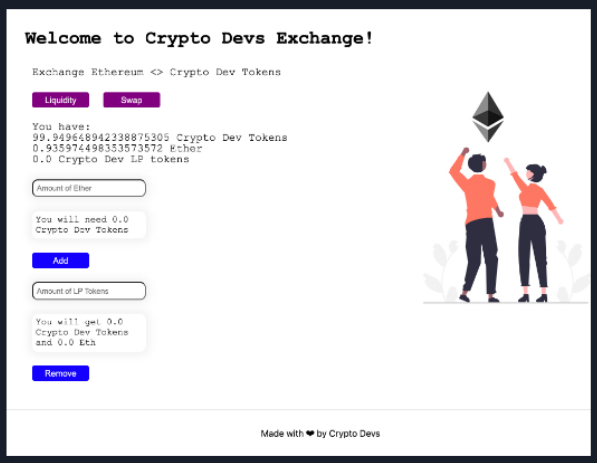
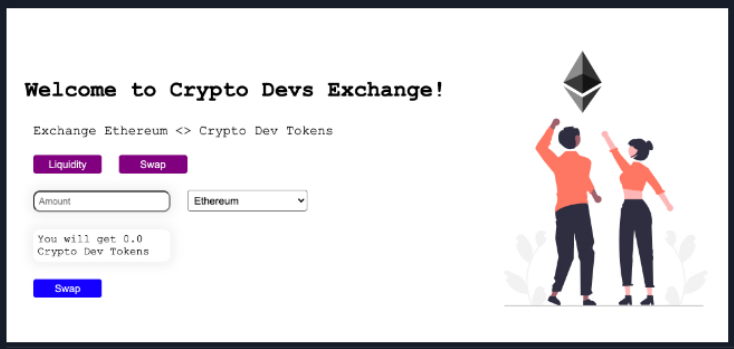
Lesson Type: Practical | Estimated Time: 8-12 hours | Current Score: 0%

Build your own Decentralized Exchange like Uniswap

Now its time for you to launch a DeFi Exchange for your `Crypto Dev` tokens





Requirements

- Build an exchange with only one asset pair (Eth / Crypto Dev)
- Your Decentralized Exchange should take a fees of `1%` on swaps
- When user adds liquidity, they should be given `Crypto Dev LP` tokens (Liquidity Provider tokens)
- CD LP tokens should be given proportional to the `Ether` user is willing to add to the liquidity

Lets start building 🚀

Prerequisites

- You have completed the `ICO Tutorial` from before
- You have completed the `Intro and deep dive into Decentralized Exchanges` tutorial from before
- You have completed the `Providers, Signers, ABIs, and Approval Flows` tutorial from before

Smart Contract

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 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 ERC20 Contract** in our `Exchange` contract

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

Create a new file inside the `contracts` directory called `Exchange.sol`. In this tutorial we would cover each part of the contract separately.

First lets start by importing `ERC20.sol` We need this because our Exchange needs to mint and create `Crypto Dev LP` tokens thats why it needs to inherit ERC20.sol

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract Exchange is ERC20 {
}
```

Now lets create a `constructor` for our contract

- It takes the address of the `_CryptoDevToken` that you deployed in the `ICO` tutorial as an input param
- It then checks if the address is a null address
- After all the checks, it assigns the value to the input param to the `cryptoDevTokenAddress` variable
- Constructor also sets the `name` and `symbol` for our `Crypto Dev LP` token

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract Exchange is ERC20 {

    address public cryptoDevTokenAddress;

    // Exchange is inheriting ERC20, because our exchange would keep track of Crypto Dev LP tokens
    constructor(address _CryptoDevtoken) ERC20("CryptoDev LP Token", "CDLP") {
        require(_CryptoDevtoken != address(0), "Token address passed is a null address");
        cryptoDevTokenAddress = _CryptoDevtoken;
    }
}
```

Time to create a function to get reserves of the `Eth` and `Crypto Dev` tokens held by the contract.

The ETH reserve would be equal to the balance of the contract and can be found using `address(this).balance` so lets just create a function only for getting reserves of the `Crypto Dev` tokens

We know that the `Crypto Dev Token` contract that we deployed is an ERC20. So we can just call the `balanceOf` to check the balance of `CryptoDev Tokens` that the contract holds

```
/**
* @dev Returns the amount of `Crypto Dev Tokens` held by the contract
*/
function getReserve() public view returns (uint) {
    return ERC20(cryptoDevTokenAddress).balanceOf(address(this));
}
```

Time to create an `addLiquidity` function which would add `liquidity` in the form of `ETH` and `Crypto Dev tokens` to our contract

If `cryptoDevTokenReserve` is zero it means that it is the first time someone is adding `Crypto Dev` tokens and `ETH` to the contract. In this case, we don't have to maintain a ratio between the tokens as we dont have any liquidity. So we accept any amount of tokens that user has sent with the initial call

If `cryptoDevTokenReserve` is not zero, then we have to make sure that when someone adds the liquidity it does not impact the price which the market currently has. To ensure this, we maintain a ratio which has to remain constant. The ratio is `(cryptoDevTokenAmount user can add/cryptoDevTokenReserve in the contract) = (Eth Sent by the user/Eth Reserve in the contract)`.

This ratio decides how much `Crypto Dev` tokens user can supply given a certain amount of ETH. When user adds liquidity, we need to provide him with some `LP` tokens because we need to keep track of the amount of liquidity he has supplied to the contract. The amount of `LP` tokens that get minted to the user are proportional to the `ETH` supplied by the user.

In the initial liquidity case, when there is no liquidity: The amount of `LP` tokens that would be minted to the user is equal to the `ETH` balance of the contract (because balance is equal to the `ETH` sent by the user in the `addLiquidity` call)

When there is already liquidity in the contract, the amount of `LP` tokens that get minted is based on a ratio. The ratio is `(LP tokens to be sent to the user (liquidity) / totalSupply of LP tokens in contract) = (Eth sent by the user) / (Eth reserve in the contract)`

> If this seems a bit confusing to you, open up the previous theory lesson again and review the mathematics behind the xy = k price function and LP tokens.

```
/**
* @dev Adds liquidity to the exchange.
*/
function addLiquidity(uint _amount) public payable returns (uint) {
    uint liquidity;
    uint ethBalance = address(this).balance;
    uint cryptoDevTokenReserve = getReserve();
    ERC20 cryptoDevToken = ERC20(cryptoDevTokenAddress);
    /*
        If the reserve is empty, intake any user supplied value for
        `Ether` and `Crypto Dev` tokens because there is no ratio currently
    */
    if(cryptoDevTokenReserve == 0) {
        // Transfer the `cryptoDevToken` from the user's account to the contract
        cryptoDevToken.transferFrom(msg.sender, address(this), _amount);
        // Take the current ethBalance and mint `ethBalance` amount of LP tokens to the user.
        // `liquidity` provided is equal to `ethBalance` because this is the first time user
        // is adding `Eth` to the contract, so whatever `Eth` contract has is equal to the one supplied
        // by the user in the current `addLiquidity` call
        // `liquidity` tokens that need to be minted to the user on `addLiquidity` call should always be proportional
        // to the Eth specified by the user
        liquidity = ethBalance;
        _mint(msg.sender, liquidity);
        // _mint is ERC20.sol smart contract function to mint ERC20 tokens
    } else {
        /*
            If the reserve is not empty, intake any user supplied value for
            `Ether` and determine according to the ratio how many `Crypto Dev` tokens
            need to be supplied to prevent any large price impacts because of the additional
            liquidity
        */
        // EthReserve should be the current ethBalance subtracted by the value of ether sent by the user
```

```
        // in the current `addLiquidity` call
        uint ethReserve =  ethBalance - msg.value;
        // Ratio should always be maintained so that there are no major price impacts when adding liquidity
        // Ratio here is -> (cryptoDevTokenAmount user can add/cryptoDevTokenReserve in the contract) = (Eth Sent by
        // So doing some maths, (cryptoDevTokenAmount user can add) = (Eth Sent by the user * cryptoDevTokenReserve /
        uint cryptoDevTokenAmount = (msg.value * cryptoDevTokenReserve)/(ethReserve);
        require(_amount >= cryptoDevTokenAmount, "Amount of tokens sent is less than the minimum tokens required");
        // transfer only (cryptoDevTokenAmount user can add) amount of `Crypto Dev tokens` from users account
        // to the contract
        cryptoDevToken.transferFrom(msg.sender, address(this), cryptoDevTokenAmount);
        // The amount of LP tokens that would be sent to the user should be proportional to the liquidity of
        // ether added by the user
        // Ratio here to be maintained is ->
        // (LP tokens to be sent to the user (liquidity)/ totalSupply of LP tokens in contract) = (Eth sent by the us
        // by some maths -> liquidity =  (totalSupply of LP tokens in contract * (Eth sent by the user))/(Eth reserve
        liquidity = (totalSupply() * msg.value)/ ethReserve;
        _mint(msg.sender, liquidity);
    }
     return liquidity;
}
```

Now lets create a function for removing liquidity from the contract.

The amount of ether that would be sent back to the user would be based on a ratio. The ratio is `(Eth sent back to the user) / (current Eth reserve) = (amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)`.

The amount of Crypto Dev tokens that would be sent back to the user would also be based on a ratio. The ratio is `(Crypto Dev sent back to the user) / (current Crypto Dev token reserve) = (amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)`.

The amount of `LP` tokens that user would use to remove liquidity would be burnt

```
/**
* @dev Returns the amount Eth/Crypto Dev tokens that would be returned to the user
* in the swap
*/
function removeLiquidity(uint _amount) public returns (uint , uint) {
    require(_amount > 0, "_amount should be greater than zero");
    uint ethReserve = address(this).balance;
    uint _totalSupply = totalSupply();
    // The amount of Eth that would be sent back to the user is based
    // on a ratio
    // Ratio is -> (Eth sent back to the user) / (current Eth reserve)
    // = (amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)
    // Then by some maths -> (Eth sent back to the user)
    // = (current Eth reserve * amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)
    uint ethAmount = (ethReserve * _amount)/ _totalSupply;
    // The amount of Crypto Dev token that would be sent back to the user is based
    // on a ratio
    // Ratio is -> (Crypto Dev sent back to the user) / (current Crypto Dev token reserve)
    // = (amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)
    // Then by some maths -> (Crypto Dev sent back to the user)
    // = (current Crypto Dev token reserve * amount of LP tokens that user wants to withdraw) / (total supply of LP t
    uint cryptoDevTokenAmount = (getReserve() * _amount)/ _totalSupply;
    // Burn the sent LP tokens from the user's wallet because they are already sent to
    // remove liquidity
    _burn(msg.sender, _amount);
    // Transfer `ethAmount` of Eth from the contract to the user's wallet
    payable(msg.sender).transfer(ethAmount);
    // Transfer `cryptoDevTokenAmount` of Crypto Dev tokens from the contract to the user's wallet
    ERC20(cryptoDevTokenAddress).transfer(msg.sender, cryptoDevTokenAmount);
    return (ethAmount, cryptoDevTokenAmount);
}
```

Next lets implement the swap functionality

Swap would go two ways. One way would be `Eth` to `Crypto Dev` tokens and other would be `Crypto Dev` to `Eth`

Its important for us to determine: Given `x` amount of `Eth`/`Crypto Dev` token sent by the user, how many `Eth`/`Crypto Dev` tokens would he receive from the swap?

So let's create a function which calculates this:

- We will charge `1%`. This means the amount of input tokens with fees would equal `Input amount with fees = (input amount - (1*(input amount)/100)) = ((input amount)*99)/100`

- We need to follow the concept of `XY = K` curve
- We need to make sure `(x + Δx) * (y - Δy) = x * y`, so the final formula is `Δy = (y * Δx) / (x + Δx)`;
- `Δy` in our case is `tokens to be received`, `Δx = ((input amount)*99)/100`, `x` = Input Reserve, `y` = Output Reserve
- Input Reserve and Ouput Reserve would depend on which swap we are implementing. `ETH` to `Crypto Dev` token or vice versa

```
/**
* @dev Returns the amount Eth/Crypto Dev tokens that would be returned to the user
* in the swap
*/
function getAmountOfTokens(
    uint256 inputAmount,
    uint256 inputReserve,
    uint256 outputReserve
) public pure returns (uint256) {
    require(inputReserve > 0 && outputReserve > 0, "invalid reserves");
    // We are charging a fee of `1%`
    // Input amount with fee = (input amount - (1*(input amount)/100)) = ((input amount)*99)/100
    uint256 inputAmountWithFee = inputAmount * 99;
    // Because we need to follow the concept of `XY = K` curve
    // We need to make sure (x + Δx) * (y - Δy) = x * y
    // So the final formula is Δy = (y * Δx) / (x + Δx)
    // Δy in our case is `tokens to be received`
    // Δx = ((input amount)*99)/100, x = inputReserve, y = outputReserve
    // So by putting the values in the formulae you can get the numerator and denominator
    uint256 numerator = inputAmountWithFee * outputReserve;
    uint256 denominator = (inputReserve * 100) + inputAmountWithFee;
    return numerator / denominator;
}
```

Now lets implement a function to swap `ETH` for `Crypto Dev` tokens

```
/**
* @dev Swaps Eth for CryptoDev Tokens
*/
function ethToCryptoDevToken(uint _minTokens) public payable {
    uint256 tokenReserve = getReserve();
    // call the `getAmountOfTokens` to get the amount of Crypto Dev tokens
    // that would be returned to the user after the swap
    // Notice that the `inputReserve` we are sending is equal to
    // `address(this).balance - msg.value` instead of just `address(this).balance`
    // because `address(this).balance` already contains the `msg.value` user has sent in the given call
    // so we need to subtract it to get the actual input reserve
    uint256 tokensBought = getAmountOfTokens(
        msg.value,
        address(this).balance - msg.value,
        tokenReserve
    );

    require(tokensBought >= _minTokens, "insufficient output amount");
    // Transfer the `Crypto Dev` tokens to the user
    ERC20(cryptoDevTokenAddress).transfer(msg.sender, tokensBought);
}
```

Now lets implement a function to swap `Crypto Dev` tokens to `ETH`

```
/**
* @dev Swaps CryptoDev Tokens for Eth
*/
function cryptoDevTokenToEth(uint _tokensSold, uint _minEth) public {
    uint256 tokenReserve = getReserve();
    // call the `getAmountOfTokens` to get the amount of Eth
    // that would be returned to the user after the swap
    uint256 ethBought = getAmountOfTokens(
        _tokensSold,
        tokenReserve,
        address(this).balance
    );
    require(ethBought >= _minEth, "insufficient output amount");
    // Transfer `Crypto Dev` tokens from the user's address to the contract
    ERC20(cryptoDevTokenAddress).transferFrom(
        msg.sender,
        address(this),
        _tokensSold
    );
    // send the `ethBought` to the user from the contract
    payable(msg.sender).transfer(ethBought);
}
```

Your final contract should look like this:

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract Exchange is ERC20 {

    address public cryptoDevTokenAddress;

    // Exchange is inheriting ERC20, because our exchange would keep track of Crypto Dev LP tokens
    constructor(address _CryptoDevtoken) ERC20("CryptoDev LP Token", "CDLP") {
        require(_CryptoDevtoken != address(0), "Token address passed is a null address");
        cryptoDevTokenAddress = _CryptoDevtoken;
    }

    /**
    * @dev Returns the amount of `Crypto Dev Tokens` held by the contract
    */
    function getReserve() public view returns (uint) {
        return ERC20(cryptoDevTokenAddress).balanceOf(address(this));
    }

    /**
    * @dev Adds liquidity to the exchange.
    */
    function addLiquidity(uint _amount) public payable returns (uint) {
        uint liquidity;
        uint ethBalance = address(this).balance;
        uint cryptoDevTokenReserve = getReserve();
        ERC20 cryptoDevToken = ERC20(cryptoDevTokenAddress);
        /*
            If the reserve is empty, intake any user supplied value for
            `Ether` and `Crypto Dev` tokens because there is no ratio currently
        */
        if(cryptoDevTokenReserve == 0) {
            // Transfer the `cryptoDevToken` address from the user's account to the contract
            cryptoDevToken.transferFrom(msg.sender, address(this), _amount);
            // Take the current ethBalance and mint `ethBalance` amount of LP tokens to the user.
            // `liquidity` provided is equal to `ethBalance` because this is the first time user
            // is adding `Eth` to the contract, so whatever `Eth` contract has is equal to the one supplied
            // by the user in the current `addLiquidity` call
            // `liquidity` tokens that need to be minted to the user on `addLiquidity` call should always be proportio
            // to the Eth specified by the user
            liquidity = ethBalance;
            _mint(msg.sender, liquidity);
        } else {
            /*
                If the reserve is not empty, intake any user supplied value for
                `Ether` and determine according to the ratio how many `Crypto Dev` tokens
                need to be supplied to prevent any large price impacts because of the additional
                liquidity
            */
            // EthReserve should be the current ethBalance subtracted by the value of ether sent by the user
            // in the current `addLiquidity` call
            uint ethReserve =  ethBalance - msg.value;
            // Ratio should always be maintained so that there are no major price impacts when adding liquidity
            // Ration here is -> (cryptoDevTokenAmount user can add/cryptoDevTokenReserve in the contract) = (Eth Sen
            // So doing some maths, (cryptoDevTokenAmount user can add) = (Eth Sent by the user * cryptoDevTokenReser
            uint cryptoDevTokenAmount = (msg.value * cryptoDevTokenReserve)/(ethReserve);
            require(_amount >= cryptoDevTokenAmount, "Amount of tokens sent is less than the minimum tokens required"
            // transfer only (cryptoDevTokenAmount user can add) amount of `Crypto Dev tokens` from users account
            // to the contract
            cryptoDevToken.transferFrom(msg.sender, address(this), cryptoDevTokenAmount);
            // The amount of LP tokens that would be sent to the user should be propotional to the liquidity of
            // ether added by the user
            // Ratio here to be maintained is ->
            // (lp tokens to be sent to the user (liquidity)/ totalSupply of LP tokens in contract) = (Eth sent by th
            // by some maths -> liquidity =  (totalSupply of LP tokens in contract * (Eth sent by the user))/(Eth res
            liquidity = (totalSupply() * msg.value)/ ethReserve;
            _mint(msg.sender, liquidity);
        }
         return liquidity;
    }

    /**
    * @dev Returns the amount Eth/Crypto Dev tokens that would be returned to the user
    * in the swap
    */
    function removeLiquidity(uint _amount) public returns (uint , uint) {
        require(_amount > 0, "_amount should be greater than zero");
```

```
        uint ethReserve = address(this).balance;
        uint _totalSupply = totalSupply();
        // The amount of Eth that would be sent back to the user is based
        // on a ratio
        // Ratio is -> (Eth sent back to the user/ Current Eth reserve)
        // = (amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)
        // Then by some maths -> (Eth sent back to the user)
        // = (current Eth reserve * amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)
        uint ethAmount = (ethReserve * _amount)/ _totalSupply;
        // The amount of Crypto Dev token that would be sent back to the user is based
        // on a ratio
        // Ratio is -> (Crypto Dev sent back to the user) / (current Crypto Dev token reserve)
        // = (amount of LP tokens that user wants to withdraw) / (total supply of LP tokens)
        // Then by some maths -> (Crypto Dev sent back to the user)
        // = (current Crypto Dev token reserve * amount of LP tokens that user wants to withdraw) / (total supply of L
        uint cryptoDevTokenAmount = (getReserve() * _amount)/ _totalSupply;
        // Burn the sent `LP` tokens from the user's wallet because they are already sent to
        // remove liquidity
        _burn(msg.sender, _amount);
        // Transfer `ethAmount` of Eth from the contract to the user's wallet
        payable(msg.sender).transfer(ethAmount);
        // Transfer `cryptoDevTokenAmount` of `Crypto Dev` tokens from the contract to the user's wallet
        ERC20(cryptoDevTokenAddress).transfer(msg.sender, cryptoDevTokenAmount);
        return (ethAmount, cryptoDevTokenAmount);
    }

    /**
    * @dev Returns the amount Eth/Crypto Dev tokens that would be returned to the user
    * in the swap
    */
    function getAmountOfTokens(
        uint256 inputAmount,
        uint256 inputReserve,
        uint256 outputReserve
    ) public pure returns (uint256) {
        require(inputReserve > 0 && outputReserve > 0, "invalid reserves");
        // We are charging a fee of `1%`
        // Input amount with fee = (input amount - (1*(input amount)/100)) = ((input amount)*99)/100
        uint256 inputAmountWithFee = inputAmount * 99;
        // Because we need to follow the concept of `XY = K` curve
        // We need to make sure (x + Δx) * (y - Δy) = x * y
        // So the final formula is Δy = (y * Δx) / (x + Δx)
        // Δy in our case is `tokens to be received`
        // Δx = ((input amount)*99)/100, x = inputReserve, y = outputReserve
        // So by putting the values in the formulae you can get the numerator and denominator
        uint256 numerator = inputAmountWithFee * outputReserve;
        uint256 denominator = (inputReserve * 100) + inputAmountWithFee;
        return numerator / denominator;
    }

    /**
    * @dev Swaps Eth for CryptoDev Tokens
    */
    function ethToCryptoDevToken(uint _minTokens) public payable {
        uint256 tokenReserve = getReserve();
        // call the `getAmountOfTokens` to get the amount of Crypto Dev tokens
        // that would be returned to the user after the swap
        // Notice that the `inputReserve` we are sending is equal to
        // `address(this).balance - msg.value` instead of just `address(this).balance`
        // because `address(this).balance` already contains the `msg.value` user has sent in the given call
        // so we need to subtract it to get the actual input reserve
        uint256 tokensBought = getAmountOfTokens(
            msg.value,
            address(this).balance - msg.value,
            tokenReserve
        );

        require(tokensBought >= _minTokens, "insufficient output amount");
        // Transfer the `Crypto Dev` tokens to the user
        ERC20(cryptoDevTokenAddress).transfer(msg.sender, tokensBought);
    }


    /**
    * @dev Swaps CryptoDev Tokens for Eth
    */
    function cryptoDevTokenToEth(uint _tokensSold, uint _minEth) public {
        uint256 tokenReserve = getReserve();
        // call the `getAmountOfTokens` to get the amount of Eth
        // that would be returned to the user after the swap
        uint256 ethBought = getAmountOfTokens(
            _tokensSold,
            tokenReserve,
            address(this).balance
        );
        require(ethBought >=  minEth, "insufficient output amount");
```

```
        // Transfer `Crypto Dev` tokens from the user's address to the contract
        ERC20(cryptoDevTokenAddress).transferFrom(
            msg.sender,
            address(this),
            _tokensSold
        );
        // send the `ethBought` to the user from the contract
        payable(msg.sender).transfer(ethBought);
    }
}
```

Let's now 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-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 also create a constants folder to keep track of any constants we have. Under the `hardhat-tutorial` folder create a new folder named `constants` and under the `constants` folder create a new file `index.js`

Inside the `index.js` file add the following lines of code. Remember to replace `ADDRESS-OF-CRYPTO-DEV-TOKEN` with the contract address of the `Crypto Dev` token contract that you deployed in the `ICO` tutorial

```
const CRYPTO_DEV_TOKEN_CONTRACT_ADDRESS = "ADDRESS-OF-CRYPTO-DEV-TOKEN";

module.exports = { CRYPTO_DEV_TOKEN_CONTRACT_ADDRESS };
```

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

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

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

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

  // here we deploy the contract
  const deployedExchangeContract = await exchangeContract.deploy(
    cryptoDevTokenAddress
  );
  await deployedExchangeContract.deployed();
```

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

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 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 Exchange Contract Address that was printed on your terminal in your notepad, you would need it further 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

```
- DeFi-Exchange
       - hardhat-tutorial
       - my-app
```

To create this `my-app`, in the terminal point to DeFi-Exchange 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 i ethers
```

In your public folder, download this **image** and rename it to `cryptodev.svg`.

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%;
}

.input {
  width: 200px;
  height: 100%;
  padding: 1%;
  margin: 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: 2%;
  font-size: 1.2rem;
}

.button {
  border-radius: 4px;
  background-color: purple;
  border: none;
  color: #ffffff;
  font-size: 15px;
  padding: 5px;
  width: 100px;
  cursor: pointer;
  margin: 2%;
}

.inputDiv {
  width: 200px;
  height: 100%;
  padding: 1%;
```

```
  margin: 2%;
  border: lightslategray;
  box-shadow: 0 0 15px 4px rgba(0, 0, 0, 0.06);
  border-radius: 10px;
}
.select {
  border-radius: 4px;
  font-size: 15px;
  padding: 5px;
  width: 175px;
  cursor: pointer;
  margin: 2%;
}

.button1 {
  border-radius: 4px;
  background-color: blue;
  border: none;
  color: #ffffff;
  font-size: 15px;
  padding: 5px;
  width: 100px;
  cursor: pointer;
  margin: 2%;
}
@media (max-width: 1000px) {
  .main {
    width: 100%;
    flex-direction: column;
    justify-content: center;
    align-items: center;
  }
}
```

Now lets create a constants folder to keep track of any constants we might have. Create a `constants` folder under `my-app` folder and inside the `constants` folder create a file names index.js Paste the following code.

- Replace `ABI-CRYPTO-DEV-TOKEN-CONTRACT` with the abi of the `Crypto Dev` token contract that you deployed in the ICO tutorial.
- Replace `ADDRESS-OF-CRYPTO-DEV-TOKEN-CONTRACT` with the address of the `Crypto Dev` token contract that you deployed in the ICO tutorial
- Replace `ABI-EXCHANGE-CONTRACT` with the abi of the Exchange Contract. To get the abi for your contract, go to your `hardhat-tutorial/artifacts/contracts/Exchange.sol` folder and from your `Exchange.json` file get the array marked under the `"abi"` key.
- Replace `ADDRESS-EXCHANGE-CONTRACT` with the address of the exchange contract that you deployed above and saved to your notepad

```
export const TOKEN_CONTRACT_ABI = "ABI-CRYPTO-DEV-TOKEN-CONTRACT";
export const TOKEN_CONTRACT_ADDRESS = "ADDRESS-OF-CRYPTO-DEV-TOKEN-CONTRACT";
export const EXCHANGE_CONTRACT_ABI = "ABI-EXCHANGE-CONTRACT";
export const EXCHANGE_CONTRACT_ADDRESS = "ADDRESS-EXCHANGE-CONTRACT";
```

Now we would create some utility files which would help us to better interact with the contract. Create a `utils` folder inside the `my-app` folder and inside the folder create 4 files: `addLiquidity.js`, `removeLiquidity.js`, `getAmounts.js`, and `swap.js`

Lets start by writing some code in `getAmounts.js`. This file is used to retrieve balances and reserves for assets

```
import { Contract } from "ethers";
import {
  EXCHANGE_CONTRACT_ABI,
  EXCHANGE_CONTRACT_ADDRESS,
  TOKEN_CONTRACT_ABI,
  TOKEN_CONTRACT_ADDRESS,
} from "../constants";

/**
 * getEtherBalance: Retrieves the ether balance of the user or the contract
```

```
 */
export const getEtherBalance = async (provider, address, contract = false) => {
  try {
    // If the caller has set the `contract` boolean to true, retrieve the balance of
    // ether in the `exchange contract`, if it is set to false, retrieve the balance
    // of the user's address
    if (contract) {
      const balance = await provider.getBalance(EXCHANGE_CONTRACT_ADDRESS);
      return balance;
    } else {
      const balance = await provider.getBalance(address);
      return balance;
    }
  } catch (err) {
    console.error(err);
    return 0;
  }
};

/**
 * getCDTokensBalance: Retrieves the Crypto Dev tokens in the account
 * of the provided `address`
 */
export const getCDTokensBalance = async (provider, address) => {
  try {
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      provider
    );
    const balanceOfCryptoDevTokens = await tokenContract.balanceOf(address);
    return balanceOfCryptoDevTokens;
  } catch (err) {
    console.error(err);
  }
};

/**
 * getLPTokensBalance: Retrieves the amount of LP tokens in the account
 * of the provided `address`
 */
export const getLPTokensBalance = async (provider, address) => {
  try {
    const exchangeContract = new Contract(
      EXCHANGE_CONTRACT_ADDRESS,
      EXCHANGE_CONTRACT_ABI,
      provider
    );
    const balanceOfLPTokens = await exchangeContract.balanceOf(address);
    return balanceOfLPTokens;
  } catch (err) {
    console.error(err);
  }
};

/**
 * getReserveOfCDTokens: Retrieves the amount of CD tokens in the
 * exchange contract address
 */
export const getReserveOfCDTokens = async (provider) => {
  try {
    const exchangeContract = new Contract(
      EXCHANGE_CONTRACT_ADDRESS,
      EXCHANGE_CONTRACT_ABI,
      provider
    );
    const reserve = await exchangeContract.getReserve();
    return reserve;
  } catch (err) {
    console.error(err);
  }
};
```

Lets now write some code for `addLiquidity.js`.

- `addLiquidity.js` has two functions `addLiquidity` and `calculateCD`
  - `addLiquidity` is used to call the `addLiquidity` function in the contract to add liquidity
  -

It also gets the `Crypto Dev` tokens approved for the contract by the user. The reason why `Crypto Dev` tokens need approval is because they are an ERC20 token. For the contract to withdraw an ERC20 from a user's account, it needs the approval from the user's account

-

`calculateCD` tells you for a given amount of `Eth`, how many `Crypto Dev` tokens can be added to the `liquidity`

-

We calculate this by maintaining a ratio. The ratio we follow is `(amount of Crypto Dev tokens to be added) / (Crypto Dev tokens balance) = (Eth that would be added) / (Eth reserve in the contract)`

-

So by maths we get `(amount of Crypto Dev tokens to be added) = (Eth that would be added * Crypto Dev tokens balance) / (Eth reserve in the contract)`

-

The ratio is needed so that adding liquidity doesn't largely impact the price

-

Note `tx.wait()` means we are waiting for the transaction to get mined

```
import { Contract, utils } from "ethers";
import {
  EXCHANGE_CONTRACT_ABI,
  EXCHANGE_CONTRACT_ADDRESS,
  TOKEN_CONTRACT_ABI,
  TOKEN_CONTRACT_ADDRESS,
} from "../constants";

/**
 * addLiquidity helps add liquidity to the exchange,
 * If the user is adding initial liquidity, user decides the ether and CD tokens he wants to add
 * to the exchange. If he is adding the liquidity after the initial liquidity has already been added
 * then we calculate the Crypto Dev tokens he can add, given the Eth he wants to add by keeping the ratios
 * constant
 */
export const addLiquidity = async (
  signer,
  addCDAmountWei,
  addEtherAmountWei
) => {
  try {
    // create a new instance of the token contract
    const tokenContract = new Contract(
      TOKEN_CONTRACT_ADDRESS,
      TOKEN_CONTRACT_ABI,
      signer
    );
    // create a new instance of the exchange contract
    const exchangeContract = new Contract(
      EXCHANGE_CONTRACT_ADDRESS,
      EXCHANGE_CONTRACT_ABI,
      signer
    );
    // Because CD tokens are an ERC20, user would need to give the contract allowance
    // to take the required number CD tokens out of his contract
    let tx = await tokenContract.approve(
      EXCHANGE_CONTRACT_ADDRESS,
      addCDAmountWei.toString()
    );
    await tx.wait();
    // After the contract has the approval, add the ether and cd tokens in the liquidity
    tx = await exchangeContract.addLiquidity(addCDAmountWei, {
      value: addEtherAmountWei,
    });
    await tx.wait();
  } catch (err) {
    console.error(err);
  }
};

/**
 * calculateCD calculates the CD tokens that need to be added to the liquidity
 * given `_addEtherAmountWei` amount of ether
 */
export const calculateCD = async (
```

```
  _addEther = "0",
  etherBalanceContract,
  cdTokenReserve
) => {
  // `_addEther` is a string, we need to convert it to a Bignumber before we can do our calculations
  // We do that using the `parseEther` function from `ethers.js`
  const _addEtherAmountWei = utils.parseEther(_addEther);

  // Ratio needs to be maintained when we add liquidty.
  // We need to let the user know for a specific amount of ether how many `CD` tokens
  // He can add so that the price impact is not large
  // The ratio we follow is (amount of Crypto Dev tokens to be added) / (Crypto Dev tokens balance) = (Eth that wou
  // So by maths we get (amount of Crypto Dev tokens to be added) = (Eth that would be added * Crypto Dev tokens ba

  const cryptoDevTokenAmount = _addEtherAmountWei
    .mul(cdTokenReserve)
    .div(etherBalanceContract);
  return cryptoDevTokenAmount;
};
```

Now add some code to `removeLiquidity.js`

- We have two functions here: One is `removeLiquidity` and the other is `getTokensAfterRemove`
- `removeLiquidity` calls the `removeLiquidity` function from the contract, to remove the amount of `LP` tokens specified by the user
- `getTokensAfterRemove` calulates the amount of `Ether` and `CD` tokens that would be sent back to the user after he removes a certain amount of `LP` tokens from the pool
- The amount of `Eth` that would be sent back to the user after he withdraws the `LP` token is calculated based on a ratio,
- Ratio is -> `(amount of Eth that would be sent back to the user / Eth reserve) = (LP tokens withdrawn) / (total supply of LP tokens)`
- By some maths we get -> `(amount of Eth that would be sent back to the user) = (Eth Reserve * LP tokens withdrawn) / (total supply of LP tokens)`
- Similarly we also maintain a ratio for the `CD` tokens, so here in our case
- Ratio is -> `(amount of CD tokens sent back to the user / CD Token reserve) = (LP tokens withdrawn) / (total supply of LP tokens)`
- Then `(amount of CD tokens sent back to the user) = (CD token reserve * LP tokens withdrawn) / (total supply of LP tokens)`

```
import { Contract, providers, utils, BigNumber } from "ethers";
import {
  EXCHANGE_CONTRACT_ABI,
  EXCHANGE_CONTRACT_ADDRESS,
} from "../constants";

/**
 * removeLiquidity: Removes the `removeLPTokensWei` amount of LP tokens from
 * liquidity and also the calculated amount of `ether` and `CD` tokens
 */
export const removeLiquidity = async (signer, removeLPTokensWei) => {
  // Create a new instance of the exchange contract
  const exchangeContract = new Contract(
    EXCHANGE_CONTRACT_ADDRESS,
```

```
    EXCHANGE_CONTRACT_ABI,
    signer
  );
  const tx = await exchangeContract.removeLiquidity(removeLPTokensWei);
  await tx.wait();
};

/**
 * getTokensAfterRemove: Calculates the amount of `Eth` and `CD` tokens
 * that would be returned back to user after he removes `removeLPTokenWei` amount
 * of LP tokens from the contract
 */
export const getTokensAfterRemove = async (
  provider,
  removeLPTokenWei,
  _ethBalance,
  cryptoDevTokenReserve
) => {
  try {
    // Create a new instance of the exchange contract
    const exchangeContract = new Contract(
      EXCHANGE_CONTRACT_ADDRESS,
      EXCHANGE_CONTRACT_ABI,
      provider
    );
    // Get the total supply of `Crypto Dev` LP tokens
    const _totalSupply = await exchangeContract.totalSupply();
    // Here we are using the BigNumber methods of multiplication and division
    // The amount of Eth that would be sent back to the user after he withdraws the LP token
    // is calculated based on a ratio,
    // Ratio is -> (amount of Eth that would be sent back to the user / Eth reserve) = (LP tokens withdrawn) / (tot
    // By some maths we get -> (amount of Eth that would be sent back to the user) = (Eth Reserve * LP tokens witho
    // Similarly we also maintain a ratio for the `CD` tokens, so here in our case
    // Ratio is -> (amount of CD tokens sent back to the user / CD Token reserve) = (LP tokens withdrawn) / (total
    // Then (amount of CD tokens sent back to the user) = (CD token reserve * LP tokens withdrawn) / (total supply
    const _removeEther = _ethBalance.mul(removeLPTokenWei).div(_totalSupply);
    const _removeCD = cryptoDevTokenReserve
      .mul(removeLPTokenWei)
      .div(_totalSupply);
    return {
      _removeEther,
      _removeCD,
    };
  } catch (err) {
    console.error(err);
  }
};
```

Now it's time to write code for `swap.js` our last `utils` file

- It has two functions `getAmountOfTokenReceivedFromSwap` and `swapTokens`
- `swapTokens` swaps certain amount of `Eth/Crypto Dev` tokens with `Crypto Dev/Eth` tokens
- If `Eth` has been selected by the user from the UI, it means that the user has `Eth` and he wants to swap it for a certain amount of `Crypto Dev` tokens
- In this case we call the `ethToCryptoDevToken` function. Note that `Eth` is sent as a value in the function because the user is paying this `Eth` to the contract. `Eth` sent is not an input param value in this case
- On the other hand, if `Eth` is not selected this means that the user wants to swap `Crypto Dev` tokens for `Eth`
- Here we call the `cryptoDevTokenToEth`
- `getAmountOfTokensReceivedFromSwap` is a function which calculates, given a certain amount of `Eth/Crypto Dev` tokens, how many `Eth/Crypto Dev` tokens would be sent back to the user

If `Eth` is selected it calls the `getAmountOfTokens` from the contract which takes in an `input` reserve and an `output` reserve. Here, input reserve would be the `Eth` balance of the contract and output reserve would be the `Crypto Dev` token reserve. Opposite would be true, if `Eth` is not selected

```
import { Contract } from "ethers";
import {
  EXCHANGE_CONTRACT_ABI,
  EXCHANGE_CONTRACT_ADDRESS,
  TOKEN_CONTRACT_ABI,
  TOKEN_CONTRACT_ADDRESS,
} from "../constants";

/*
    getAmountOfTokensReceivedFromSwap:  Returns the number of Eth/Crypto Dev tokens that can be received
    when the user swaps `_swapAmountWei` amount of Eth/Crypto Dev tokens.
*/
export const getAmountOfTokensReceivedFromSwap = async (
  _swapAmountWei,
  provider,
  ethSelected,
  ethBalance,
  reservedCD
) => {
  // Create a new instance of the exchange contract
  const exchangeContract = new Contract(
    EXCHANGE_CONTRACT_ADDRESS,
    EXCHANGE_CONTRACT_ABI,
    provider
  );
  let amountOfTokens;
  // If `Eth` is selected this means our input value is `Eth` which means our input amount would be
  // `_swapAmountWei`, the input reserve would be the `ethBalance` of the contract and output reserve
  // would be the `Crypto Dev` token reserve
  if (ethSelected) {
    amountOfTokens = await exchangeContract.getAmountOfTokens(
      _swapAmountWei,
      ethBalance,
      reservedCD
    );
  } else {
    // If `Eth` is not selected this means our input value is `Crypto Dev` tokens which means our input amount woul
    // `_swapAmountWei`, the input reserve would be the `Crypto Dev` token reserve of the contract and output reser
    // would be the `ethBalance`
    amountOfTokens = await exchangeContract.getAmountOfTokens(
      _swapAmountWei,
      reservedCD,
      ethBalance
    );
  }

  return amountOfTokens;
};

/*
  swapTokens: Swaps `swapAmountWei` of Eth/Crypto Dev tokens with `tokenToBeReceivedAfterSwap` amount of Eth/Crypto
*/
export const swapTokens = async (
  signer,
  swapAmountWei,
  tokenToBeReceivedAfterSwap,
  ethSelected
) => {
  // Create a new instance of the exchange contract
  const exchangeContract = new Contract(
    EXCHANGE_CONTRACT_ADDRESS,
    EXCHANGE_CONTRACT_ABI,
    signer
  );
  const tokenContract = new Contract(
    TOKEN_CONTRACT_ADDRESS,
    TOKEN_CONTRACT_ABI,
    signer
  );
  let tx;
  // If Eth is selected call the `ethToCryptoDevToken` function else
  // call the `cryptoDevTokenToEth` function from the contract
  // As you can see you need to pass the `swapAmount` as a value to the function because
  // it is the ether we are paying to the contract, instead of a value we are passing to the function
  if (ethSelected) {
    tx = await exchangeContract.ethToCryptoDevToken(
      tokenToBeReceivedAfterSwap,
      {
```

```
        value: swapAmountWei,
      }
    );
  } else {
    // User has to approve `swapAmountWei` for the contract because `Crypto Dev` token
    // is an ERC20
    tx = await tokenContract.approve(
      EXCHANGE_CONTRACT_ADDRESS,
      swapAmountWei.toString()
    );
    await tx.wait();
    // call cryptoDevTokenToEth function which would take in `swapAmountWei` of `Crypto Dev` tokens and would
    // send back `tokenToBeReceivedAfterSwap` amount of `Eth` to the user
    tx = await exchangeContract.cryptoDevTokenToEth(
      swapAmountWei,
      tokenToBeReceivedAfterSwap
    );
  }
  await tx.wait();
};
```

Now its time for the final stages of our app, lets add some code to the `pages/index.js` file which next already gives you. Replace all the contents of the file with the following content

```
import { BigNumber, providers, utils } from "ethers";
import Head from "next/head";
import React, { useEffect, useRef, useState } from "react";
import Web3Modal from "web3modal";
import styles from "../styles/Home.module.css";
import { addLiquidity, calculateCD } from "../utils/addLiquidity";
import {
  getCDTokensBalance,
  getEtherBalance,
  getLPTokensBalance,
  getReserveOfCDTokens,
} from "../utils/getAmounts";
import {
  getTokensAfterRemove,
  removeLiquidity,
} from "../utils/removeLiquidity";
import { swapTokens, getAmountOfTokensReceivedFromSwap } from "../utils/swap";

export default function Home() {
  /** General state variables */
  // loading is set to true when the transaction is mining and set to false when
  // the transaction has mined
  const [loading, setLoading] = useState(false);
  // We have two tabs in this dapp, Liquidity Tab and Swap Tab. This variable
  // keeps track of which Tab the user is on. If it is set to true this means
  // that the user is on `liquidity` tab else he is on `swap` tab
  const [liquidityTab, setLiquidityTab] = useState(true);
  // This variable is the `0` number in form of a BigNumber
  const zero = BigNumber.from(0);
  /** Variables to keep track of amount */
  // `ethBalance` keeps track of the amount of Eth held by the user's account
  const [ethBalance, setEtherBalance] = useState(zero);
  // `reservedCD` keeps track of the Crypto Dev tokens Reserve balance in the Exchange contract
  const [reservedCD, setReservedCD] = useState(zero);
  // Keeps track of the ether balance in the contract
  const [etherBalanceContract, setEtherBalanceContract] = useState(zero);
  // cdBalance is the amount of `CD` tokens help by the users account
  const [cdBalance, setCDBalance] = useState(zero);
  // `lpBalance` is the amount of LP tokens held by the users account
  const [lpBalance, setLPBalance] = useState(zero);
  /** Variables to keep track of liquidity to be added or removed */
  // addEther is the amount of Ether that the user wants to add to the liquidity
  const [addEther, setAddEther] = useState(zero);
  // addCDTokens keeps track of the amount of CD tokens that the user wants to add to the liquidity
  // in case when there is no initial liquidity and after liquidity gets added it keeps track of the
  // CD tokens that the user can add given a certain amount of ether
  const [addCDTokens, setAddCDTokens] = useState(zero);
  // removeEther is the amount of `Ether` that would be sent back to the user based on a certain number of `LP` token
  const [removeEther, setRemoveEther] = useState(zero);
  // removeCD is the amount of `Crypto Dev` tokens that would be sent back to the user based on a certain number of `
  // that he wants to withdraw
  const [removeCD, setRemoveCD] = useState(zero);
  // amount of LP tokens that the user wants to remove from liquidity
  const [removeLPTokens, setRemoveLPTokens] = useState("0");
  /** Variables to keep track of swap functionality */
  // Amount that the user wants to swap
  const [swapAmount, setSwapAmount] = useState("");
```

```
// This keeps track of the number of tokens that the user would receive after a swap completes
const [tokenToBeReceivedAfterSwap, settokenToBeReceivedAfterSwap] =
  useState(zero);
// Keeps track of whether  `Eth` or `Crypto Dev` token is selected. If `Eth` is selected it means that the user
// wants to swap some `Eth` for some `Crypto Dev` tokens and vice versa if `Eth` is not selected
const [ethSelected, setEthSelected] = useState(true);
/** Wallet connection */
// Create a reference to the Web3 Modal (used for connecting to Metamask) which persists as long as the page is open
const web3ModalRef = useRef();
// walletConnected keep track of whether the user's wallet is connected or not
const [walletConnected, setWalletConnected] = useState(false);

/**
 * getAmounts call various functions to retrive amounts for ethbalance,
 * LP tokens etc
 */
const getAmounts = async () => {
  try {
    const provider = await getProviderOrSigner(false);
    const signer = await getProviderOrSigner(true);
    const address = await signer.getAddress();
    // get the amount of eth in the user's account
    const _ethBalance = await getEtherBalance(provider, address);
    // get the amount of `Crypto Dev` tokens held by the user
    const _cdBalance = await getCDTokensBalance(provider, address);
    // get the amount of `Crypto Dev` LP tokens held by the user
    const _lpBalance = await getLPTokensBalance(provider, address);
    // gets the amount of `CD` tokens that are present in the reserve of the `Exchange contract`
    const _reservedCD = await getReserveOfCDTokens(provider);
    // Get the ether reserves in the contract
    const _ethBalanceContract = await getEtherBalance(provider, null, true);
    setEtherBalance(_ethBalance);
    setCDBalance(_cdBalance);
    setLPBalance(_lpBalance);
    setReservedCD(_reservedCD);
    setReservedCD(_reservedCD);
    setEtherBalanceContract(_ethBalanceContract);
  } catch (err) {
    console.error(err);
  }
};

/**** SWAP FUNCTIONS ****/

/**
 * swapTokens: Swaps  `swapAmountWei` of Eth/Crypto Dev tokens with `tokenToBeReceivedAfterSwap` amount of Eth/Crypto
 */
const _swapTokens = async () => {
  try {
    // Convert the amount entered by the user to a BigNumber using the `parseEther` library from `ethers.js`
    const swapAmountWei = utils.parseEther(swapAmount);
    // Check if the user entered zero
    // We are here using the `eq` method from BigNumber class in `ethers.js`
    if (!swapAmountWei.eq(zero)) {
      const signer = await getProviderOrSigner(true);
      setLoading(true);
      // Call the swapTokens function from the `utils` folder
      await swapTokens(
        signer,
        swapAmountWei,
        tokenToBeReceivedAfterSwap,
        ethSelected
      );
      setLoading(false);
      // Get all the updated amounts after the swap
      await getAmounts();
      setSwapAmount("");
    }
  } catch (err) {
    console.error(err);
    setLoading(false);
    setSwapAmount("");
  }
};

/**
 * _getAmountOfTokensReceivedFromSwap:  Returns the number of Eth/Crypto Dev tokens that can be received
 * when the user swaps `_swapAmountWEI` amount of Eth/Crypto Dev tokens.
 */
const _getAmountOfTokensReceivedFromSwap = async (_swapAmount) => {
  try {
    // Convert the amount entered by the user to a BigNumber using the `parseEther` library from `ethers.js`
    const _swapAmountWEI = utils.parseEther(_swapAmount.toString());
    // Check if the user entered zero
    // We are here using the `eq` method from BigNumber class in `ethers.js`
```

```
    if (!_swapAmountWEI.eq(zero)) {
      const provider = await getProviderOrSigner();
      // Get the amount of ether in the contract
      const _ethBalance = await getEtherBalance(provider, null, true);
      // Call the `getAmountOfTokensReceivedFromSwap` from the utils folder
      const amountOfTokens = await getAmountOfTokensReceivedFromSwap(
        _swapAmountWEI,
        provider,
        ethSelected,
        _ethBalance,
        reservedCD
      );
      settokenToBeReceivedAfterSwap(amountOfTokens);
    } else {
      settokenToBeReceivedAfterSwap(zero);
    }
  } catch (err) {
    console.error(err);
  }
};

/*** END ***/

/**** ADD LIQUIDITY FUNCTIONS ****/

/**
 * _addLiquidity helps add liquidity to the exchange,
 * If the user is adding initial liquidity, user decides the ether and CD tokens he wants to add
 * to the exchange. If he is adding the liquidity after the initial liquidity has already been added
 * then we calculate the crypto dev tokens he can add, given the Eth he wants to add by keeping the ratios
 * constant
 */
const _addLiquidity = async () => {
  try {
    // Convert the ether amount entered by the user to Bignumber
    const addEtherWei = utils.parseEther(addEther.toString());
    // Check if the values are zero
    if (!addCDTokens.eq(zero) && !addEtherWei.eq(zero)) {
      const signer = await getProviderOrSigner(true);
      setLoading(true);
      // call the addLiquidity function from the utils folder
      await addLiquidity(signer, addCDTokens, addEtherWei);
      setLoading(false);
      // Reinitialize the CD tokens
      setAddCDTokens(zero);
      // Get amounts for all values after the liquidity has been added
      await getAmounts();
    } else {
      setAddCDTokens(zero);
    }
  } catch (err) {
    console.error(err);
    setLoading(false);
    setAddCDTokens(zero);
  }
};

/**** END ****/

/**** REMOVE LIQUIDITY FUNCTIONS ****/

/**
 * _removeLiquidity: Removes the `removeLPTokensWei` amount of LP tokens from
 * liquidity and also the calculated amount of `ether` and `CD` tokens
 */
const _removeLiquidity = async () => {
  try {
    const signer = await getProviderOrSigner(true);
    // Convert the LP tokens entered by the user to a BigNumber
    const removeLPTokensWei = utils.parseEther(removeLPTokens);
    setLoading(true);
    // Call the removeLiquidity function from the `utils` folder
    await removeLiquidity(signer, removeLPTokensWei);
    setLoading(false);
    await getAmounts();
    setRemoveCD(zero);
    setRemoveEther(zero);
  } catch (err) {
    console.error(err);
    setLoading(false);
    setRemoveCD(zero);
    setRemoveEther(zero);
  }
};

/**
```

```
 * _getTokensAfterRemove: Calculates the amount of `Ether` and `CD` tokens
 * that would be returned back to user after he removes `removeLPTokenWei` amount
 * of LP tokens from the contract
 */
const _getTokensAfterRemove = async (_removeLPTokens) => {
  try {
    const provider = await getProviderOrSigner();
    // Convert the LP tokens entered by the user to a BigNumber
    const removeLPTokenWei = utils.parseEther(_removeLPTokens);
    // Get the Eth reserves within the exchange contract
    const _ethBalance = await getEtherBalance(provider, null, true);
    // get the crypto dev token reserves from the contract
    const cryptoDevTokenReserve = await getReserveOfCDTokens(provider);
    // call the getTokensAfterRemove from the utils folder
    const { _removeEther, _removeCD } = await getTokensAfterRemove(
      provider,
      removeLPTokenWei,
      _ethBalance,
      cryptoDevTokenReserve
    );
    setRemoveEther(_removeEther);
    setRemoveCD(_removeCD);
  } catch (err) {
    console.error(err);
  }
};

/**** END ****/

/**
 * 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);
  }
};

/**
 * 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, etc.
 *
 * A `Signer` is a special type of Provider used in case a `write` transaction
 * needs to be made to the blockchain, which involves the connected account
 * needing to make a digital signature to authorize the transaction being
 * sent. Metamask exposes a Signer API to allow your website to 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();
    getAmounts();
  }
}, [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 (liquidityTab) {
    return (
      <div>
        <div className={styles.description}>
          You have:
          <br />
          {/* Convert the BigNumber to string using the formatEther function from ethers.js */}
          {utils.formatEther(cdBalance)} Crypto Dev Tokens
          <br />
          {utils.formatEther(ethBalance)} Ether
          <br />
          {utils.formatEther(lpBalance)} Crypto Dev LP tokens
        </div>
        <div>
          {/* If reserved CD is zero, render the state for liquidity zero where we ask the user
          how much initial liquidity he wants to add else just render the state where liquidity is not zero and
          we calculate based on the `Eth` amount specified by the user how much `CD` tokens can be added */}
          {utils.parseEther(reservedCD.toString()).eq(zero) ? (
            <div>
              <input
                type="number"
                placeholder="Amount of Ether"
                onChange={(e) => setAddEther(e.target.value || "0")}
                className={styles.input}
              />
              <input
                type="number"
                placeholder="Amount of CryptoDev tokens"
                onChange={(e) =>
                  setAddCDTokens(
                    BigNumber.from(utils.parseEther(e.target.value || "0"))
                  )
                }
                className={styles.input}
              />
              <button className={styles.button1} onClick={_addLiquidity}>
                Add
              </button>
            </div>
          ) : (
            <div>
              <input
                type="number"
                placeholder="Amount of Ether"
                onChange={async (e) => {
                  setAddEther(e.target.value || "0");
                  // calculate the number of CD tokens that
                  // can be added given  `e.target.value` amount of Eth
                  const _addCDTokens = await calculateCD(
                    e.target.value || "0",
                    etherBalanceContract,
                    reservedCD
                  );
                  setAddCDTokens(_addCDTokens);
                }}
                className={styles.input}
```

```
            />
            <div className={styles.inputDiv}>
              {/* Convert the BigNumber to string using the formatEther function from ethers.js */}
              {`You will need ${utils.formatEther(addCDTokens)} Crypto Dev
              Tokens`}
            </div>
            <button className={styles.button1} onClick={_addLiquidity}>
              Add
            </button>
          </div>
        )}
        <div>
          <input
            type="number"
            placeholder="Amount of LP Tokens"
            onChange={async (e) => {
              setRemoveLPTokens(e.target.value || "0");
              // Calculate the amount of Ether and CD tokens that the user would receive
              // After he removes `e.target.value` amount of `LP` tokens
              await _getTokensAfterRemove(e.target.value || "0");
            }}
            className={styles.input}
          />
          <div className={styles.inputDiv}>
            {/* Convert the BigNumber to string using the formatEther function from ethers.js */}
            {`You will get ${utils.formatEther(removeCD)} Crypto
          Dev Tokens and ${utils.formatEther(removeEther)} Eth`}
          </div>
          <button className={styles.button1} onClick={_removeLiquidity}>
            Remove
          </button>
        </div>
      </div>
    </div>
  );
} else {
  return (
    <div>
      <input
        type="number"
        placeholder="Amount"
        onChange={async (e) => {
          setSwapAmount(e.target.value || "");
          // Calculate the amount of tokens user would receive after the swap
          await _getAmountOfTokensReceivedFromSwap(e.target.value || "0");
        }}
        className={styles.input}
        value={swapAmount}
      />
      <select
        className={styles.select}
        name="dropdown"
        id="dropdown"
        onChange={async () => {
          setEthSelected(!ethSelected);
          // Initialize the values back to zero
          await _getAmountOfTokensReceivedFromSwap(0);
          setSwapAmount("");
        }}
      >
        <option value="eth">Ethereum</option>
        <option value="cryptoDevToken">Crypto Dev Token</option>
      </select>
      <br />
      <div className={styles.inputDiv}>
        {/* Convert the BigNumber to string using the formatEther function from ethers.js */}
        {ethSelected
          ? `You will get ${utils.formatEther(
              tokenToBeReceivedAfterSwap
            )} Crypto Dev Tokens`
          : `You will get ${utils.formatEther(
              tokenToBeReceivedAfterSwap
            )} Eth`}
      </div>
      <button className={styles.button1} onClick={_swapTokens}>
        Swap
      </button>
    </div>
  );
}
};

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 Exchange!</h1>
          <div className={styles.description}>
            Exchange Ethereum &#60;&#62; Crypto Dev Tokens
          </div>
          <div>
            <button
              className={styles.button}
              onClick={() => {
                setLiquidityTab(true);
              }}
            >
              Liquidity
            </button>
            <button
              className={styles.button}
              onClick={() => {
                setLiquidityTab(false);
              }}
            >
              Swap
            </button>
          </div>
          {renderButton()}
        </div>
        <div>
          <img className={styles.image} src="./cryptodev.svg" />
        </div>
      </div>

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

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

```
npm run dev
```

Your Exchange dapp should now work without errors 🚀

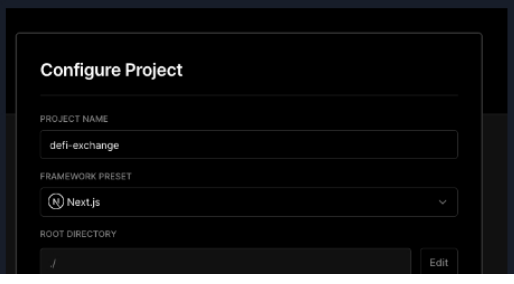
Push your Code to Github

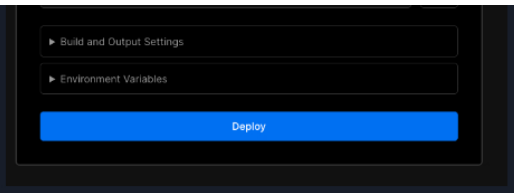
Make sure you push all your code to github before moving to the next step of deployment

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 Defi-Exchange 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`
- 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

Submit Practical

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

Ethereum Rinkeby | 0x... | Submit