT.tokenOfOwnerByIndex(sender, i);
            // if the tokenId has not been claimed, increase the amount
            if (!tokenIdsClaimed[tokenId]) {
                amount += 1;
                tokenIdsClaimed[tokenId] = true;
            }
        }
        // If all the token Ids have been claimed, revert the transaction;
        require(amount > 0, "You have already claimed all the tokens");
        // call the internal function from Openzeppelin's ERC20 contract
        // Mint (amount * 10) tokens for each NFT
        _mint(msg.sender, amount * tokensPerNFT);
    }

    /**
      * @dev withdraws all ETH and tokens sent to the contract
      * Requirements:
      * wallet connected must be owner's address
      */
    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 we will install the `dotenv` package to be able to import the env file and use it in our config. Open up a terminal pointing to the `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"
```

Let's deploy the contract to the `goerli` network. Create a new file (or replace the existing default file) named `deploy.js` under the `scripts` folder.

Now we will write some code to deploy the contract in `deploy.js` file.

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

async function main() {
  // Address of the Crypto Devs NFT contract that you deployed in the previous module
```

```
  const cryptoDevsNFTContract = CRYPTO_DEVS_NFT_CONTRACT_ADDRESS;

  /*
    A ContractFactory in ethers.js is an abstraction used to deploy new smart contracts,
    so cryptoDevsTokenContract here is a factory for instances of our CryptoDevToken contract.
    */
  const cryptoDevsTokenContract = await ethers.getContractFactory(
    "CryptoDevToken"
  );

  // deploy the contract
  const deployedCryptoDevsTokenContract = await cryptoDevsTokenContract.deploy(
    cryptoDevsNFTContract
  );

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

You can see that the `deploy.js` file requires a constant. Let's create a `constants` folder under `hardhat-tutorial` folder. Inside the `constants` folder create a new file named `index.js` and add the following lines to it.

Replace "address-of-the-nft-contract" with the address of the `CryptoDevs.sol` that you deployed in the previous module(`NFT-Collection`):

```
// Address of the NFT Contract that you deployed
const CRYPTO_DEVS_NFT_CONTRACT_ADDRESS = "address-of-the-nft-contract";

module.exports = { CRYPTO_DEVS_NFT_CONTRACT_ADDRESS };
```

Now open the `hardhat.config.js` file, we'll 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 to the `hardhat-tutorial` directory and execute this command:

```
npx hardhat compile
```

Execute this command in the same directory to deploy the contract:

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

Save the CryptoDevToken Contract Address that was printed to your terminal in your notepad. You will need it later in the tutorial.

Website

To develop the website we will 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. You first need to create a new `next` app. Your folder structure should look something like this:

```
- ICO
       - hardhat-tutorial
       - my-app
```

To create this `my-app`, in the terminal point to the `ICO` 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 let's install the 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 to `my-app` directory and execute this command:

```
npm install web3modal
```

In the same terminal also install `ethers.js`:

```
npm install ethers
```

In the `public` folder, download the following image (https://github.com/LearnWeb3DAO/NFT-Collection/tree/main/my-app/public/cryptodevs/0.svg). Make sure that the name of the downloaded image is `0.svg`.

Now go to styles folder and replace all the contents of `Home.modules.css` file with the following code. This will 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%;
}

.input {
  width: 200px;
  height: 100%;
  padding: 1%;
  margin-bottom: 2%;
  box-shadow: 0 0 15px 4px rgba(0, 0, 0, 0.06);
```

```
  border-radius: 10px;
}

.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: 5px;
  width: 100px;
  cursor: pointer;
  margin-bottom: 2%;
}
@media (max-width: 1000px) {
  .main {
    width: 100%;
    flex-direction: column;
    justify-content: center;
    align-items: center;
  }
}
```

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

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

export default function Home() {
  // Create a BigNumber `0`
  const zero = BigNumber.from(0);
  // walletConnected keeps 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);
  // tokensToBeClaimed keeps track of the number of tokens that can be claimed
  // based on the Crypto Dev NFT's held by the user for which they havent claimed the tokens
  const [tokensToBeClaimed, setTokensToBeClaimed] = useState(zero);
  // balanceOfCryptoDevTokens keeps track of number of Crypto Dev tokens owned by an address
  const [balanceOfCryptoDevTokens, setBalanceOfCryptoDevTokens] =
    useState(zero);
  // amount of the tokens that the user wants to mint
  const [tokenAmount, setTokenAmount] = useState(zero);
  // tokensMinted is the total number of tokens that have been minted till now out of 10000(max total supply)
  const [tokensMinted, setTokensMinted] = useState(zero);
  // isOwner gets the owner of the contract through the signed address
  const [isOwner, setIsOwner] = useState(false);
  // Create a reference to the Web3 Modal (used for connecting to Metamask) which persists as long as the page is open
  const web3ModalRef = useRef();

  /**
   * getTokensToBeClaimed: checks the balance of tokens that can be claimed by the user
   */
  const getTokensToBeClaimed = 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();
      // Create an instance of NFT Contract
      const nftContract = new Contract(
        NFT_CONTRACT_ADDRESS,
        NFT_CONTRACT_ABI,
```

```
      provider
    );
    // Create an instance of tokenContract
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      provider
    );
    // 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();
    // call the balanceOf from the NFT contract to get the number of NFT's held by the user
    const balance = await nftContract.balanceOf(address);
    // balance is a Big number and thus we would compare it with Big number `zero`
    if (balance === zero) {
      setTokensToBeClaimed(zero);
    } else {
      // amount keeps track of the number of unclaimed tokens
      var amount = 0;
      // For all the NFT's, check if the tokens have already been claimed
      // Only increase the amount if the tokens have not been claimed
      // for a an NFT(for a given tokenId)
      for (var i = 0; i < balance; i++) {
        const tokenId = await nftContract.tokenOfOwnerByIndex(address, i);
        const claimed = await tokenContract.tokenIdsClaimed(tokenId);
        if (!claimed) {
          amount++;
        }
      }
      //tokensToBeClaimed has been initialized to a Big Number, thus we would convert amount
      // to a big number and then set its value
      setTokensToBeClaimed(BigNumber.from(amount));
    }
  } catch (err) {
    console.error(err);
    setTokensToBeClaimed(zero);
  }
};

/**
 * getBalanceOfCryptoDevTokens: checks the balance of Crypto Dev Tokens's held by an address
 */
const getBalanceOfCryptoDevTokens = 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();
    // Create an instace of token contract
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      provider
    );
    // 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();
    // call the balanceOf from the token contract to get the number of tokens held by the user
    const balance = await tokenContract.balanceOf(address);
    // balance is already a big number, so we dont need to convert it before setting it
    setBalanceOfCryptoDevTokens(balance);
  } catch (err) {
    console.error(err);
    setBalanceOfCryptoDevTokens(zero);
  }
};

/**
 * mintCryptoDevToken: mints `amount` number of tokens to a given address
 */
const mintCryptoDevToken = async (amount) => {
  try {
    // We need a Signer here since this is a 'write' transaction.
    // Create an instance of tokenContract
    const signer = await getProviderOrSigner(true);
    // Create an instance of tokenContract
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      signer
    );
    // Each token is of `0.001 ether`. The value we need to send is `0.001 * amount`
    const value = 0.001 * amount;
    const tx = await tokenContract.mint(amount, {
      // value signifies the cost of one crypto dev token which is "0.001" eth
```

```
      // value signifies the cost of one crypto dev token which is "0.001" eth.
      // We are parsing `0.001` string to ether using the utils library from ethers.js
      value: utils.parseEther(value.toString()),
    });
    setLoading(true);
    // wait for the transaction to get mined
    await tx.wait();
    setLoading(false);
    window.alert("Sucessfully minted Crypto Dev Tokens");
    await getBalanceOfCryptoDevTokens();
    await getTotalTokensMinted();
    await getTokensToBeClaimed();
  } catch (err) {
    console.error(err);
  }
};

/**
 * claimCryptoDevTokens: Helps the user claim Crypto Dev Tokens
 */
const claimCryptoDevTokens = async () => {
  try {
    // We need a Signer here since this is a 'write' transaction.
    // Create an instance of tokenContract
    const signer = await getProviderOrSigner(true);
    // Create an instance of tokenContract
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      signer
    );
    const tx = await tokenContract.claim();
    setLoading(true);
    // wait for the transaction to get mined
    await tx.wait();
    setLoading(false);
    window.alert("Sucessfully claimed Crypto Dev Tokens");
    await getBalanceOfCryptoDevTokens();
    await getTotalTokensMinted();
    await getTokensToBeClaimed();
  } catch (err) {
    console.error(err);
  }
};

/**
 * getTotalTokensMinted: Retrieves how many tokens have been minted till now
 * out of the total supply
 */
const getTotalTokensMinted = 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();
    // Create an instance of token contract
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      provider
    );
    // Get all the tokens that have been minted
    const _tokensMinted = await tokenContract.totalSupply();
    setTokensMinted(_tokensMinted);
  } catch (err) {
    console.error(err);
  }
};

/**
 * getOwner: gets the contract owner by connected address
 */
const getOwner = async () => {
  try {
    const provider = await getProviderOrSigner();
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      provider
    );
    // call the owner function from the contract
    const _owner = await tokenContract.owner();
    // we get signer to extract address of currently connected Metamask account
    const signer = await getProviderOrSigner(true);
    // Get the address associated to signer which is connected to Metamask
    const address = await signer.getAddress();
    if (address.toLowerCase() === _owner.toLowerCase()) {
```

```
      setIsOwner(true);
    }
  } catch (err) {
    console.error(err.message);
  }
};

/**
 * withdrawCoins: withdraws ether and tokens by calling
 * the withdraw function in the contract
 */
const withdrawCoins = async () => {
  try {
    const signer = await getProviderOrSigner(true);
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      signer
    );

    const tx = await tokenContract.withdraw();
    setLoading(true);
    await tx.wait();
    setLoading(false);
    await getOwner();
  } 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;
};

/*
      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);
  }
};

// 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",
```

```
      network: "goerli",
      providerOptions: {},
      disableInjectedProvider: false,
    });
    connectWallet();
    getTotalTokensMinted();
    getBalanceOfCryptoDevTokens();
    getTokensToBeClaimed();
    withdrawCoins();
  }
}, [walletConnected]);

/*
      renderButton: Returns a button based on the state of the dapp
    */
const renderButton = () => {
  // If we are currently waiting for something, return a loading button
  if (loading) {
    return (
      <div>
        <button className={styles.button}>Loading...</button>
      </div>
    );
  }
  // if owner is connected, withdrawCoins() is called
  if (walletConnected && isOwner) {
    return (
      <div>
        <button className={styles.button1} onClick={withdrawCoins}>
          Withdraw Coins
        </button>
      </div>
    );
  }
  // If tokens to be claimed are greater than 0, Return a claim button
  if (tokensToBeClaimed > 0) {
    return (
      <div>
        <div className={styles.description}>
          {tokensToBeClaimed * 10} Tokens can be claimed!
        </div>
        <button className={styles.button} onClick={claimCryptoDevTokens}>
          Claim Tokens
        </button>
      </div>
    );
  }
  // If user doesn't have any tokens to claim, show the mint button
  return (
    <div style={{ display: "flex-col" }}>
      <div>
        <input
          type="number"
          placeholder="Amount of Tokens"
          // BigNumber.from converts the `e.target.value` to a BigNumber
          onChange={(e) => setTokenAmount(BigNumber.from(e.target.value))}
          className={styles.input}
        />
      </div>

      <button
        className={styles.button}
        disabled={!(tokenAmount > 0)}
        onClick={() => mintCryptoDevToken(tokenAmount)}
      >
        Mint Tokens
      </button>
    </div>
  );
};

return (
  <div>
    <Head>
      <title>Crypto Devs</title>
      <meta name="description" content="ICO-Dapp" />
      <link rel="icon" href="/favicon.ico" />
    </Head>
    <div className={styles.main}>
      <div>
        <h1 className={styles.title}>Welcome to Crypto Devs ICO!</h1>
        <div className={styles.description}>
          You can claim or mint Crypto Dev tokens here
        </div>
        {walletConnected ? (
          <div>
```

```
              <div className={styles.description}>
                {/* Format Ether helps us in converting a BigNumber to string */}
                You have minted {utils.formatEther(balanceOfCryptoDevTokens)} Crypto
                Dev Tokens
              </div>
              <div className={styles.description}>
                {/* Format Ether helps us in converting a BigNumber to string */}
                Overall {utils.formatEther(tokensMinted)}/10000 have been minted!!!
              </div>
              {renderButton()}
            </div>
          ) : (
            <button onClick={connectWallet} className={styles.button}>
              Connect your wallet
            </button>
          )}
        </div>
        <div>
          <img className={styles.image} src="./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 called `index.js` and paste the following code:

```
export const NFT_CONTRACT_ABI = "abi-of-your-nft-contract";
export const NFT_CONTRACT_ADDRESS = "address-of-your-nft-contract";
export const TOKEN_CONTRACT_ABI = "abi-of-your-token-contract";
export const TOKEN_CONTRACT_ADDRESS = "address-of-your-token-contract";
```

- Replace `"abi-of-your-nft-contract"` with the abi of the NFT contract that you deployed in the last tutorial.
- Replace `"address-of-your-nft-contract"` with the address of the NFT contract that you deployed in your previous tutorial.
- Replace `"abi-of-your-token-contract"` by the abi of the token contract. To get the abi of the Token contract, go to `hardhat-tutorial/artifacts/contracts/CryptoDevToken.sol` and then from `CryptoDevToken.json` file get the array marked under the `"abi"` key.
- Replace `"address-of-your-token-contract"` with the address of the token contract that you saved to your notepad earlier in the tutorial.

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

```
npm run dev
```

Your ICO dapp should now work without errors 🚀

Push to Github

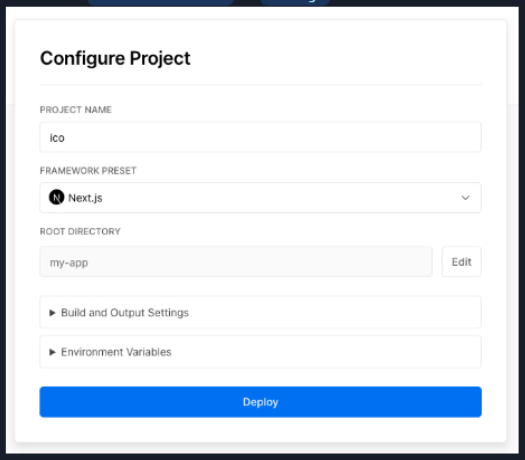
Make sure to push all your **code to Github before proceeding to the next step**.

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 ICO dApp 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 Preset` as `Next.js`.



- Click `Deploy`
- Now you can see your deployed website by going to your dashboard, selecting your project, and copying the URL from there!

CONGRATULATIONS! You're all done!

Hopefully you enjoyed this tutorial. Don't forget to share your ICO website in the `#showcase` channel on Discord :D

Submit Practical

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

Ethereum Rinkeby | 0x... | Submit