

CS5576 - Blockchain Technologies

Author: Varshith Thota

Build a DApp to send digital tokens

A DApp is basically just an application that doesn't have a backend system controlled by a single point of contact. That means, the outlook of the application is just like any other application, but unlike the tradition; the backend works on blockchain.

There are 2 main components while building a DApp:

- a. User Interface: could be as simple as using HTML/CSS and JavaScript
- b. Smart contracts: smart contracts are the backend logic of a DApp. They keep track of the tokens, balances, transactions etc.,.

To access or use any DApp, we don't use email IDs and passwords for authentication. Instead, we use decentralized wallets that use keys for authentication. Some of the very popularly used decentralized wallets are Metamask, Brave Wallet, Coinbase wallet. In this experiment, we will use Metamask.

Part 0: Setup

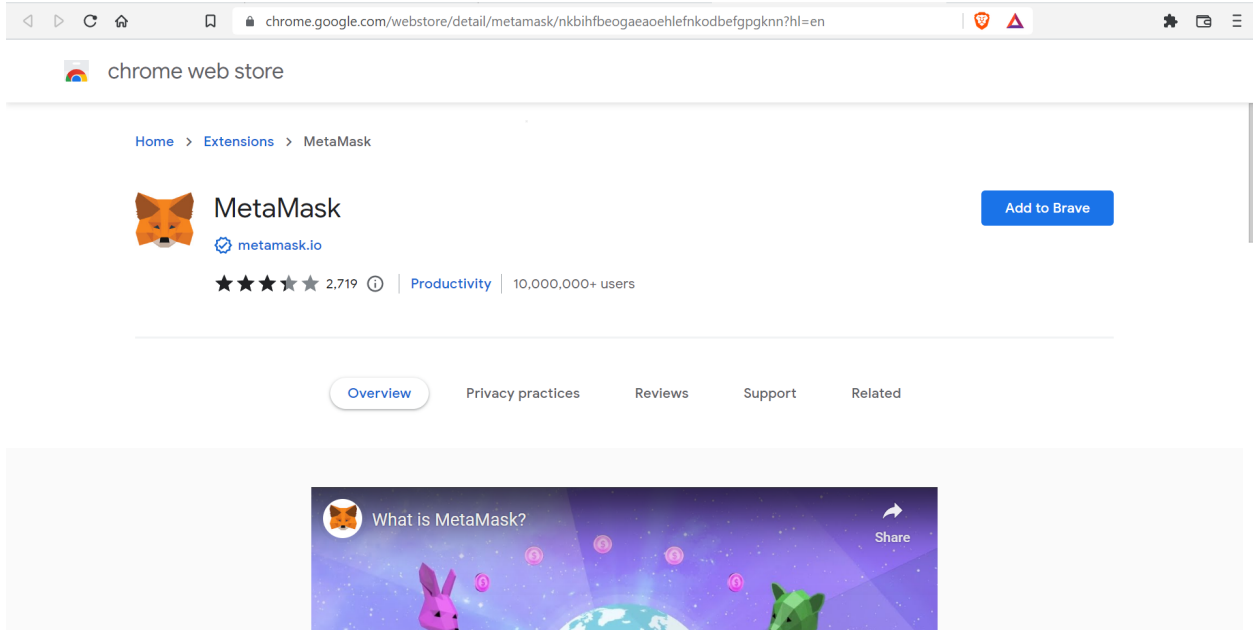
1. Node.js installation: <https://nodejs.org/>
2. Metamask wallet installation: [MetaMask](#)
3. A nice text editor like [Visual Studio Code](#)

Part 1: Developing the user interface

We are building a DApp that runs on a smart contract that lets us transfer digital tokens from our wallet to a recipient's wallet.

1. Build a user interface.

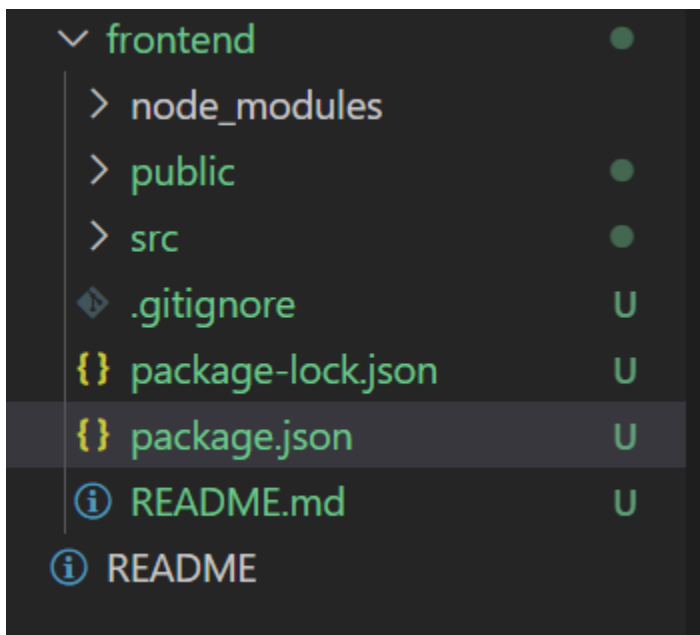
In a DApp, we don't use any email IDs or passwords for authentication, instead we have our own decentralized wallets that work as our authentication tool. Create your wallet on Metamask.



2. In the root, create a new directory with the name: “frontend”. Move into the directory and start creating a new React application by typing this in the terminal:

`npx create-react-app .`

You will see the file structure like this:



3. After the user is authenticated, since we are building the application for transferring tokens, build the UI component with two inputs: Recipient’s wallet address and amount of tokens being sent.

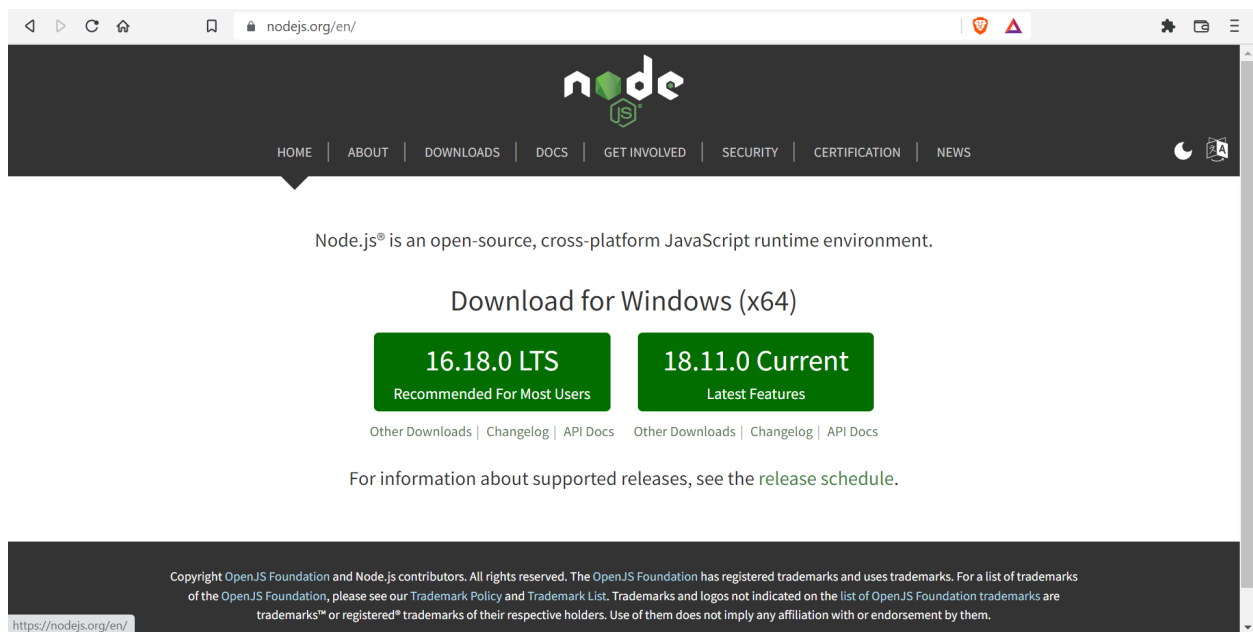
4. Then a button which says “Send”, it acts as a submit button and fires up the Metamask again for transaction confirmation.

Part 2: Building a smart contract

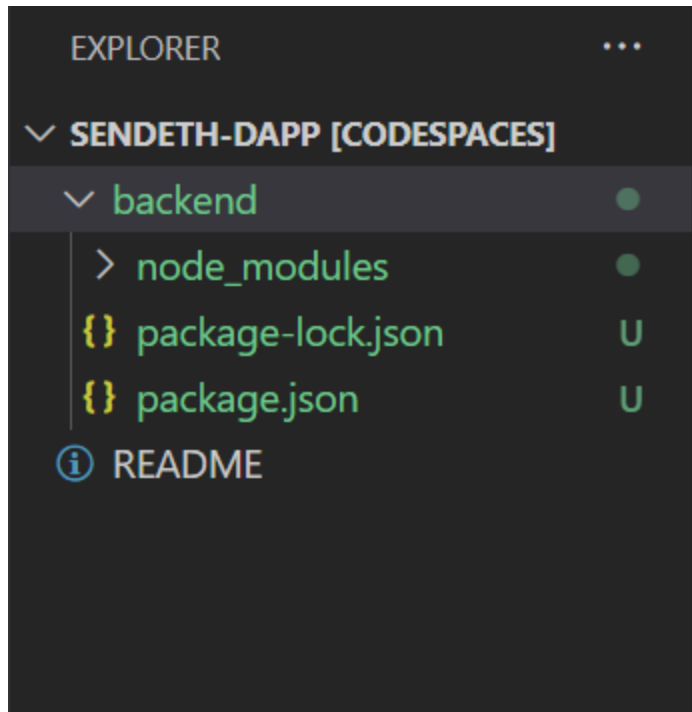
This is the backbone of our DApp, we are building a smart contract that transfers tokens. Fortunately, it is very easy to write smart contracts and deploy them. Smart contracts are generally written in Solidity programming language which is developed by Ethereum foundation. It shares a similar syntax with that of JavaScript. There is no need to start writing everything from scratch because a lot of open-source libraries are already available which provide us with a structure to get started.

We are using the “Hardhat” library, it is an Ethereum development environment.

1. Make sure to have Node.js installed on your computer.



2. In the root directory create a new folder with the name “**backend**”; move into the folder and run **npm install --save-dev hardhat @nomiclabs/hardhat-waffle ethereum-waffle chai @nomiclabs/hardhat-ethers ethers** to install the hardhat library in your project. It downloads the hardhat package into your project.



You will see the file structure similar to the one in the picture.

3. After the installation is done, run **npx hardhat** for executing the library pack.

You should see the content in the picture on your terminal. Choose “create a JavaScript project” and select the default options at each step.

```

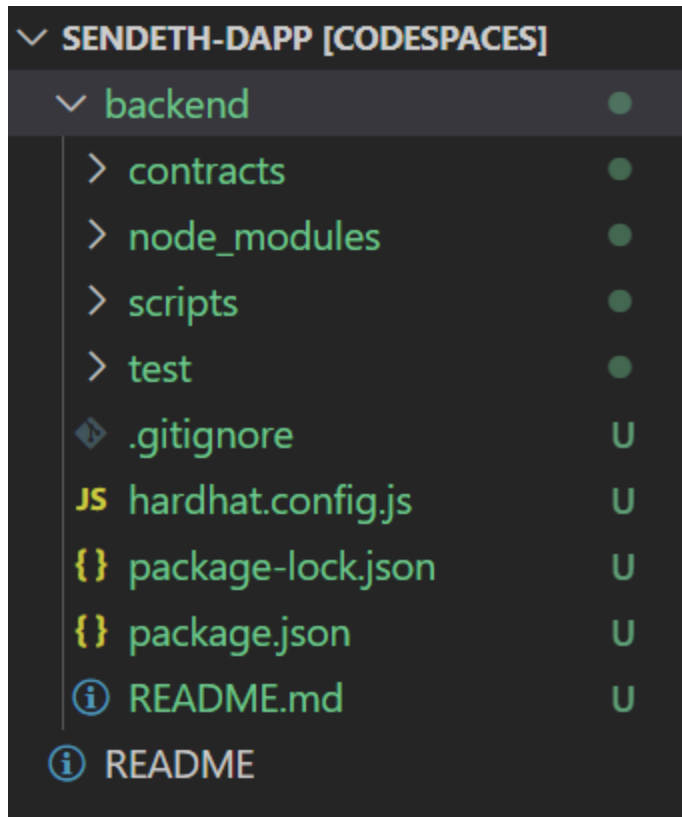
@v4rsh1th → /workspaces/sendeth-dapp/backend (master X) $ npx hardhat
888      888      888 888      888
888      888      888 888      888
888      888      888 888      888
88888888888 8888b. 888d888 .d88888 88888b. 8888b. 888888
888      888      "88b 888P" d88" 888 888 "88b  "88b 888
888      888 .d888888 888 888 888 888 .d888888 888
888      888 888 888 888 Y88b 888 888 888 888 Y88b.
888      888 "Y888888 888 "Y88888 888 888 "Y888888 "Y888

Welcome to Hardhat v2.12.0

? What do you want to do? ...
  > Create a JavaScript project
    Create a TypeScript project
    Create an empty hardhat.config.js
    Quit

```

Now, the file structure will look like this:



4. In the “contracts” folder, we write all of our smart contracts. You will see a file with the name “Lock.sol” (or with any other file name). Delete the file, we don’t need it.
5. Create a new file: “Transactions.sol” (or any name!)
6. Write a smart contract that transfers tokens in Solidity.
 - a. Choose the version of Solidity that we are going to use: **pragma solidity ^0.8.0;**
It gives a warning message that we need a license. Type:
// SPDX-License-Identifier: MIT on the first line.
 - b. Create a contract: “Transactions”.
Syntax: **contract Transactions { }**
 - c. Create a function: “addToFunction()”
Syntax: **function addToBlockchain() public { }**
7. Now, the smart contract needs to be deployed to the Ethereum network. We are going to deploy the contract on an Ethereum Testnet called “Goerli”. It’s easy to deploy the contract since we are using the Hardhat development package.
8. To deploy the smart contract, in the “scripts” folder, create a new file with the name “deploy.js”. This file will contain just the simple JavaScript code to deploy it to a network.

```
const main = async () => {  
  const Transactions = await  
  hre.ethers.getContractFactory("Transactions");
```

```

    const transactions = await Transactions.deploy();
    await transactions.deployed();
    console.log("Transactions deployed to:", transactions.address);
  };

  const runMain = async () => {
    try {
      await main();
      process.exit(0);
    } catch (error) {
      console.error(error);
      process.exit(1);
    }
  };

  runMain();

```

9. In the “hardhat.config.js” file, we need to write down some parameters to successfully deploy the contract. It takes the private key of the wallet you are using while deploying the smart contract and the RPC of the Ethereum network i.e., Goerli

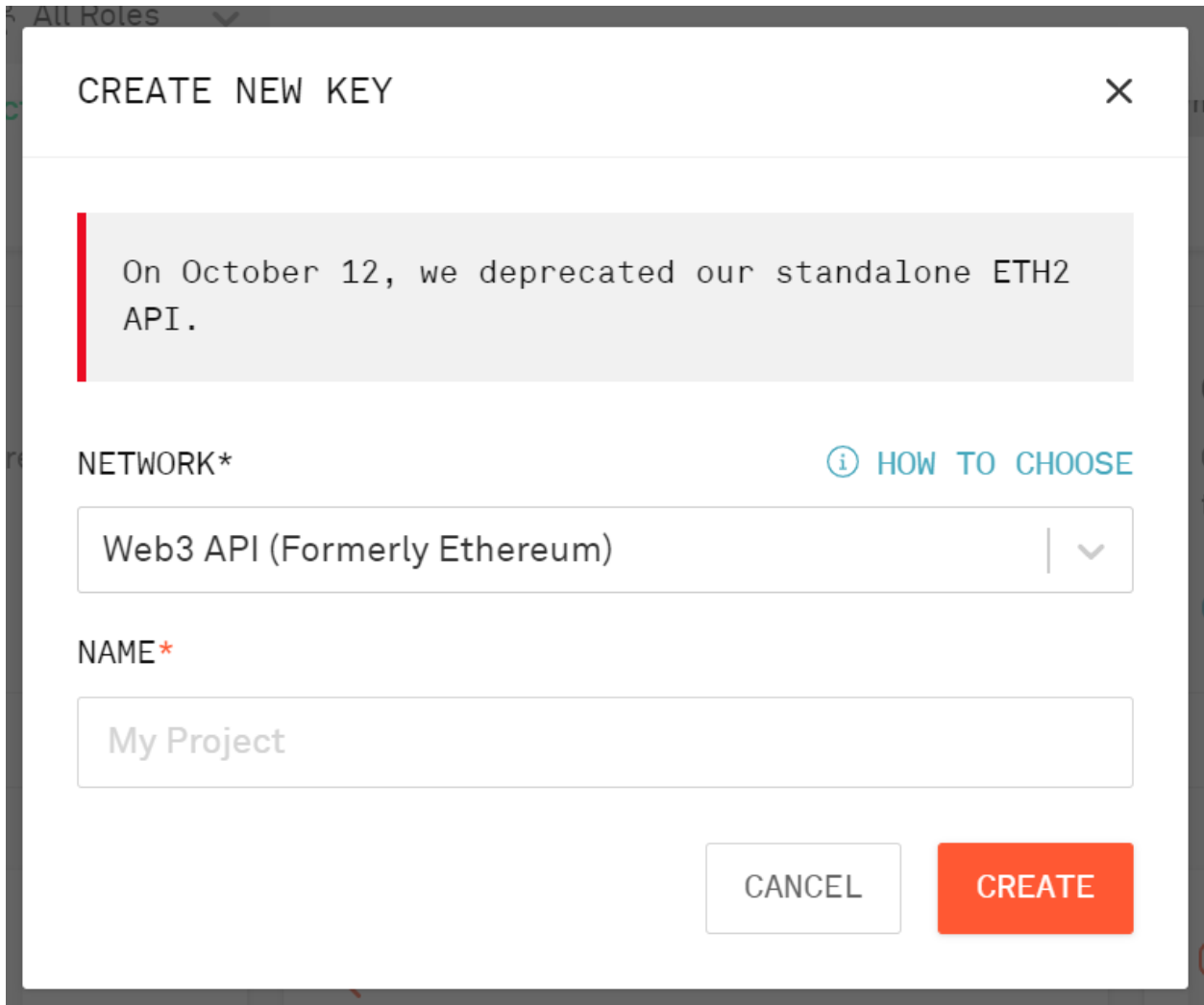
10. There are platforms which provide us the API endpoints/RPC for the Ethereum networks. One of the most used platforms is “Infura”. Create an account on Infura. Then, you will be redirected to your Dashboard. You will see an option: “create a Web3 project”

CREATE A WEB3 PROJECT

More scalable, more secure, and more sustainable.

[GET STARTED →](#)

Then, you will see a prompt:



CREATE NEW KEY

On October 12, we deprecated our standalone ETH2 API.

NETWORK* [HOW TO CHOOSE](#)

Web3 API (Formerly Ethereum)

NAME*

My Project

CANCEL CREATE

Select “Web3 API (Formerly Ethereum)” option and type your project’s name. Hit CREATE.

11. Then, you will see “**Network Endpoints**”. In Ethereum, select Goerli and copy the URL.

2350. More on setting up, writing and deploying the smart contracts can be found in the official documentation: <https://hardhat.org/hardhat-runner/docs/getting-started>

12. In the hardhat.config.js file:

```
require("@nomiclabs/hardhat-waffle");
```

```
module.exports = {  
  solidity: '0.8.0',  
  networks: {  
    goerli: {  
      url: "<TYPE_INFURA_GORLI_URL_HERE",  
      accounts: ["<TYPE_YOUR_WALLET_PRIVATE_KEY_HERE"]  
    }  
  }  
}
```

```

    }
  }
}

```

13. Save the file. In the terminal, type:

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

You will see the following on your terminal. It generates a contract address.

```

@v4rsh1th → /workspaces/sendeth-dapp/backend (master X) $ npx hardhat run scripts/deploy.js --network goerli
Downloading compiler 0.8.0
Compiled 1 Solidity file successfully
Transactions deployed to: 0x1234567890123456789012345678901234567890
@v4rsh1th → /workspaces/sendeth-dapp/backend (master X) $ 

```

14. After the deployment, we see two new directories: “artifacts” and “cache”. In the artifacts folder, we see a JSON file with the name “Transactions.json”. It is called the ABI (Application Binary Interface). In general, an ABI is the interface between two program modules, one of which is often at the level of machine code. The interface is the de facto method for encoding/decoding data into/out of the machine code. In Ethereum, it’s basically how you can encode Solidity contract calls for the EVM and, backwards, how to read the data out of transactions.

15. Now move into the “frontend” folder. Run: **npm install ethers**

16. In the **App.js** file, clear everything and start it fresh.

```

JS App.js  U X
frontend > src > JS App.js > App
1  import React, { useEffect, useState } from 'react';
2  import { ethers } from "ethers";
3  import './App.css';
4
5  function App() {
6    return (
7      <div className="App">
8
9      </div>
10   );
11 }
12
13 export default App;
14

```

This is how it should look like to get started. Import “**React**”, “**ethers**” library.

17. Copy the “**Transactions.json**” file from the **artifacts** folder and paste it in a new folder called “**utils**” inside the “**frontend**” folder.

18. In the “**utils**” folder, create a new file called “**constants.js**” and put this code:

```
import abi from "../Transactions.json";
export const contractABI = abi.abi;
export const contractAddress = "<YOUR_CONTRACT_ADDRESS>";
```

19. Move to the “**src**” folder, and create a new folder named “**context**”. And create a new file named “**TransactionContext.jsx**” and import:

```
import React, { useEffect, useState } from "react";
import { ethers } from "ethers";
import { contractABI, contractAddress } from
"../utils/constants";
```

20. Create an Ethereum object, to interact with the Ethereum wallets:

```
if (typeof window !== "undefined") {
  window.ethereum;
}
```

21. Create a new function named: “**getEthereumContract()**” this function initiates the smart contract in our application:

```
const getEthereumContract = () => {
  const provider = new ethers.providers.Web3Provider(ethereum);
  const signer = provider.getSigner();
  const transactionContract = new ethers.Contract(
    contractAddress,
    contractABI,
    signer
  );
  return transactionContract;
};
```

22. Now, go to **App.js** and import:

```
import React, {
  useContext,
  useState,
  useEffect
```

```

} from 'react';
import { ethers } from "ethers";
import { TransactionContext } from
'../context/TransactionContext';

```

Inside the App() function, type this code:

The function “handleSubmit()” is the function that is going to fire up when the user clicks the “Send” button.

```

const {
  connectWallet,
  currentAccount,
  formData,
  sendTransaction,
  handle_change
} = useContext(TransactionContext);

const handleSubmit = (e) => {
  const { addressTo, amount } = formData;
  e.preventDefault();
  if (!addressTo || !amount) return;
  sendTransaction();
};

```

23. Inside the **TransactionContext.jsx** file, we need to export all the data to the main function i.e. App() in the **App.js** file. Create a new function:

```

export const TransactionProvider = ({ children }) => {
  return (
    <TransactionContext.Provider value={{ }}>
      {children}
    </TransactionContext.Provider>
  );
};

```

Now, all this data should also be exported to the root file i.e. **index.js**

```
JS App.js M TransactionContext.jsx U JS index.js M X JS constants.js
frontend > src > JS index.js > ...
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import './index.css';
4 import App from './App';
5 import reportWebVitals from './reportWebVitals';
6
7 import TransactionProvider from "../context/TransactionContext";
8
9 const root = ReactDOM.createRoot(document.getElementById('root'));
10 root.render(
11   <TransactionProvider>
12     <React.StrictMode>
13       <App />
14     </React.StrictMode>
15   </TransactionProvider>
16 );
17
18 // If you want to start measuring performance in your app, pass a function
19 // to log results (for example: reportWebVitals(console.log))
20 // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
21 reportWebVitals();
22
```

24. Inside the **TransactionProvider()** function, create a function to connect to a wallet:

```
const [currentAccount, setCurrentAccount] = useState();
const connectWallet = async () => {
  if (typeof window.ethereum !== 'undefined') {
    const accounts = await ethereum.request({ method:
'eth_requestAccounts' });
    setCurrentAccount(accounts[0]);
  } else {
    window.open("https://metamask.io");
  }
};
```

25. Go to **App.js** and type this inside App.js() function to get the wallet details updated on the page.

```
const { connectWallet, currentAccount } =
useContext(TransactionContext);
```

26. Now, inside the **TransactionProvider()**, create a new function called **sendTransaction()**

```
const [formData, setFormData] = useState({addressTo: "",
  amount: "",
});
const handle_change = (e, name) => {
  setFormData(prevState => ({ ...prevState, [name]:
e.target.value }));
};

const sendTransaction = async (connectedAccount =
currentAccount) => {
  try {
    if (!ethereum) return alert("Please install Metamask.");
    const { addressTo, amount } = formData;
    const transactionContract = getEthereumContract();
    const parsedAmount = ethers.utils.parseEther(amount);
  } catch (error) {
    console.error(error);
  }
};
```

27. This is the full code for **TransactionContext.jsx**:

```
import React, { useEffect, useState } from "react";
import { ethers } from "ethers";
import { contractABI, contractAddress } from "../utils/constants";

export const TransactionContext = React.createContext();

if (typeof window !== "undefined") {
  window.ethereum;
}

const getEthereumContract = () => {
  const provider = new ethers.providers.Web3Provider(ethereum);
  const signer = provider.getSigner();
  const transactionContract = new ethers.Contract(
    contractAddress,
```

```

        contractABI,
        signer
    );
    return transactionContract;
};

export const TransactionProvider = ({ children }) => {
    const [currentAccount, setCurrentAccount] = useState();
    const [isLoading, setIsLoading] = useState(false);
    const [formData, setFormData] = useState({
        addressTo: "",
        amount: "",
    });

    const handle_change = (e, name) => {
        setFormData(prevState => ({ ...prevState, [name]: e.target.value
    }));
    };

    const connectWallet = async () => {
        if (typeof window.ethereum !== 'undefined') {
            const accounts = await ethereum.request({ method:
'eth_requestAccounts' });
            setCurrentAccount(accounts[0]);
        } else {
            window.open("https://metamask.io");
        }
    };

    const sendTransaction = async (connectedAccount = currentAccount) =>
    {
        try {
            if (!ethereum) return alert("Please install Metamask.");

            const { addressTo, amount } = formData;
            const transactionContract = getEthereumContract();
            const parsedAmount = ethers.utils.parseEther(amount);
        } catch (error) {

```

```

        console.error(error);
    }
};

return (
    <TransactionContext.Provider value={{
        connectWallet,
        currentAccount,
        formData,
        setFormData,
        handle_change,
        sendTransaction
    }}>
        {children}
    </TransactionContext.Provider>
);
};

```

28. Now, move to **App.js** and build the user interface, i.e. the “connect to wallet” button and the form field to send the tokens to a recipient.

29. This button fires up the **connectWallet()** function from the **TransactionContext.jsx** and fires up the Metamask wallet.

```

<button id={styles.bigger_btn}
        type="button"
        onClick={connectWallet}
    >

```

30. Now after logging in, the user shall be seeing a form which has two inputs: recipient’s address and the amount of tokens being sent.

31. Create a form:

This input field is for inputting the recipient’s wallet address:

```

<input name="addressTo" type="text" placeholder="0x address"
    onChange={e => handle_change(e, 'addressTo')} />

```

This input field is for inputting the amount of tokens being sent:

```

<input name="amount" type="number" placeholder="0.0" onChange={e =>
    handle_change(e, 'amount')} />

```

Now create a “Send” button to submit all the data inputted in the input fields.

```
<button type="button" onClick={handleSubmit}>Send</button>
```

-X-

At this point, the DApp is ready to send the tokens to any Ethereum based wallet address. This DApp only works for sending the tokens, there are a lot of other functionalities we can add such as: liquidity, pooling, exchanging between the tokens etc.

I have personally built a DApp that sends that tokens from one wallet to other (that is the only functionality, *for now*). There are tons of functionalities that I'm intending to add later and expand on. It is deployed at: <https://apexchain.vercel.app>. Also, the code is on GitHub: <https://github.com/v4rsh1th/apex>, you can have a look at the codebase and get the idea of how it is built even before getting started.