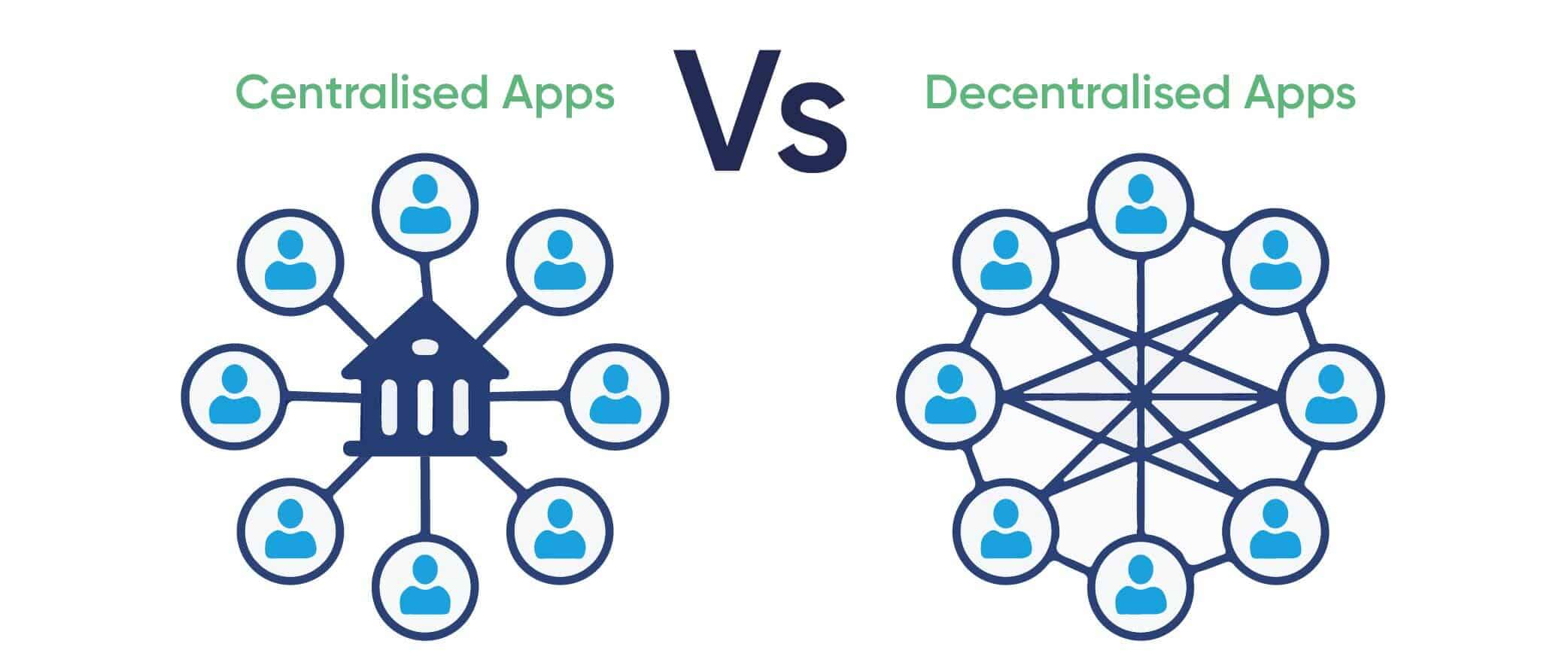
**What is a Blockchain?**

A Blockchain is a digital ledger of transactions that is **secure**, **immutable**, and **decentralized**. It consists of a chain of blocks and each block contains a secure group of transactions of money, bitcoins, contracts, property, etc. without any need for third-party middle-man such as central authorities, banks, government, etc. Transactions are verified by a network of computers. Once a block of information is created in the chain, it can’t be changed or deleted. This makes the blockchain very secure and trustworthy



Instead of having a network, a central server, and a database, *the blockchain is a network and a database all in one*. A blockchain is a peer-to-peer network of computers, called nodes, that share all the data and the code in the network. So, if you’re a device connected to the blockchain, you are a node in the network, and you talk to all the other computer nodes in the network. You now have a copy of all the data and the code on the blockchain. There are no more central servers. Just a bunch of computers that talk to one another on the same network

A table with text on it

Description automatically generated

**What is a Smart Contract?**

Ethereum blockchain allows us to execute code with the Ethereum Virtual Machine (EVM) on the blockchain with something called a smart contract.

Smart contracts are where all the business logic of our application lives. This is where we’ll actually code the decentralized portion our app. Smart contracts are in charge of reading and writing data to the blockchain, as well as executing business logic. Smart contacts are written in a programming language called Solidity, which looks a lot like JavaScript.

A diagram of a browser and a cloud

Description automatically generated

**what is Solidity**

Solidity is a high-level, statically-typed programming language designed primarily for developing smart contracts on the Ethereum blockchain and other blockchain platforms. It was developed by the Ethereum Project and is influenced by languages such as JavaScript, Python, and C++. Solidity is used to write smart contracts, which are self-executing contracts where the terms are directly written into code. These contracts enable developers to create decentralized applications (DApps) that can run on the Ethereum Virtual Machine (EVM).

Key features of Solidity include:

* **Contract-oriented**: Solidity focuses on the development of smart contracts.
* **Static typing**: The type of each variable should be known at compile time.
* **Inheritance**: Solidity supports multiple inheritance, allowing contracts to inherit properties and functions from other contracts.
* **Libraries**: Solidity allows the use of libraries to reuse code and create modular and maintainable codebases.
* **Events**: Solidity provides a way to log events, which can be used to trigger actions on the client-side applications.
* **Function modifiers**: These are used to modify the behaviour of functions, such as access control.

**Sample Hello World Program**

**A screenshot of a computer

Description automatically generated**

*pragma solidity >=0.8.2 <0.9.0;*

*contract HelloWorld {*

*string private str = "Hello World";*

*function get() public view returns (string memory) {*

*return str;*

*}*

*}*

**What is a general ERC 20 token**

An ERC-20 is a standard for creating and issuing tokens on the Ethereum blockchain. It defines a common list of rules that all Ethereum tokens must adhere to, ensuring compatibility with various platforms, wallets, and exchanges. ERC-20 tokens are fungible, meaning each token is identical in type and value.

The ERC-20 standard outlines a set of six mandatory functions that a smart contract must implement, as well as three optional functions and two events.

Here is an overview of the key components of the ERC-20 standard:

**Mandatory Functions**

1. **totalSupply()**: Returns the total supply of tokens.
2. **balanceOf(address account)**: Returns the balance of tokens for a specific address.
3. **transfer(address recipient, uint256 amount)**: Transfers a specified amount of tokens to a specified address.
4. **transferFrom(address sender, address recipient, uint256 amount)**: Transfers a specified amount of tokens from one address to another, typically used for allowances.
5. **approve(address spender, uint256 amount)**: Allows a spender to withdraw a set amount of tokens from the owner's account.
6. **allowance(address owner, address spender)**: Returns the remaining number of tokens that a spender is allowed to withdraw from the owner's account.

**Events**

1. **Transfer(address indexed from, address indexed to, uint256 value)**: Emitted when tokens are transferred.
2. **Approval(address indexed owner, address indexed spender, uint256 value)**: Emitted when the allowance of a spender is set by a call to approve.

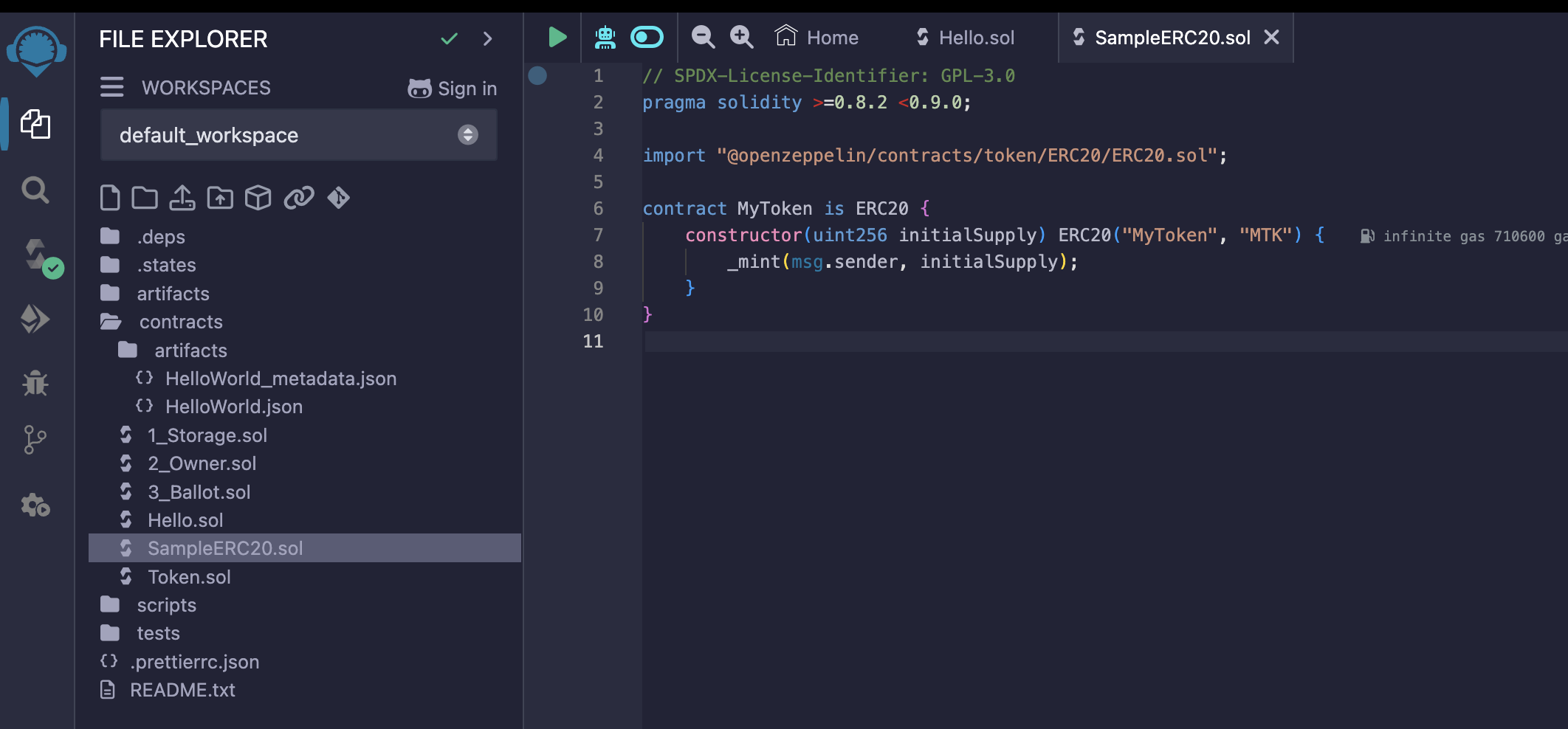
**Optional Functions**

These are not required by the ERC-20 standard but are commonly included:

1. **name()**: Returns the name of the token.
2. **symbol()**: Returns the symbol of the token.
3. **decimals()**: Returns the number of decimals the token uses.

ERC20 code by OpenZeppelin:- <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol>

Sample code to develop ERC20 token



*pragma solidity ^0.8.0;*

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

*contract MyToken is ERC20 {*

*constructor(uint256 initialSupply) ERC20("MyToken", "MTK") {*

*\_mint(msg.sender, initialSupply);*

*}*

*}*

**What is eHKD token**

The e-HKD (electronic Hong Kong Dollar) refers to a proposed digital currency issued by the Hong Kong Monetary Authority (HKMA). It is part of a broader initiative to explore and potentially implement central bank digital currencies (CBDCs). The e-HKD aims to modernize the financial system, enhance payment efficiency, and provide an additional option for digital transactions in Hong Kong

This CBDC (Central Bank Digital Currency) smart contract extends the ERC-1365 standard to enable the implementation of a digital currency system. The contract is designed to allow for the creation, management, and distribution of a digital currency (CBDC) on the blockchain. It includes the functionality of an **allowlist** and **denylist** system, enabling control over which addresses can interact with the token and under which conditions, such as freezing or restricting transfers. Additionally, the contract supports governance and administrative control over token distributions, ensuring security, compliance, and regulatory oversight.

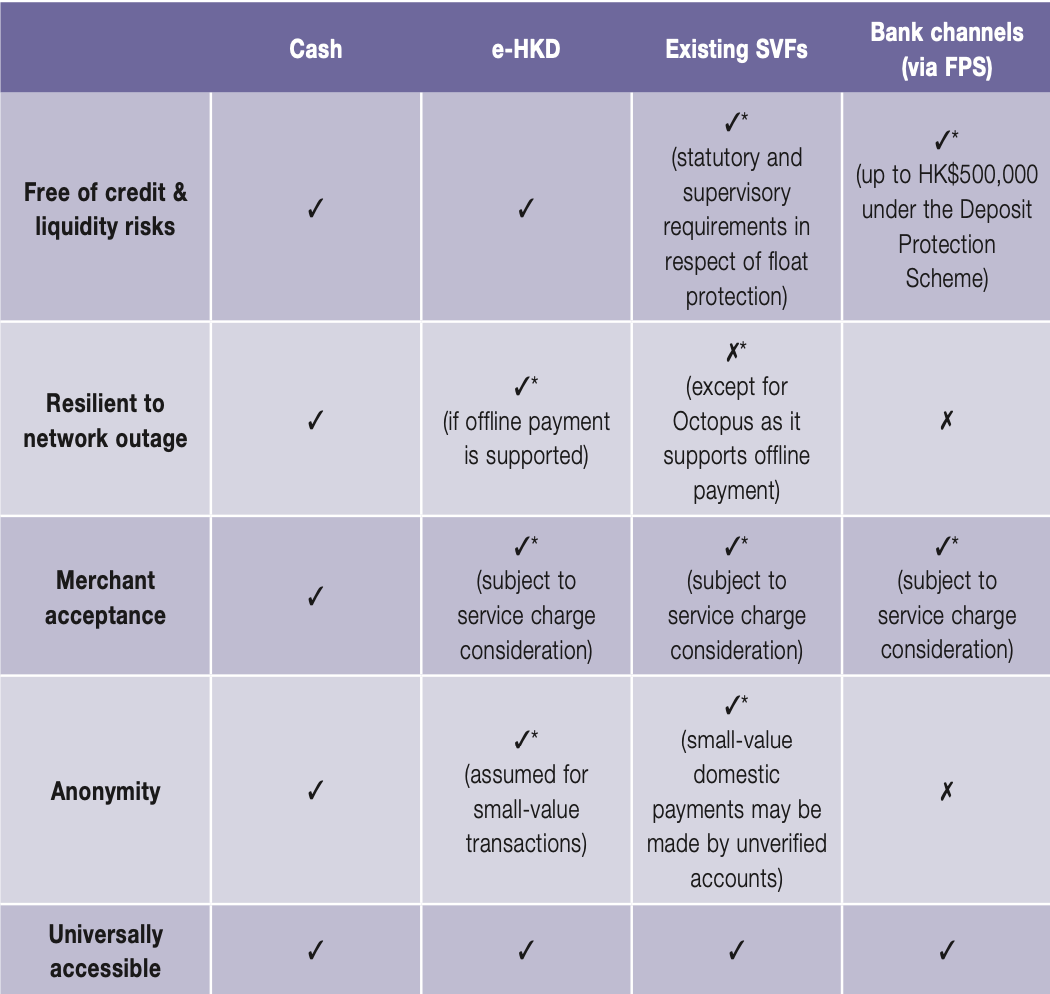
**Benefits of eHKD**

1. **Improving the availability and usability of central bank money** : - CBDCs can provide financial services to unbanked or underbanked populations by offering a digital alternative to cash. Also it can facilitate seamless, instantaneous transactions without the need for intermediaries, reducing transaction times and enhancing convenience for users
2. **Positioning for the challenges of new forms of money** :- As the landscape of money and financial transactions evolves with the advent of new forms of money such as cryptocurrencies, stablecoins, and Central Bank Digital Currencies (CBDCs), central banks, financial institutions, and regulators need to position themselves effectively to address the associated challenges
3. **Supporting innovation and meeting future payment needs in a digital economy** :- This can foster an ecosystem of innovation as fintech companies explore new ways to leverage CBDCs for payments, lending, savings, and more. CBDCs can enable real-time, instant settlement of transactions, reducing the time and cost associated with traditional payment systems. For example, it could support smart contracts (i.e. protocols that self-execute when certain conditions are met) to facilitate automated payment (e.g. on insurance policies such that travellers can receive compensation automatically in the event of flight delay with the fulfilment of the specified pre-condition(s))

A diagram of a travel agency

Description automatically generated with medium confidence

1. **Improving resilience and efficiency of the payment system :-** The introduction of e-HKD may create a perception of intensifying the competition with other payment options, it is worth noting that the objective of introducing e-HKD is to provide consumers with an additional payment option that may address certain limitations of the existing payment options (Table 1), instead of removing or becoming a substitute for these options.

****

1. **Reinforcing the transmission of monetary policy:-** With CBDCs, central banks can implement monetary policy changes more swiftly and with immediate effect. CBDCs give central banks more precise tools for managing the money supply.CBDCs can provide central banks with real-time data on economic transactions, enabling more accurate and timely economic analysis.

**Challenges of eHKD**

1. **Implications for bank funding and its consequences :-** If consumers and businesses prefer holding CBDCs over traditional bank deposits, commercial banks could face a significant reduction in their deposit base. Deposits are a primary source of funding for bank loans. A decrease in deposits due to the adoption of CBDCs could reduce banks' ability to extend credit.
2. **Increasing cyber security and software risks:-** The introduction and widespread adoption of Central Bank Digital Currencies (CBDCs) and other digital financial systems inevitably increase cybersecurity and software risks. Cybercriminals may attempt to breach CBDC systems to steal funds, manipulate data, or disrupt operations. Attackers might launch DDoS attacks to overwhelm the CBDC network, causing service outages and disrupting transactions. The centralized nature of CBDCs could make them a prime target for data breaches, where sensitive user information might be stolen and misused
3. **Increasing economic vulnerability to power/network outages:-** Like most other types of digital payment systems, transactions of CBDC would still rely on the availability of electricity and data networks.

**Smart contract functions of eHKD**

1. **Mint** :– Mint function refers to the creation of new units of the CBDC. This function is typically controlled by the central bank or an authorized entity to ensure that the supply of the digital currency remains within regulatory limits.
2. **Burn:-** Burn function refers to the destruction or removal of CBDC units from circulation. This function reduces the total supply of the digital currency
3. **Transfer:-** Transfer is the basic function that allows the movement of CBDC units from one account to another
4. **TransferAndCall:-** TransferAndCall function is an advanced function that not only transfers CBDC units but also triggers a subsequent action (or call) in the recipient's smart contract
5. **Pause/Unpause:-** Pause/Unpause functions allow the central bank or an authorized entity to temporarily halt or resume the execution of certain smart contract functions, such as transfers.
6. AddRules/RemoveRules/CheckRules
7. Permission control/ deny list
8. Admin roles control
9. Upgradable (The e-HKD smart contract will be operated behind a proxy contract. When a new version of the e-HKD is released, the admin can update the proxy contract to point to the new implementation while preserving the original data.)

**Guide to develop full stack dApp with eHKD**

A Decentralized Application (dApp) is a software application that runs on a decentralized computing system, typically a blockchain. Unlike traditional applications that run on centralized servers, dApps operate on a peer-to-peer network, which means they are not controlled by any single entity or individual.

**Prerequisites**

* Node.js (v12 or later)
* npm (Node package manager)
* MetaMask (browser extension for managing Ethereum accounts)

**Step 1: Setting Up Your Development Environment**

* 1. **Initialize a Node.js Project**

First, create a new directory for your project and initialize a new Node.js project.

*mkdir my-dapp*

*cd my-dapp*

*npm init -y*

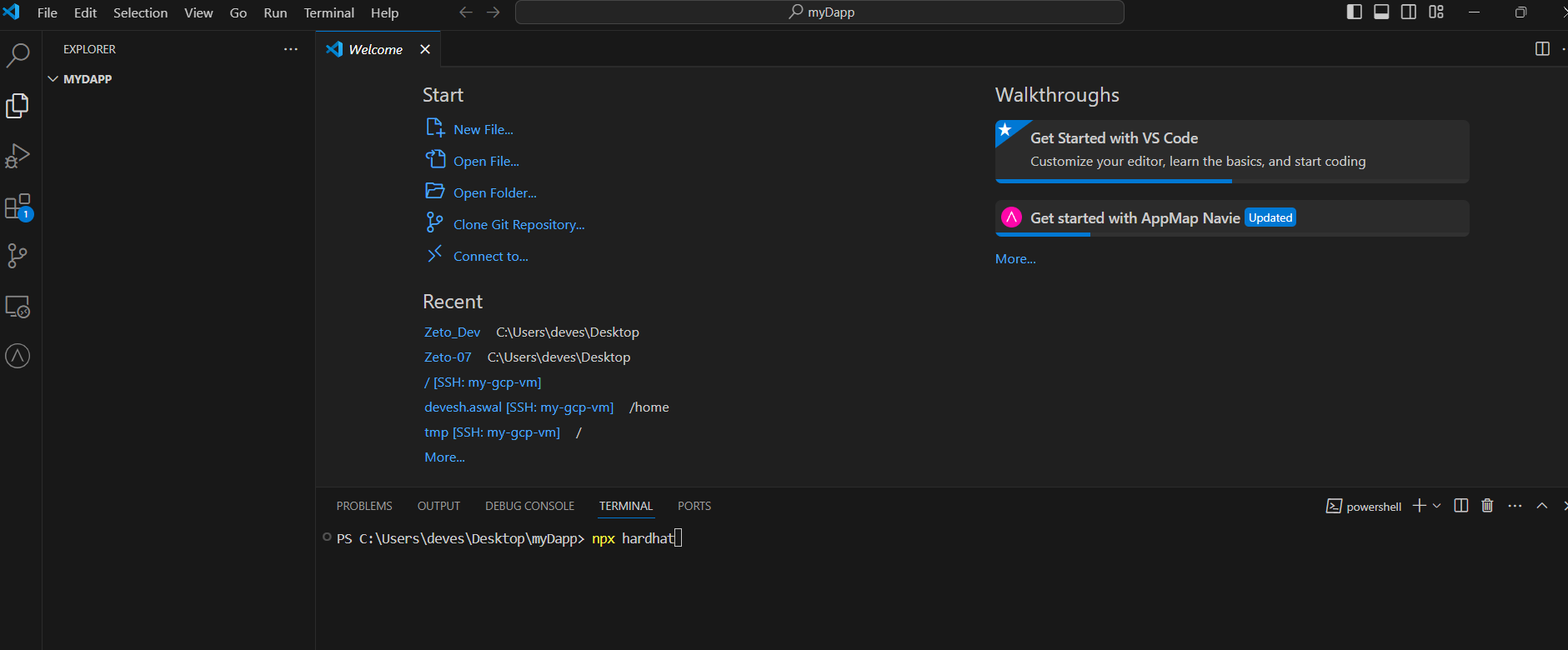
**1.2 Install Hardhat**

Hardhat is a development environment to compile, deploy, test, and debug your Ethereum software. Install Hardhat by running the following command:

*npm install --save-dev hardhat*

Then, run Hardhat to create a new project:

*npx hardhat*

****

*Select below options*

**A screenshot of a computer

Description automatically generated**

*All the related files will be created*

****

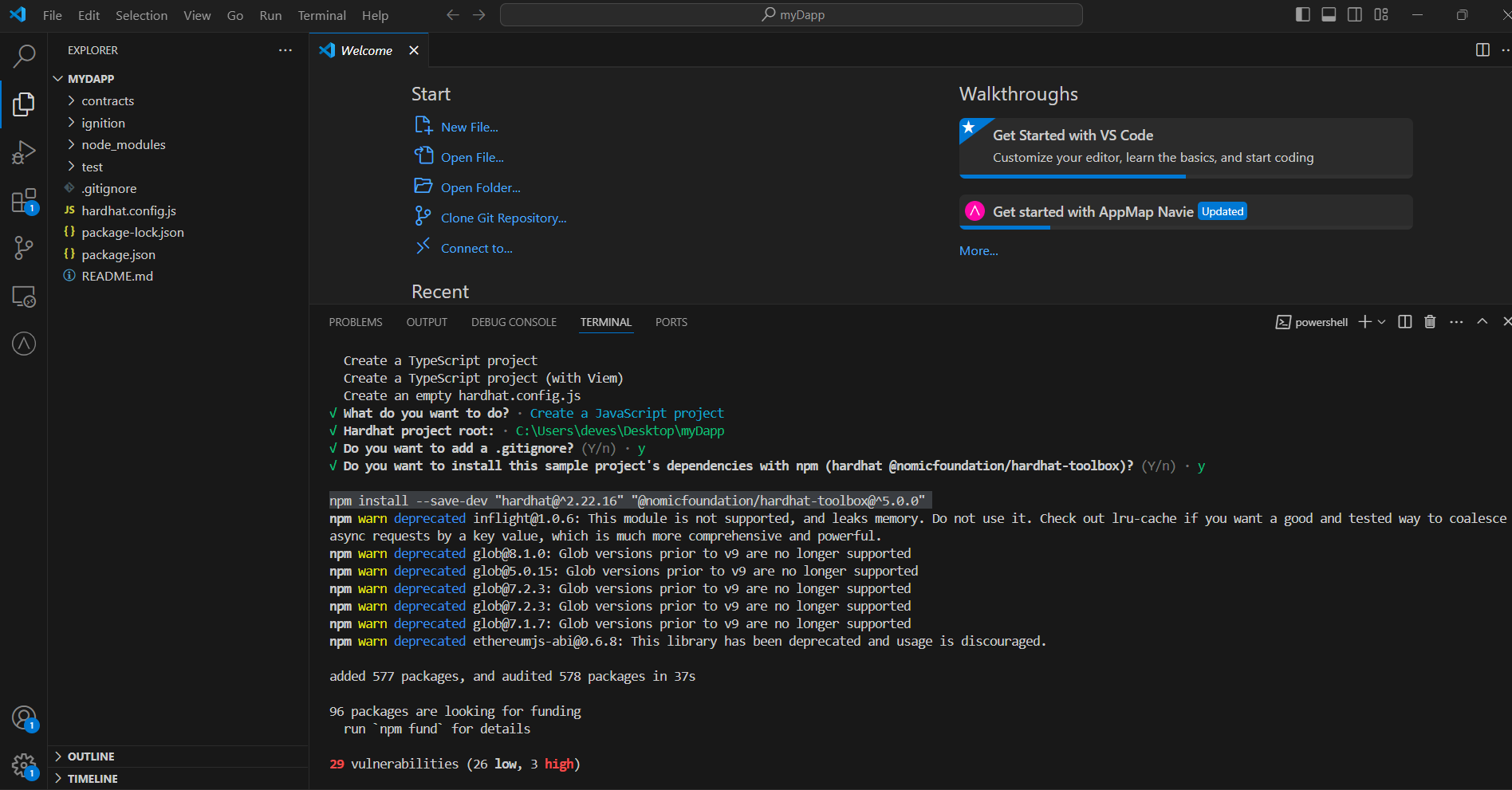
Follow the prompts to create a basic sample project.

**1.3 Install Other Dependencies**

Install the necessary dependencies for developing a DApp:

*npm install --save-dev @nomiclabs/hardhat-ethers ethers*

*npm install --save react react-dom next*

****

**Step 2: Writing Your Smart Contract**

Now under contracts section you will find lock.sol ,it’s a sample smart contract. Change the name and code of it as below.

*// contracts/HelloWorld.sol*

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract HelloWorld {

string private str = "HelloWorld";

function get() public view returns (string memory){

return str;

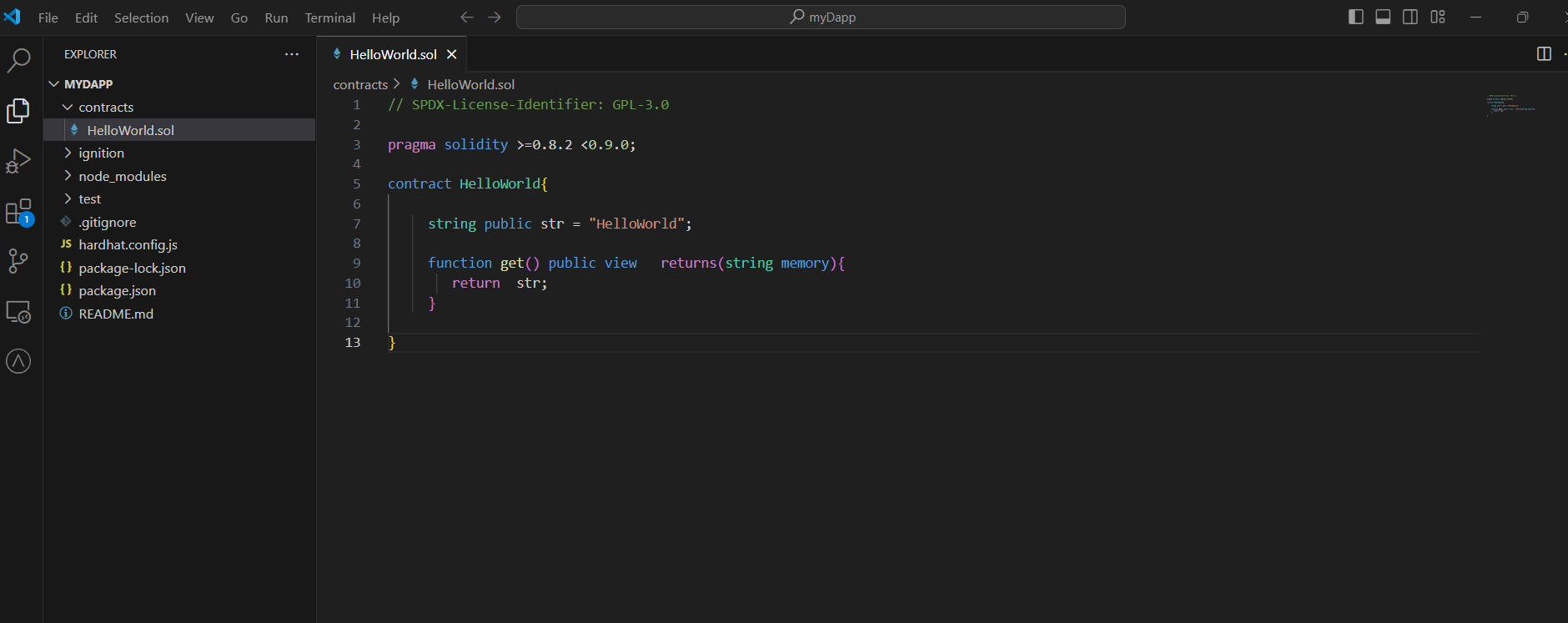
}

function set(string memory \_str) public {

str = \_str;

}

}

****

You can also use remix to write and compile smart contract code. Once your code is compiled successfully in remix then you can add in your project. Below are the steps

* Search “remix ide” in browser and select the first option

A screenshot of a computer

Description automatically generated

* Select “new file” option under contracts to create new smart contract. Create HelloWorld.sol file and write below code.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* Now go to solidity compiler option on left side and compile your smart contract. Error will be displayed below if any.

A screenshot of a computer

Description automatically generated

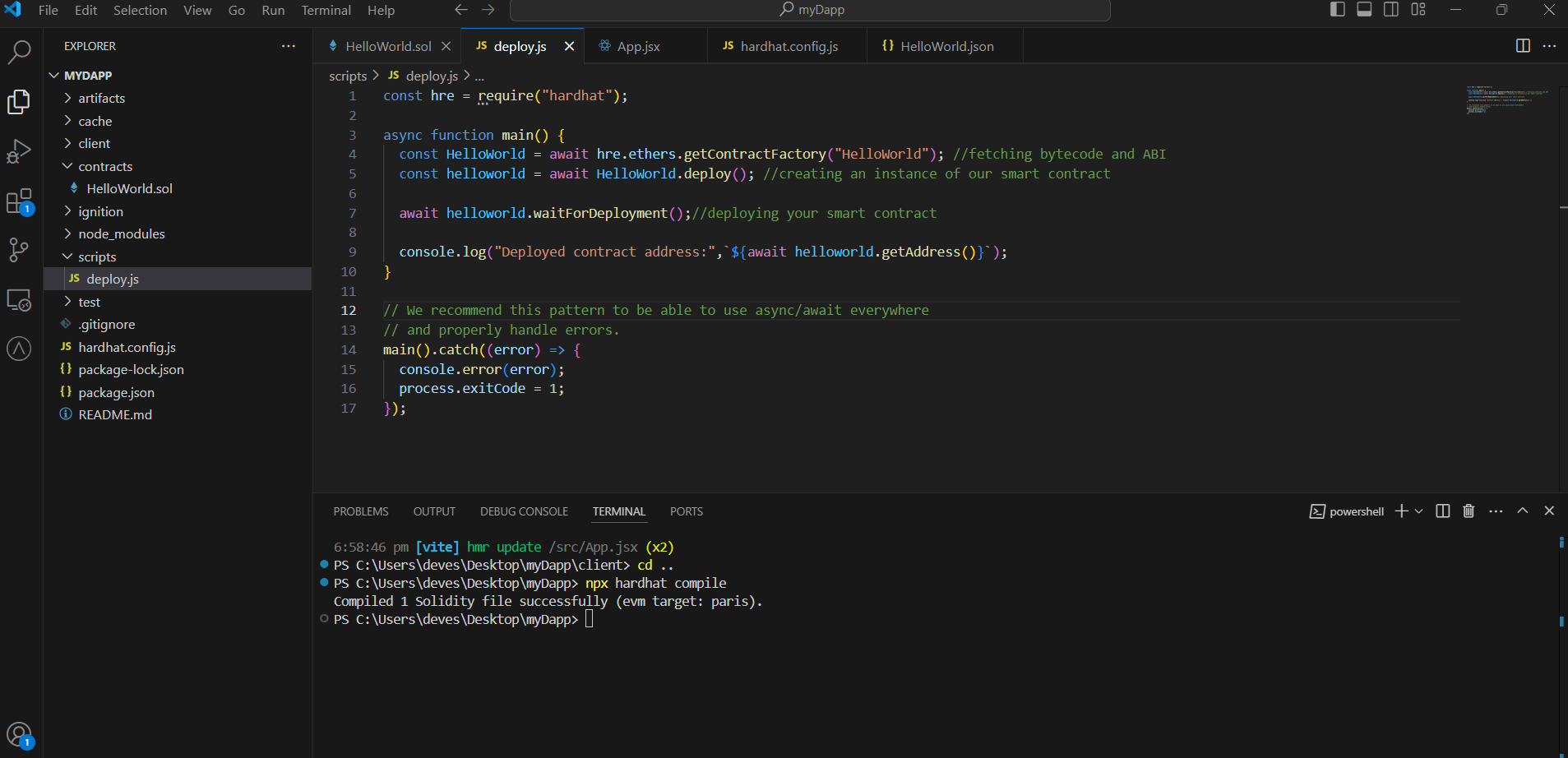
**Step 3: Compiling and Deploying Your Smart Contract**

In Step 2 we complied smart contract using remix but we can do same using hardhat also.

**3.1 Compile the Contract**

To compile your smart contract, run:

npx hardhat compile

****

You will observe artifacts folder is created. It will have Json file which contain bytecode and abi.

**3.2 Deploy the Contract**

Create a deployment script in the scripts directory. In script folder modify deploy.js. Change smart contract name in deploy.js. If script folder is not available, please create manually

**A screen shot of a computer program

Description automatically generated**

*// scripts/deploy.js*

*const hre = require("hardhat");*

*async function main() {*

*const HelloWorld = await hre.ethers.getContractFactory("HelloWorld"); //fetching bytecode and ABI*

*const helloworld = await HelloWorld.deploy(); //creating an instance of our smart contract*

*await helloworld.waitForDeployment();//deploying your smart contract*

*console.log("Deployed contract address:",`${await helloworld.getAddress()}`);*

*}*

*// We recommend this pattern to be able to use async/await everywhere*

*// and properly handle errors.*

*main().catch((error) => {*

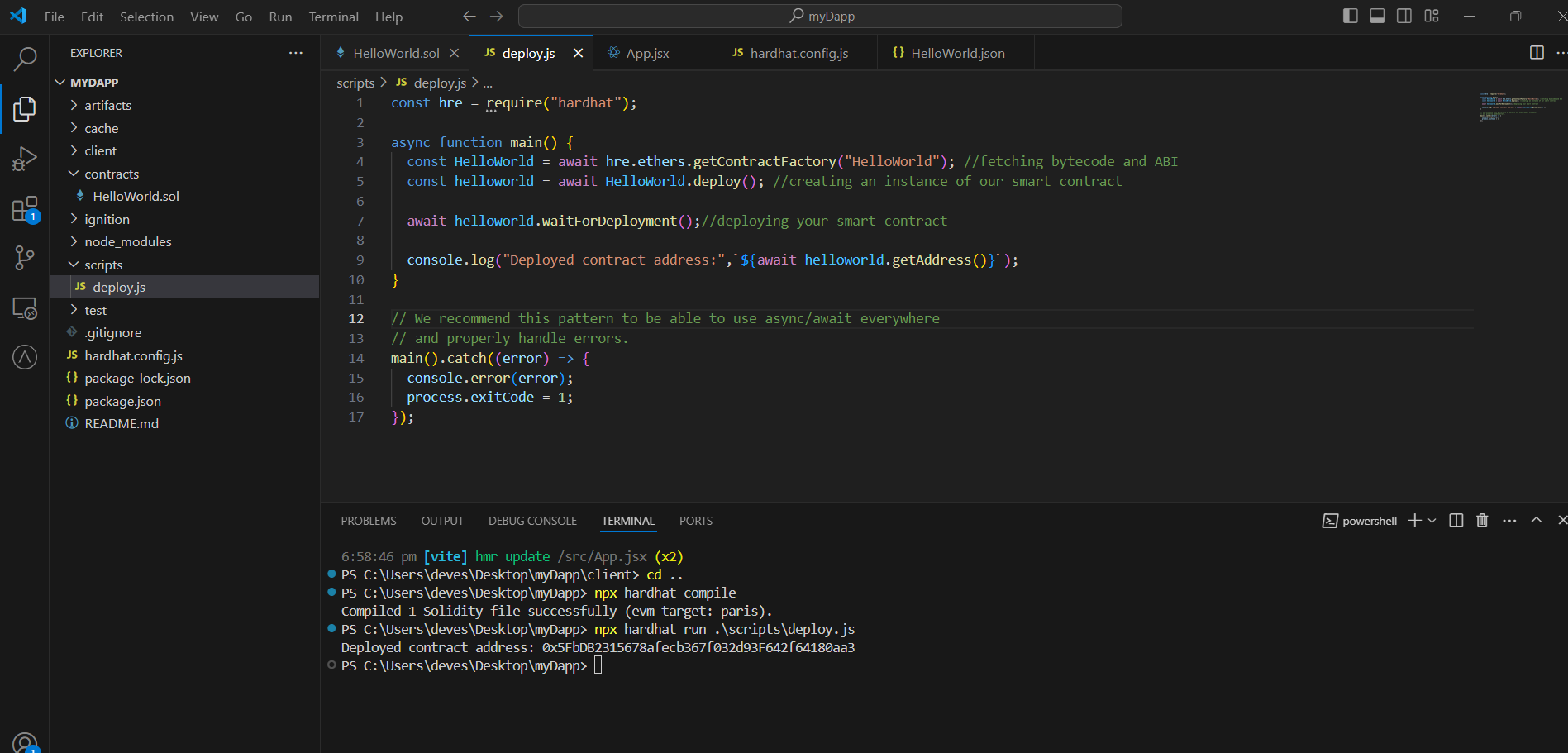
*console.error(error);*

*process.exitCode = 1;*

*});*

To deploy the contract on a local Hardhat network, run:

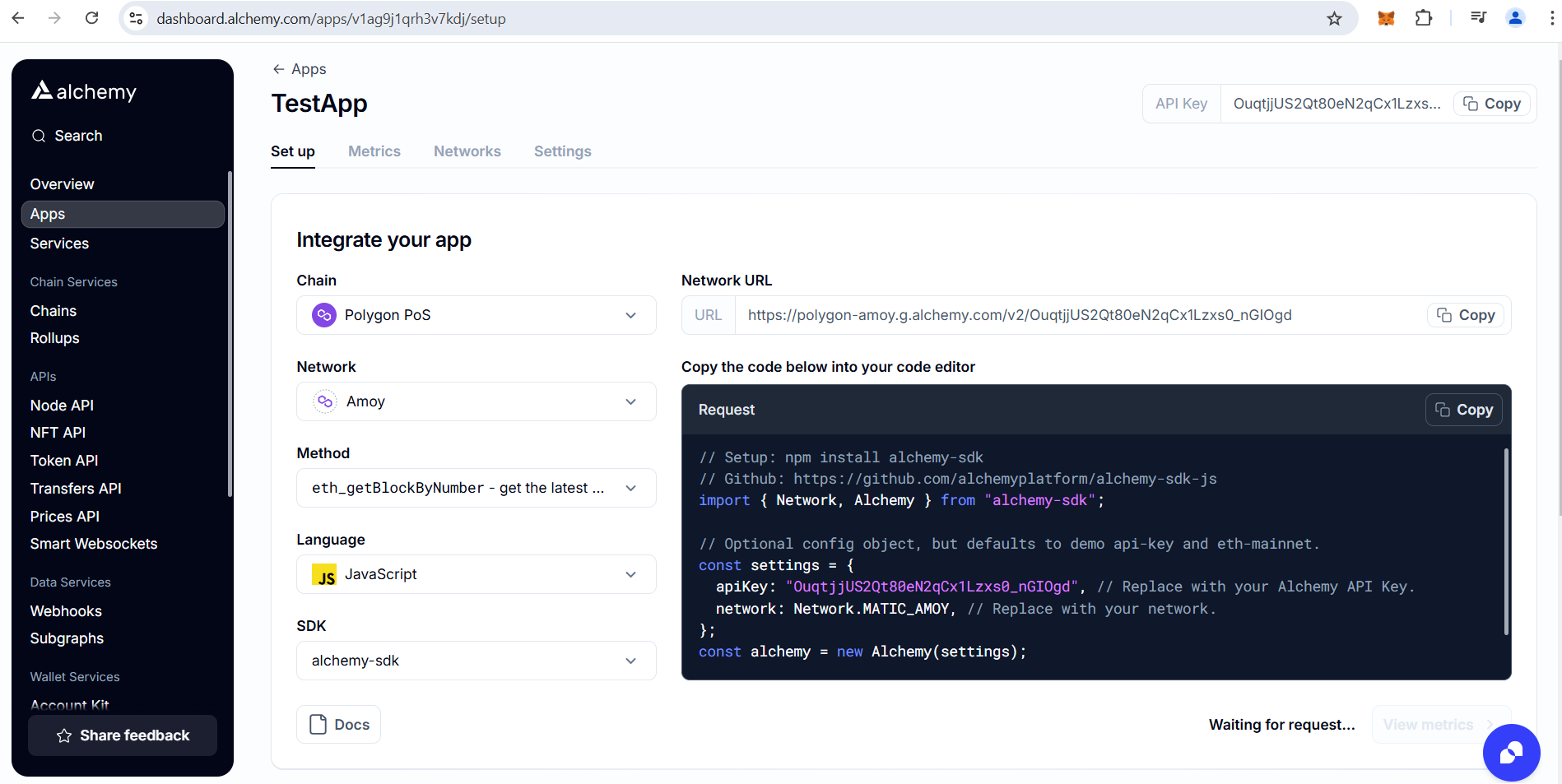
*npx hardhat run scripts/deploy.js*

****

Currently we have deployed our smart contract in local hardhat network. But we want deploy our smart contract in public blockchain test network. I will be deploying on Polygon Amoy , you can deploy on any public blockchain of your choice.

**Step 4: Setting Up Alchemy**

To deploy your contract to a testnet or mainnet, you'll need an Alchemy project. Go to [**https://www.alchemy.com/**](https://www.alchemy.com/), sign up, create a new project with blockchain of your choice, and get the project ID and node RPC endpoint.

****

****

Update your hardhat.config.js to include the Alchemy project ID and your Ethereum account's private key:

*require('@nomiclabs/hardhat-ethers');*

*module.exports = {*

*networks: {*

*polygon: {*

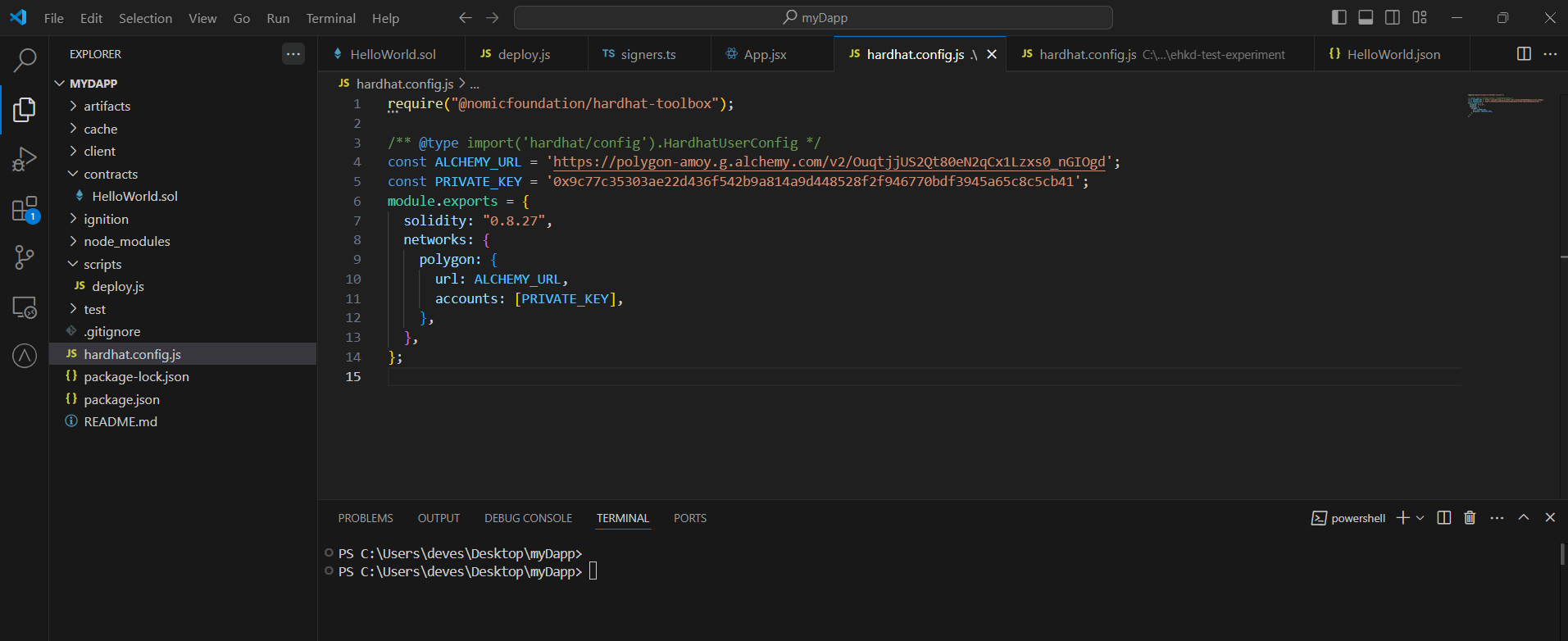
*url: `https://rinkeby.infura.io/v3/YOUR\_ALCHEMY\_PROJECT\_ID`,*

*accounts: [`0x${YOUR\_PRIVATE\_KEY}`],*

*},*

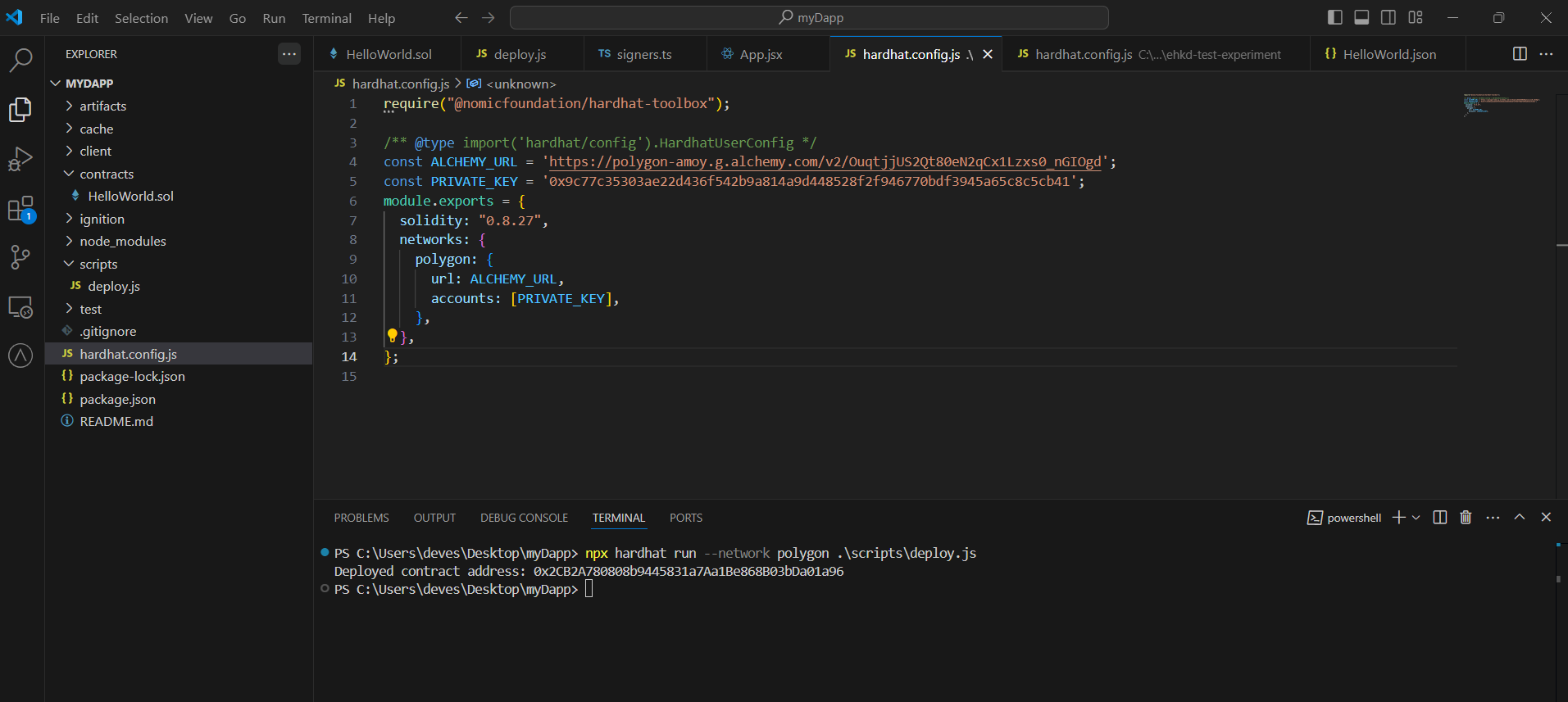
*},*

*};*

****

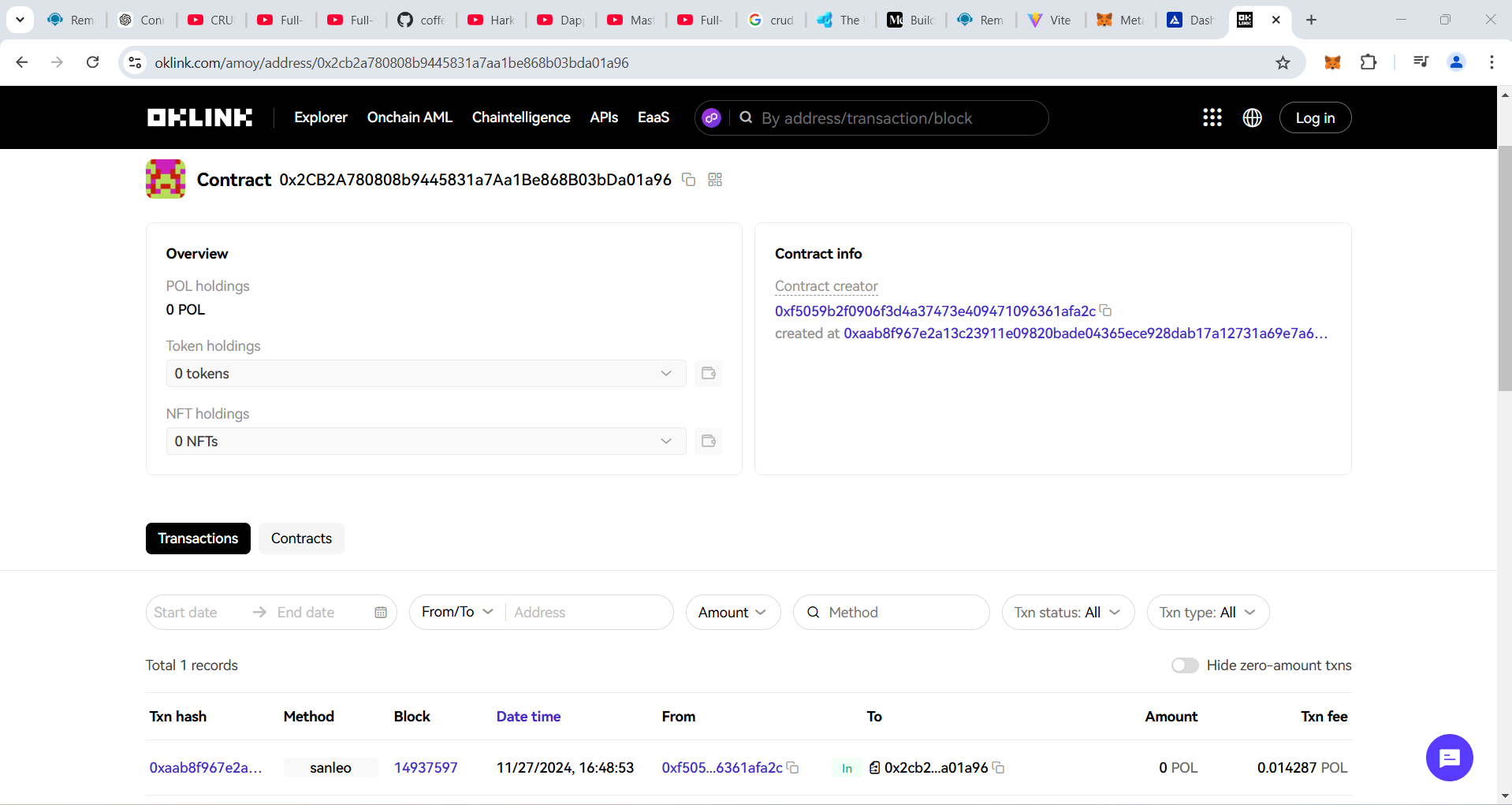
To deploy the contract to the Polygon Amoy testnet, run:

npx hardhat run --network polygon .\scripts\deploy.js

****

Contract will be deployed and you can find contract address in console.

To confirm if the contract is deployed successfully, go to polygon amoy explorer and search with contract address **:-** [**www.oklink.com/amoy**](http://www.oklink.com/amoy)

****

You can verify time and to address, it should be your account address.

**Step 5: Setting Up MetaMask**

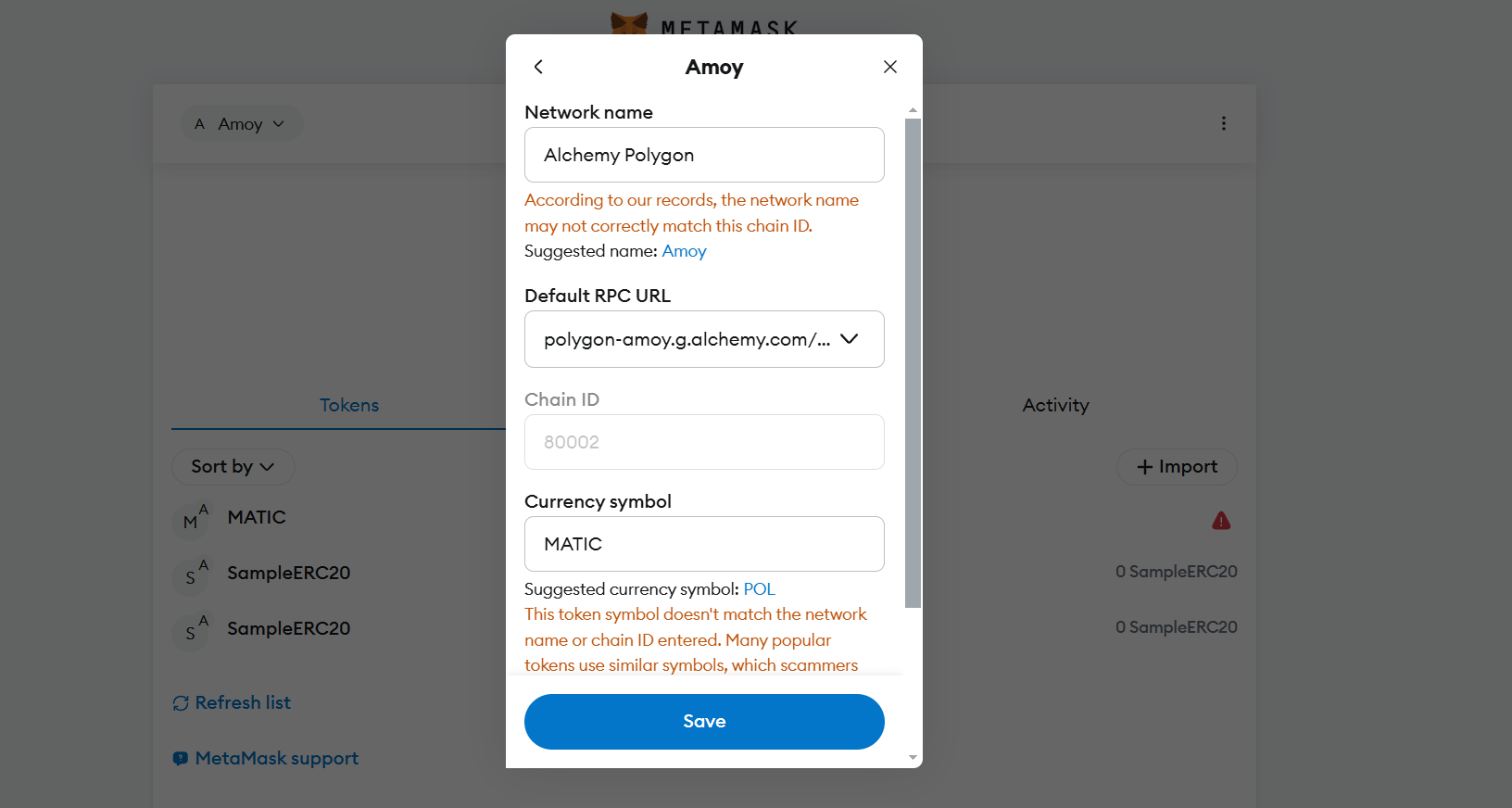
MetaMask is a browser extension that allows you to interact with the Ethereum blockchain. It is essential for managing your Ethereum accounts and for interacting with your DApp.

**5.1 Install MetaMask**

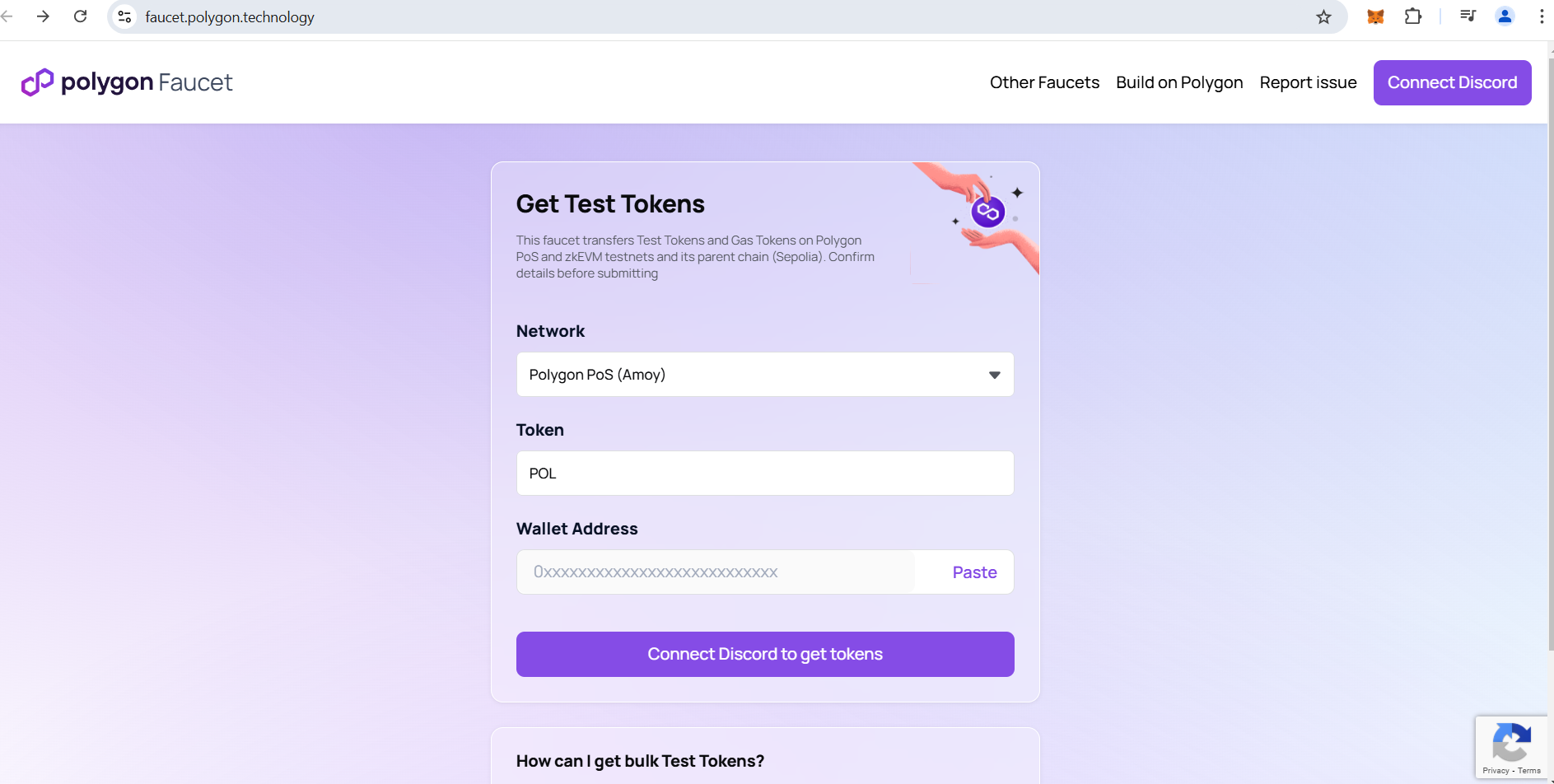
If you haven't already, install the MetaMask extension for your browser from [metamask.io](https://metamask.io/).

**5.2 Configure MetaMask**

1. Open MetaMask and create a new wallet or import an existing one.
2. Connect MetaMask to the Polygon Amoy test network.
   * Click on the network dropdown at the top of the MetaMask window.
   * Select “Add a custom network”
   * Fill the details as below and click on save

****

Initially your balance will be zero. You can get test MATIC from faucet :- https://faucet.polygon.technology/

****

copy your account address from metamask wallet and paste in wallet address field. Faucet will send .2MATIC.

**Step 6: Creating the Frontend with React**

**6.1 Initialize a React App**

Use Create React App to set up the front end:

Npm create vite@latest

**A screen shot of a computer

Description automatically generated**

Vite-project folder is created. I have renamed it to client(optional).

Now run below commands

*cd client*

*npm install*

**6.2 Install ethers.js**

Install ethers.js to interact with the Ethereum blockchain:

*npm install ethers*

**6.3 Connect Frontend to Smart Contract**

In your client folder, update file App.jsx:

*import React, { useState } from "react";*

*import { ethers } from "ethers";*

*import abi from "../../artifacts/contracts/HelloWorld.sol/HelloWorld.json";*

*// Replace these with your contract details*

*const CONTRACT\_ADDRESS = "0x2CB2A780808b9445831a7Aa1Be868B03bDa01a96";*

*const CONTRACT\_ABI = abi;*

*function App() {*

*const [getValue, setGetValue] = useState(""); // For the "Get" text box*

*const [setValue, setSetValue] = useState(""); // For the "Set" text box*

*// Function to handle setting a value on the blockchain*

*const handleSetClick = async () => {*

*try {*

*// Connect to MetaMask*

*if (!window.ethereum) {*

*alert("MetaMask not detected!");*

*return;*

*}*

*const provider = new ethers.BrowserProvider(window.ethereum);*

*// Request accounts from MetaMask (to ensure the user is connected)*

*await provider.send("eth\_requestAccounts", []);*

*const signer = await provider.getSigner();*

*const contract = new ethers.Contract(CONTRACT\_ADDRESS, CONTRACT\_ABI.abi, signer);*

*// Interact with the contract*

*const tx = await contract.set(setValue);*

*await tx.wait(); // Wait for the transaction to be mined*

*alert("Value successfully set on blockchain!");*

*setSetValue(""); // Clear the input box*

*setGetValue("");*

*} catch (error) {*

*console.error("Error setting value:", error);*

*alert("Failed to set value on blockchain.");*

*}*

*};*

*// Function to handle getting a value from the blockchain*

*const handleGetClick = async () => {*

*try {*

*// Connect to MetaMask*

*if (!window.ethereum) {*

*alert("MetaMask not detected!");*

*return;*

*}*

*const provider = new ethers.BrowserProvider(window.ethereum);*

*// Ensure the user is connected to the wallet*

*//await provider.send("eth\_requestAccounts", []);*

*const contract = new ethers.Contract(CONTRACT\_ADDRESS, CONTRACT\_ABI.abi, provider);*

*// Interact with the contract*

*const value = await contract.get();*

*//  console.log("\*\*\*\*\*\*\*\*\*value\*\*\*\*\*\*\*\*",value);*

*setGetValue(value); // Update the "Get" text box*

*} catch (error) {*

*console.error("Error getting value:", error);*

*alert("Failed to fetch value from blockchain.");*

*}*

*};*

*return (*

*<div style={{ padding: "20px", fontFamily: "Arial" }}>*

*<h2>Blockchain Get and Set Example</h2>*

*{/\* Get Section \*/}*

*<div style={{ marginBottom: "10px" }}>*

*<button*

*onClick={handleGetClick}*

*style={{*

*padding: "10px 20px",*

*backgroundColor: "#007BFF",*

*color: "#fff",*

*border: "none",*

*borderRadius: "5px",*

*cursor: "pointer",*

*}}*

*>*

*Get*

*</button>*

*<input*

*type="text"*

*value={getValue}*

*readOnly*

*style={{*

*marginLeft: "10px",*

*width: "300px",*

*padding: "5px",*

*}}*

*/>*

*</div>*

*{/\* Set Section \*/}*

*<div>*

*<button*

*onClick={handleSetClick}*

*style={{*

*padding: "10px 20px",*

*backgroundColor: "#28A745",*

*color: "#fff",*

*border: "none",*

*borderRadius: "5px",*

*cursor: "pointer",*

*}}*

*>*

*Set*

*</button>*

*<input*

*type="text"*

*value={setValue}*

*onChange={(e) => setSetValue(e.target.value)}*

*style={{*

*marginLeft: "10px",*

*width: "300px",*

*padding: "5px",*

*}}*

*/>*

*</div>*

*</div>*

*);*

*}*

*export default App;*

**6.4 Run Your React App**

To start your React app, run:

*npm run dev*

Your DApp should now be running, allowing you to view and update the message stored on the Ethereum blockchain.

**A screenshot of a computer

Description automatically generated**

Click on Get to view current message

**A screenshot of a computer

Description automatically generated**

Now enter new text in set textbox and click on set button. On set we are modify blockchain so we need to pay some gas. So as soon as you click on set metamask will open and request for confirmation .

****

Once you confirm it will take few seconds to update on blockchain, you will get successfully updated pop up.

You can click on get method to check updated value.