The Ultimate Guide
React dApps

Eric Morgan

Version v0.0.001, Dec. 30, 2021

Table of Contents

Chapter 1. Introduction

Chapter 2. Setup

2.1. Install MetaMask

2.2. Install Hardhat

- Configure `artifacts` to build into the UI folder

Chapter 3. Project Structure

Chapter 4. Unit Testing

I am putting unit testing so close to the beginning of the book to stress how important they are. Once your contract is deployed, it **cannot** be changed, so you should make an effort to test every possible piece of functionality to ensure it's working properly.

Explain how to run unit tests on an actual testnet (rather than locally)

Chapter 5. Libraries

Chapter 6. Data Model

Chapter 7. Components

Chapter 8. Client API

Chapter 9. Miscellaneous

Open Zeppelin Contracts: I strongly recommend sticking to these as much as you can. They have been tested and are standardized.

Flattening Contracts

Verifying Contracts

Keep your private keys **private** at all costs. Do not share them with anyone. If you are using source control (such as Github), take care to not commit any private keys to your repository.

Testing on Local/TestNet/MainNet

IPFS

OpenSea

Glossary/Random Tidbits

ABI standard for Application Binary Interface. Each Smart Contract has an ABI which defines the interface exposed by the Smart Contract. The ABI is often needed when trying to interact with a Smart Contract from a JavaScript framework.

`EtherScan` is a website that allows you to view the transactions made by an address on the network. That address could be for a Smart Contract or for an individual wallet. Other networks will have a similar website, such as `PolygonScan` for the Polygon network.

TODO: Add more to this NOTE: ERC20 tokens have a property called `decimals`. When you are interacting with these tokens, be sure to include the correct number of "decimals".

TODO: Explain ERC20 approvals

TODO: Explain `onlyOwner`

9.1. Wallet Interaction

Users will need to connect their wallets in order to perform any transactions in your application. We will set up some basic functions to allow us to interact with a user's wallet. For this guide, we will use the MetaMask wallet. MetaMask injects an API into the `window.ethereum` property and this is how we will be able to interact with MetaMask. You can check the MetaMask documentation (https://docs.metamask.io/guide/) for more technical details.

There are libraries out there that provide additional functionality such as working with multiple wallets and can simply be imported into your application. But for simplicity and learning purposes, we will build our own.

The functionality we want to implement is:

- Connect to a wallet
- Disconnect from a wallet
- Add a network to a wallet
- Switch to a network

Create a new file called `WalletUtils.ts` with the following stub functions:

WalletUtils.ts

```
import { MetaMaskInpageProvider } from "@metamask/providers";

declare global {
    interface Window {
        ethereum: MetaMaskInpageProvider;
    }
}

export default class WalletUtils {
    // Connect user's wallet...
    public static async connectWallet(): Promise<null | string> {

    }

    // Get the currently-connected wallet...
    public static async getCurrentWallet(): Promise<null | string> {

    }

    // Switch to the specified network, or add it if it doesn't exist...
    public static async switchToNetwork(chainId: number, rpcUrl: string): Promise
<void> {

    }

    // Add the specified network to the wallet...
    public static async addNetwork(chainId: number, rpcUrl: string): Promise<void>
{

    }
}
```

Before implementing these functions, let's take a quick review of the code.

```
import { MetaMaskInpageProvider } from "@metamask/providers";

declare global {
    interface Window {
        ethereum: MetaMaskInpageProvider;
    }
}
```

MetaMask injects `window.ethereum` so this code is grabbing the types for `ethereum` and specifying that the `window` object will have a property called `ethereum`. Without this, TypeScript would give us an error and we wouldn't get any intellisense.

We are creating a class with all `static` methods simply as a way of organizing the code. When we use the functions, we will use them like: `WalletUtils.connectWallet()`. We have also made all of the functions `async` so that we can use `await` to get their results.

Now let's implement each function.

9.1.1. connectWallet

In `connectWallet`, we will return either `null` or the wallet address. If you'd like, you can return an object with a `boolean` if connecting was successful or not, but for simplicity, we will just use `null` to indicate that the connection could not be made.

WalletUtils.ts

```
// Connect user's wallet...
public static async connectWallet(): Promise<null | string> {

}
```

9.1.2. getCurrentWallet

WalletUtils.ts

```
// Connect user's wallet...
public static async connectWallet(): Promise<null | string> {
    // If MetaMask is not available, return not successful...
    if (!window.ethereum) { return null; }
    const provider = window.ethereum;

    // Try to connect wallet...
    try  {
        const arrAddress: any = await provider.request({ method: "eth_requestAccounts"
});
        if (!arrAddress || arrAddress.length <= 0) { return null; }
        return arrAddress[0];
    } catch(err) { return null; }
}
```

We first check if MetaMask has injected the `window.ethereum` API. If not, we will not be able to use the API and so we will just return `null` to indicate that we could not connect to a wallet.

We then call the `eth_requestAccounts` API, which gives us an array of addresses. If there is at least one address, we will return the first one.

9.1.3. getCurrentWallet

WalletUtils.ts

```
// Get the currently-connected wallet...
public static async getCurrentWallet(): Promise<null | string> {
    // If MetaMask is not available, return not successful...
    if (!window.ethereum) { return null; }
    const provider = window.ethereum;

    // Try to get currently-connected wallet...
    try  {
        const arrAddress: any = await provider.request({ method: "eth_accounts", });
        if (!arrAddress || arrAddress.length <= 0) { return null; }
        return arrAddress[0];
    } catch(err) { return null; }
}
```

`getCurrentWallet` is useful in case the user has already connected their wallet to our app and perhaps refreshed the page or something. Rather than trying to connect again, we can just grab the current wallet that is connected by using the `eth_accounts` API.

Again, we are using `null` to indicate that no wallet address was found.

9.1.4. switchToNetwork

WalletUtils.ts

```
// Switch to the specified network, or add it if it doesn't exist...
public static async switchToNetwork(chainId: number, rpcUrl: string): Promise<void> {
    // If MetaMask is not available, return not successful...
    if (!window.ethereum) { return; }
    const provider = window.ethereum;

    try {
        const sChainId = `0x${chainId.toString(16)}`;
        await provider.request({
            method: 'wallet_switchEthereumChain',
            params: [{ chainId: sChainId }],
        });
    } catch (switchError: any) {
        // This error code indicates that the chain has not been added to MetaMask.
        if (switchError.code === 4902) {
            await WalletUtils.addNetwork(chainId, rpcUrl);
        }
    }
}
```

`switchToNetwork` is useful in the event that a user has connected their wallet to our application, but is currently on the wrong network. In this case, we can allow them to switch to the correct network that our application needs to work with.

As parameters, we must take the network's Chain ID and RPC URL. We format the Chain ID appropriately and then call `wallet_switchEthereumChain` to attempt to switch to the specified network. This API will only be successful if the user has already added that network to their wallet.

If the user has not added the network to their wallet, the API will throw an error with code `4902`. In this case, we can then add the network for them using the `addNetwork` function described next.

9.1.5. addNetwork

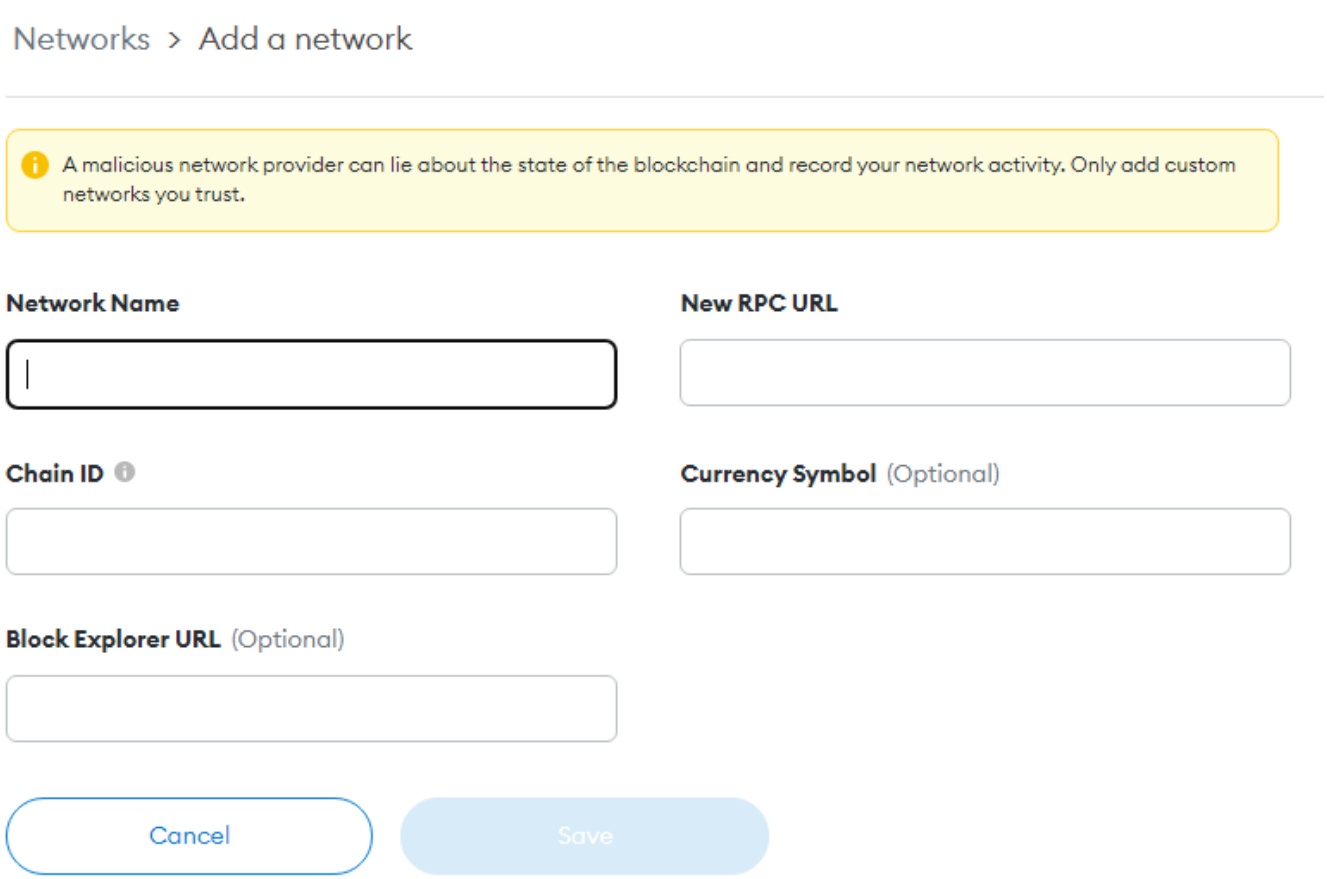
WalletUtils.ts

```
// Add the specified network to the wallet...
public static async addNetwork(chainId: number, rpcUrl: string): Promise<void> {
    // If MetaMask is not available, return not successful...
    if (!window.ethereum) { return; }
    const provider = window.ethereum;

    try {
        const sChainId = `0x${chainId.toString(16)}`;
        await provider.request({
            method: 'wallet_addEthereumChain',
            params: [{ chainId: sChainId, rpcUrl: rpcUrl }],
        });
    } catch (err) { }
}
```

If a user has not yet added the network to their wallet, we can do it for them to make things easier for them. We take the Chain ID and RPC URL for the network and then call `wallet_addEthereumChain` to add the network to their wallet.

This is similar to adding a network in MetaMask:



9.1.6. React Hook

To make this easier to use in a React application, we will wrap up this functionality inside of a React

Hook.

Create a new file called `useWallet.ts`.

useWallet.ts

```
import { useState, useEffect, useCallback } from 'react';
import WalletUtils from './WalletUtils';

export interface IUseWalletProps {
    onWalletAccountChanged?: (address: string) => void;
    onWalletConnected?: (address: string) => void;
    onWalletDisconnected?: () => void;
}

export function useWallet(props: IUseWalletProps) {
    const { onWalletAccountChanged, onWalletConnected, onWalletDisconnected } =
props;
    const [isInitialized, setIsInitialized] = useState<boolean>(false);
    const [isConnected, setIsConnected] = useState<boolean>(false);
    const [address, setAddress] = useState<string>("");

    // Initialize wallet connection...
    useEffect(() => {
        if (!isInitialized) {

            setIsInitialized(true);
            WalletUtils.getCurrentWallet().then((address) => {
                // If we found an address...
                if (!!address) {
                    // Set state...
                    setAddress(address);
                    setIsConnected(true);

                    // Call client function...
                    if (onWalletConnected) { onWalletConnected(address ?? ""); }
                }
            });
        }
    }, [address, isInitialized, onWalletConnected]);

    // Listener for when a wallet account is changed...
    const _onWalletAccountChanged = useCallback((accounts: any) => {
        // Update state...
        const address = accounts.length > 0 ? accounts[0] : "";
        setAddress(address);
        setIsConnected(accounts.length > 0);

        // Call client function...
        if (onWalletAccountChanged) { onWalletAccountChanged(address); }
    }, [setIsConnected, setAddress, onWalletAccountChanged]);
```

```
45
46      // Listen for wallet changes...
47      useEffect(() => {
48          if (!window.ethereum) { return; }
49
50          // Listen to accountsChanged event...
51          window.ethereum.on("accountsChanged", _onWalletAccountChanged);
52      }, [_onWalletAccountChanged]);
53
54      // Disconnect wallet...
55      const disconnect = useCallback(() => {
56          // Disconnect wallet...
57          setIsConnected(false);
58          setAddress("");
59
60          // Call client function...
61          if (onWalletDisconnected) { onWalletDisconnected(); }
62      }, [setIsConnected, setAddress, onWalletDisconnected]);
63
64      // Connect wallet...
65      const connect = useCallback(async () => {
66          // Connect to wallet...
67          const address = await WalletUtils.connectWallet();
68          setIsConnected(address !== null);
69          setAddress(address ?? "");
70
71          // Call client function...
72          if (onWalletConnected) { onWalletConnected(address ?? ""); }
73      }, [setIsConnected, setAddress, onWalletConnected]);
74
75      return { address, isConnected, connect, disconnect };
76  }
```

Now let's review the code and then explain how to use the hook.

```
import { useState, useEffect, useCallback } from 'react';
import WalletUtils from './WalletUtils';
```

Here we are importing the `WalletUtils` functionality that we created earlier, as well as some React APIs we'll be using.

```
export interface IUseWalletProps {
    onWalletAccountChanged?: (address: string) => void;
    onWalletConnected?: (address: string) => void;
    onWalletDisconnected?: () => void;
}
```

We want our hook to be able to report events back to the client, such as when a wallet is connected. The client may want to do something when this happens, such as recalculate a user's balance of ERC-20 tokens. We define an interface with functions that the client can pass into our hook.

```
const [isInitialized, setIsInitialized] = useState<boolean>(false);
const [isConnected, setIsConnected] = useState<boolean>(false);
const [address, setAddress] = useState<string>("");
```

`isInitialized` is used to determine whether we have tried to get the connected wallet or not. This will prevent the code from repeatedly trying to grab the connected wallet after the user has disconnected.

`isConnected` tracks whether the user's wallet is connected or not.

`address` holds the address of the currently-connected wallet.

```
return { address, isConnected, connect, disconnect };
```

Our hook will return some properties and functions that the client can access when they use this hook. `connect` and `disconnect` will allow the client to connect and disconnect a user's wallet.

```
// Listen for wallet changes...
useEffect(() => {
    if (!window.ethereum) { return; }

    // Listen to accountsChanged event...
    window.ethereum.on("accountsChanged", _onWalletAccountChanged);
}, [_onWalletAccountChanged]);
```

We will listen for the `accountsChanged` event, which tells us when the connected wallet has changed. WHen this happens, we should update our `address` state to reflect the newly connected wallet.

You can review the rest of the code on your own. Essentially, we are just calling the functions in `WalletUtils` and then updating the `state` accordingly.

9.1.7. useWallet Usage

Now our hook can be used as simply as...

```
const wallet = useWallet();

//
// ...
//

await wallet.connect();
if (wallet.isConnected) { console.log("Connected!"); }
```

9.2. ERC-20 Interaction

ERC20Utils.ts

```
export class ERC20Utils {
    public static approve(abiToken: string, addrToken: string,
        addrSpender: string, amount: number): Promise<void> {

    }

    public static getBalance(addrToken: string, addrWallet: string): Promise<void> {
        if (typeof window.ethereum == "undefined") { return null; }

        const provider = new ethers.providers.Web3Provider(window.ethereum);
        const erc20 = new ethers.Contract(addrERC20, IERC20.abi, provider);
        try {
            const balance = await erc20.balanceOf(addrWallet);
            return balance;
        } catch(err) {
            return null;
        }
    }

    public static getAllowance(addrToken: string, addrOwner: string, addrSpender:
string): Promise<void | null> {
        if (typeof window.ethereum == "undefined") { return null; }

    }

}
```

ERC721Utils.ts

```
// Calls setApprovalForAll() for {addrApproval} for the given ERC721 token...
export const approveERC721Token = async (abiToken, addrToken, addrApproval) => {
    if (typeof window.ethereum == "undefined") { return { success: false, status:
"Ethereum not defined" }; }

    // Get contract...
    const signer = (new ethers.providers.Web3Provider(window.ethereum)).getSigner();
    const contract = new ethers.Contract(addrToken, abiToken, signer);

    // Send transaction...
    try {
        const txHash = await contract.setApprovalForAll(addrApproval, true);
        return { success: true, status: "Check out your transaction: " + txHash }
    } catch (error) {
        return { success: false, status: "Something went wrong: " + error.message }
    }
};

// Calls setApprovalForAll() for {addrApproval} for the given ERC721 token...
export const isERC721ApprovedForAll = async (addrERC721, owner, operator) => {
    if (typeof window.ethereum == "undefined") { return false; }

    const provider = new ethers.providers.Web3Provider(window.ethereum);
    const erc721 = new ethers.Contract(addrERC721, IERC721.abi, provider);
    try {
        const isApproved = await erc721.isApprovedForAll(owner, operator);
        return isApproved;
    } catch(err) {
        return false;
    }
};
```

For the approve method, we will need the following

TODO: We don't need to pass in abiToken

Parameters

abiToken	The ABI of the token
addrToken	Address of the ERC-20 token
addrSpender	Address of the spender of this token
amount	Amount you are approving the spender to spend on your behalf

9.3. ERC-721 Interaction

ERC721Utils.ts

```
export class ERC721Utils {

}
```

Chapter 10. User Authentication

Chapter 11. Firebase Functions

Chapter 12. Database Security

Chapter 13. Deployment

Chapter 14. Miscellaneous

- Connecting a wallet
- ERC-721 API
- ERC-20 API
- Format wallet address function
- Mention the JSON interface stuff (artifacts)

Importance of UNIT TESTING Using TESTNETS wei vs ETH / decimals

Setup stuff… - Setup MetaMask Wallet - Add networks

Chapter 15. Part 2

Chapter 16. Sample App: NFT Minting Site

Chapter 17. Sample App: Coin Exchange

Appendix A: Code Reference

A.1. WalletUtils

WalletUtils.ts

```
import { MetaMaskInpageProvider } from "@metamask/providers";

declare global {
    interface Window {
        ethereum: MetaMaskInpageProvider;
    }
}

export default class WalletUtils {
    // Connect user's wallet...
    public static async connectWallet(): Promise<null | string> {
        // If MetaMask is not available, return not successful...
        if (!window.ethereum) { return null; }
        const provider = window.ethereum;

        // Try to connect wallet...
        try  {
            const arrAddress: any = await provider.request({ method:
"eth_requestAccounts" });
            if (!arrAddress || arrAddress.length <= 0) { return null; }
            return arrAddress[0];
        } catch(err) { return null; }
    }

    // Get the currently-connected wallet...
    public static async getCurrentWallet(): Promise<null | string> {
        // If MetaMask is not available, return not successful...
        if (!window.ethereum) { return null; }
        const provider = window.ethereum;

        // Try to get currently-connected wallet...
        try  {
            const arrAddress: any = await provider.request({ method:
"eth_accounts", });
            if (!arrAddress || arrAddress.length <= 0) { return null; }
            return arrAddress[0];
        } catch(err) { return null; }
    }

    // Switch to the specified network, or add it if it doesn't exist...
    public static async switchToNetwork(chainId: number, rpcUrl: string): Promise
<void> {
        // If MetaMask is not available, return not successful...
```

```
        if (!window.ethereum) { return; }
        const provider = window.ethereum;

        try {
            const sChainId = `0x${chainId.toString(16)}`;
            await provider.request({
                method: 'wallet_switchEthereumChain',
                params: [{ chainId: sChainId }],
            });
        } catch (switchError: any) {
            // This error code indicates that the chain has not been added to MetaMask.
            if (switchError.code === 4902) {
                await WalletUtils.addNetwork(chainId, rpcUrl);
            }
        }
    }

    // Add the specified network to the wallet...
    public static async addNetwork(chainId: number, rpcUrl: string): Promise<void> {
        // If MetaMask is not available, return not successful...
        if (!window.ethereum) { return; }
        const provider = window.ethereum;

        try {
            const sChainId = `0x${chainId.toString(16)}`;
            await provider.request({
                method: 'wallet_addEthereumChain',
                params: [{ chainId: sChainId, rpcUrl: rpcUrl }],
            });
        } catch (err) { }
    }

    // Adds the specified ERC-20 token to user's wallet...
    public static async addERC20Token(address: string, symbol: string, decimals: number): Promise<void> {
        // If MetaMask is not available, return not successful...
        if (!window.ethereum) { return; }
        const provider = window.ethereum;

        try {
            const options = { address: address, symbol: symbol, decimals: decimals };
            await provider.request({
                method: 'wallet_watchAsset',
                params: { type: 'ERC20', options: options }
            });
        } catch (err) { }
    }
}
```

A.2. useWallet

useWallet.ts

```ts
import { useState, useEffect, useCallback } from 'react';
import WalletUtils from './WalletUtils';

export interface IUseWalletProps {
    onWalletAccountChanged?: (address: string) => void;
    onWalletConnected?: (address: string) => void;
    onWalletDisconnected?: () => void;
}

export function useWallet(props: IUseWalletProps) {
    const { onWalletAccountChanged, onWalletConnected, onWalletDisconnected } =
props;
    const [isInitialized, setIsInitialized] = useState<boolean>(false);
    const [isConnected, setIsConnected] = useState<boolean>(false);
    const [address, setAddress] = useState<string>("");

    // Initialize wallet connection...
    useEffect(() => {
        if (!isInitialized) {

            setIsInitialized(true);
            WalletUtils.getCurrentWallet().then((address) => {
                // If we found an address...
                if (!!address) {
                    // Set state...
                    setAddress(address);
                    setIsConnected(true);

                    // Call client function...
                    if (onWalletConnected) { onWalletConnected(address ?? ""); }
                }
            });
        }
    }, [address, isInitialized, onWalletConnected]);

    // Listener for when a wallet account is changed...
    const _onWalletAccountChanged = useCallback((accounts: any) => {
        // Update state...
        const address = accounts.length > 0 ? accounts[0] : "";
        setAddress(address);
        setIsConnected(accounts.length > 0);

        // Call client function...
        if (onWalletAccountChanged) { onWalletAccountChanged(address); }
    }, [setIsConnected, setAddress, onWalletAccountChanged]);

    // Listen for wallet changes...
```

```
47      useEffect(() => {
48          if (!window.ethereum) { return; }
49
50          // Listen to accountsChanged event...
51          window.ethereum.on("accountsChanged", _onWalletAccountChanged);
52
53          // Clean up listener when finished...
54          return () => { window.ethereum.removeListener('accountsChanged',
    _onWalletAccountChanged); };
55      }, [_onWalletAccountChanged]);
56
57      // Disconnect wallet...
58      const disconnect = useCallback(() => {
59          // Disconnect wallet...
60          setIsConnected(false);
61          setAddress("");
62
63          // Call client function...
64          if (onWalletDisconnected) { onWalletDisconnected(); }
65      }, [setIsConnected, setAddress, onWalletDisconnected]);
66
67      // Connect wallet...
68      const connect = useCallback(async () => {
69          // Connect to wallet...
70          const address = await WalletUtils.connectWallet();
71          setIsConnected(address !== null);
72          setAddress(address ?? "");
73
74          // Call client function...
75          if (onWalletConnected) { onWalletConnected(address ?? ""); }
76      }, [setIsConnected, setAddress, onWalletConnected]);
77
78      return { address, isConnected, connect, disconnect };
79  }
```