Lesson Type: Practical | Estimated Time: 4-8 hours | Current Score: 0%

Build a whitelist dApp for your upcoming NFT Collection

You are launching your NFT collection named `Crypto Devs`. You want to give your early supporters access to a whitelist for your collection, so here you are creating a whitelist dapp for `Crypto Devs`

Requirements

- Whitelist access should be given to the first `10` users for free who want to get in.
- There should be a website where people can go and enter into the whitelist.

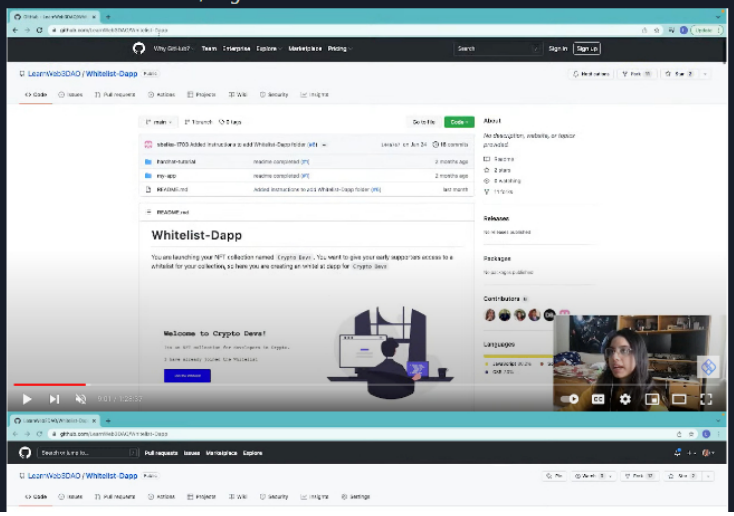
Lets start building 🚀

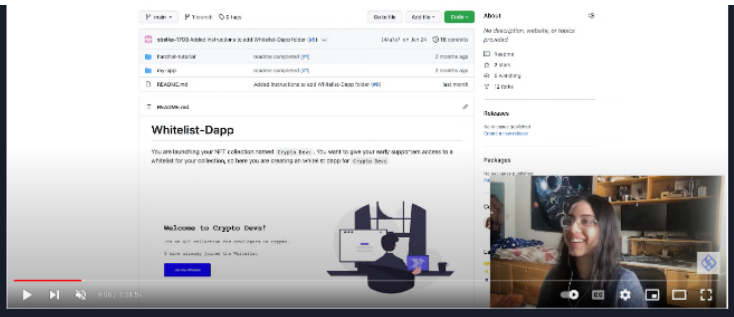
Prerequisites

- You can write code in JavaScript (Freshman Track - Introduction to Programming)
- Have set up a Metamask Wallet (Freshman Track - Setting up a crypto wallet)
- Your computer has Node.js installed. If not download from here

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!





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 contract, deploy them, run tests, and debug your code.

First, you need to create a Whitelist-Daap folder where the Hardhat project and your Next.js app will later go. Open up a terminal and execute these commands

```
mkdir Whitelist-Dapp
cd Whitelist-Dapp
```

Then, in Whitelist-Dapp folder, we will set up a Hardhat project

```
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.

Once your project is set up, start by creating a new file inside the `contracts` directory called `Whitelist.sol`.

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


contract Whitelist {

    // Max number of whitelisted addresses allowed
    uint8 public maxWhitelistedAddresses;

    // Create a mapping of whitelistedAddresses
    // if an address is whitelisted, we would set it to true, it is false by default for all other addresses.
    mapping(address => bool) public whitelistedAddresses;

    // numAddressesWhitelisted would be used to keep track of how many addresses have been whitelisted
    // NOTE: Don't change this variable name, as it will be part of verification
    uint8 public numAddressesWhitelisted;

    // Setting the Max number of whitelisted addresses
    // User will put the value at the time of deployment
    constructor(uint8 _maxWhitelistedAddresses) {
        maxWhitelistedAddresses =  _maxWhitelistedAddresses;
    }

    /**
        addAddressToWhitelist - This function adds the address of the sender to the
        whitelist
     */
```

```
    function addAddressToWhitelist() public {
        // check if the user has already been whitelisted
        require(!whitelistedAddresses[msg.sender], "Sender has already been whitelisted");
        // check if the numAddressesWhitelisted < maxWhitelistedAddresses, if not then throw an error.
        require(numAddressesWhitelisted < maxWhitelistedAddresses, "More addresses cant be added, limit reached");
        // Add the address which called the function to the whitelistedAddress array
        whitelistedAddresses[msg.sender] = true;
        // Increase the number of whitelisted addresses
        numAddressesWhitelisted += 1;
    }

}
```

We will deploy the contract to the `goerli` network. Create a new file, or replace the 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");

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

  // here we deploy the contract
  const deployedWhitelistContract = await whitelistContract.deploy(10);
  // 10 is the Maximum number of whitelisted addresses allowed

  // Wait for it to finish deploying
  await deployedWhitelistContract.deployed();

  // print the address of the deployed contract
  console.log("Whitelist Contract Address:", deployedWhitelistContract.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 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"
```

Now we will 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 open the hardhat.config.js file, we would add 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: "0.8.9",
  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 Whitelist 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 will use **React** and **Next Js**. React is a javascript framework used to make websites and Next.js is a React framework that also allows writing backend APIs code along with the frontend, so you don't need two separate frontend and backend services.

First, You will need to create a new `next` app. To create this `next-app`, in the terminal point to Whitelist-Dapp folder and type

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

and press `enter` for all the questions

Your folder structure should look something like

```
- Whitelist-Dapp
    - hardhat-tutorial
    - my-app
```

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 easily allow their users to connect to your dApps with all sorts of different wallets. By default Web3Modal Library supports injected providers like (Metamask, Dapper, Gnosis Safe, Frame, Web3 Browsers, etc) and WalletConnect, You can also easily configure the library to support Portis, Fortmatic, Squarelink, Torus, Authereum, D'CENT Wallet and Arkane. (Here's a live example on **Codesandbox.io**)

Open up a terminal pointing at `my-app` directory and execute this command

```
npm install web3modal
```

In the same terminal also install `ethers.js`

In the same terminal also install ethers.js

```
npm install ethers
```

In your my-app/public folder, download **this image** and rename it to `crypto-devs.svg` Now go to your `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. Make sure you read about React and **React Hooks**, **React Hooks Tutorial** if you are not familiar with them.

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

export default function Home() {
  // walletConnected keep track of whether the user's wallet is connected or not
  const [walletConnected, setWalletConnected] = useState(false);
  // joinedWhitelist keeps track of whether the current metamask address has joined the Whitelist or not
  const [joinedWhitelist, setJoinedWhitelist] = useState(false);
  // loading is set to true when we are waiting for a transaction to get mined
  const [loading, setLoading] = useState(false);
  // numberOfWhitelisted tracks the number of addresses's whitelisted
  const [numberOfWhitelisted, setNumberOfWhitelisted] = useState(0);
```

```
// Create a reference to the Web3 Modal (used for connecting to Metamask) which persists as long as the page is ope
const web3ModalRef = useRef();

/**
 * 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;
};

/**
 * addAddressToWhitelist: Adds the current connected address to the whitelist
 */
const addAddressToWhitelist = 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 whitelistContract = new Contract(
      WHITELIST_CONTRACT_ADDRESS,
      abi,
      signer
    );
    // call the addAddressToWhitelist from the contract
    const tx = await whitelistContract.addAddressToWhitelist();
    setLoading(true);
    // wait for the transaction to get mined
    await tx.wait();
    setLoading(false);
    // get the updated number of addresses in the whitelist
    await getNumberOfWhitelisted();
    setJoinedWhitelist(true);
  } catch (err) {
    console.error(err);
  }
};

/**
 * getNumberOfWhitelisted:  gets the number of whitelisted addresses
 */
const getNumberOfWhitelisted = 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 whitelistContract = new Contract(
      WHITELIST_CONTRACT_ADDRESS,
      abi,
      provider
    );
    // call the numAddressesWhitelisted from the contract
    const _numberOfWhitelisted =
      await whitelistContract.numAddressesWhitelisted();
    setNumberOfWhitelisted(_numberOfWhitelisted);
  } catch (err) {
    console.error(err);
  }
```

```
};

/**
 * checkIfAddressInWhitelist: Checks if the address is in whitelist
 */
const checkIfAddressInWhitelist = async () => {
  try {
    // We will need the signer later to get the user's address
    // Even though it is a read transaction, since Signers are just special kinds of Providers,
    // We can use it in it's place
    const signer = await getProviderOrSigner(true);
    const whitelistContract = new Contract(
      WHITELIST_CONTRACT_ADDRESS,
      abi,
      signer
    );
    // Get the address associated to the signer which is connected to  MetaMask
    const address = await signer.getAddress();
    // call the whitelistedAddresses from the contract
    const _joinedWhitelist = await whitelistContract.whitelistedAddresses(
      address
    );
    setJoinedWhitelist(_joinedWhitelist);
  } 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);

    checkIfAddressInWhitelist();
    getNumberOfWhitelisted();
  } catch (err) {
    console.error(err);
  }
};

/*
  renderButton: Returns a button based on the state of the dapp
*/
const renderButton = () => {
  if (walletConnected) {
    if (joinedWhitelist) {
      return (
        <div className={styles.description}>
          Thanks for joining the Whitelist!
        </div>
      );
    } else if (loading) {
      return <button className={styles.button}>Loading...</button>;
    } else {
      return (
        <button onClick={addAddressToWhitelist} className={styles.button}>
          Join the Whitelist
        </button>
      );
    }
  } else {
    return (
      <button onClick={connectWallet} className={styles.button}>
        Connect your wallet
      </button>
    );
  }
};

// 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();
    }
  }, [walletConnected]);

  return (
    <div>
      <Head>
        <title>Whitelist Dapp</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}>
            {numberOfWhitelisted} have already joined the Whitelist
          </div>
          {renderButton()}
        </div>
        <div>
          <img className={styles.image} src="./crypto-devs.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.

```
export const WHITELIST_CONTRACT_ADDRESS = "YOUR_WHITELIST_CONTRACT_ADDRESS";
export const abi = YOUR_ABI;
```

Replace `"YOUR_WHITELIST_CONTRACT_ADDRESS"` with the address of the whitelist contract that you deployed. Replace `"YOUR_ABI"` with the ABI of your Whitelist Contract. To get the ABI for your contract, go to your `hardhat-tutorial/artifacts/contracts/Whitelist.sol` folder and from your `Whitelist.json` file get the array marked under the `"abi"` key (it will be. a huge array, close to 100 lines if not more).

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

```
npm run dev
```

Your whitelist dapp should now work without errors 🚀

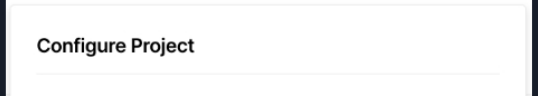
Push to github

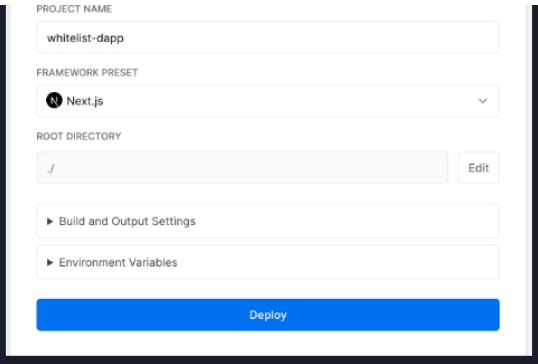
Make sure before proceeding you have **pushed all your code to github** :)

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 **Vercel** and sign in with your GitHub
- Then click on `New Project` button and then select your Whitelist 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 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!

Share your website in Discord :D

Verification

To verify this level, make sure you have whitelisted some addresses in your contract. Input your contract address into the Smart Contract Verification box, and select the test network you deployed on.

Submit Practical

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

Ethereum Rinkeby | 0x... | Submit