Lab 2 – Programmatically Issuing Transactions

COMP6452 Software Architecture for Blockchain Applications
Dilum Bandara

2023 Term 2

1 Learning Outcomes

In this lab, you will learn how to interact with a smart contract using a program. After completing the lab, you will be able to:

- use a standard smart contract within your smart contract
- develop a simple program to interact with a blockchain node
- use the program to deploy a smart contract to a blockchain
- use the program to interact with a smart contract already deployed on a blockchain

2 Introduction

Rather than using Remix IDE and MetaMask to manage interactions with smart contracts, we can develop custom programs to automate the interaction. While you could connect to the Ethereum blockchain using several programming abstractions, the Ethereum Web3.js library [1] is the most popular option. Web3.js is a collection of JavaScript libraries that allow interacting with a local or remote Ethereum node, via an HTTP, WebSocket, or IPC (Inter-Process Communication) connection. It provides an extensive set of functions to compile and deploy a smart contract; sign and issue transactions; check the state of blocks, transactions, and data; and call functions in a smart contract. Web3.js is usually used within Node.js applications. While it is not essential to develop a Node.JS application to interact with smart contracts, it is helpful to know how to deploy smart contracts and issue transactions programmatically. While this lab introduces a relatively simple example, it is highly recommended to follow additional labs such as [2], [3], as they could be helpful in Project 2.

JavaScript is a lightweight, interpreted, or just-in-time compiled programming language. It runs in a web browser and is used to develop web-based applications' front-end. Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on a JavaScript Engine and executes JavaScript code outside a web browser. It is used to build the back-end of scalable network applications. We will develop the program with TypeScript, a strict syntactical superset of JavaScript and add optional static typing to the language to minimise errors. It is designed to develop large applications, and eventually, the code is converted to JavaScript for execution.

The lab has two parts. In Section 3, we still use the Remix IDE to develop the program to deploy and interact with a smart contract. Because it provides a zero-setup environment to develop and execute programs, it is an excellent starting point for first-time developers. Alternatively, you may use an IDE like Visual Studio Code. In Section 4, you will set up everything from installing a test blockchain node to setting up the project structure, developing a complete program, and executing it.

3 Running Scripts with Remix IDE

For this section, use the sample code given in the `Remix` folder of `Lab2_Code.zip` file.

Step 1. Go to https://remix.ethereum.org/. Create a new smart contract file named `MyToken.sol` in the `contracts` folder and copy the sample code.

Click on `Solidity compiler` icon. Then click on `Compile MyToken.sol` button to compile the contract.

Carefully go through the source code to understand what is taking place. In line 5, we import a file named `ERC20.sol`. The ERC-20 token standard is the de-facto standard for fungible tokens in the Ethereum ecosystem. We use the ERC-20 implementation from OpenZeppelin as it is one of the most robust and secure implementations of the standard. It is a good practice to reuse time tested code to minimise bugs and security weaknesses. Note that `@openzeppelin` is the name of the `npm` (node package manager) library that provides access to a bunch of contracts from OpenZeppelin. Glance through the OpenZeppelin documentation at https://docs.openzeppelin.com/contracts/2.x/api/token/erc20 to understand the contract and functions it exposes.

Line 11 defines our contract while inheriting from the ERC20 contract. The constructor (lines 21-23) accepts three arguments as inputs, and two are passed to the parent contract, i.e., ERC20. `_name` and `_symbol` parameters can be used to specify a name and a short symbol for our token. Whereas `_totalSupply` indicates the total number of tokens minted. In line 22, we mint the given number of tokens and assign them to the contract creator.

Step 2. Create a new file in the `scripts` folder and name it `web3-lib-MyToken.ts`. Copy the sample code from the file with the same name.

Read through the Typescript code to understand what it does. For example, `GasHelper` is a helper/utility class (lines 13-24) that calculates a gas limit greater than the estimated gas used. This is useful in ensuring that a transaction sets a sufficiently high gas limit (in line 14, we set it 25% higher) to overcome unexpected gas consumption due to state changes on the ledger between estimating the gas used and inclusion of the transaction in a block.

`getAccounts` (lines 30-32) gets a list of accounts available from the chosen Web3 provider. Web3 provider is an object used to connect with a Ethereum blockchain node. `async` is a keyword that allows the function to contain `await` expressions. Await expressions make promise-returning functions behave as though they are synchronous by suspending execution until the returned promise/result is fulfilled or rejected. The resolved value of the promise is treated as the return value of the await expression (see [4]).

The `deploy` function deploys the specified contract. The function accepts a set of parameters, namely Web3 provider, contract name to deploy, list of constructor parameters, account used to send the smart contract deploy transaction, and an optional gas limit. In lines 55-56, we use the Web3.js function `getGasPrice` to request the current gas price from the connected blockchain node. The node returns estimated gas prices based on the median gas price of the last few blocks. Lines 59-62 create the contracts to be send to the blockchain and set the constructor parameters required while deploying the contract. Finally, lines 65-69 send the smart contract to the blockchain node while setting the sender address, gas limit, and gas price. If the deployment is successful, the transaction receipt returned from the function will contain the newly deployed smart contract's address.

The `call` function invokes a smart contract function that does not update the ledger state. Compared to the `deploy` function, we pass the deployed contract's address, function name to call, and its arguments/parameters to the `call` function. Note that we do not set a gas limit or gas price, as read-only functions (aka view functions) do not consume gas when executing a user-generated transaction.

Line 88 is the key statement within this function, which uses the We3.js `call` function to invoke the given function on the smart contract. `...args` is a special syntax that helps unpack multiple function parameters to be passed into the function. If the transaction is successful, we will receive a transaction receipt containing the transaction hash, and any values returned from the called smart contract function.

In contrast to the `call` function, the `send` executes a smart contract function that updates the ledger state. While the function parameters remain the same, we need to set a gas limit and gas price, as a transaction that updates the ledger state will consume some amount of gas.

Step 3. Create another new file in the `scripts` folder and name it `index-MyToken.ts`. Copy the sample code from the file with the same name.

Read through the Typescript code to understand what it does. Essentially, this file deploys our `MyToken` contract by setting its name, symbol, and total supply. It then invokes a set of getter functions exposed by the ERC20 contract and then transfers token from the contract creator's address to another.

In line 6 we import the `deploy`, `call`, `send`, and `getAccounts` functions from the `web3-lib-MyToken.ts` file. Line 16 sets the Web3 provider, which is set to `Remix VM` as per Step 4 (see next). The `accounts`

variable holds the list of available accounts from the chosen Web3 provider.

Between lines 20-26 we deploy the MyToken contract, while setting the constructor parameters. Lines 29-42 verify that the token symbol and total supply were properly set by invoking the `symbol` and `totalSypply` getter functions. Line 46 invokes the `balanceOf` function on the contract by generating a transaction using the `call` function. In line 54, the `send` function issues a transaction that transfers a set of tokens. Then we check the balances of both `from` and `to` addresses.

Step 4. Click on the `Deploy & run transactions` icon. Set the `Environment` to `Remix VM (Shanghai)` but do not deploy the contract. This allows us to deploy the contract to a test EVM, and use ten pre-funded accounts to issue transactions.

Step 5. Click on the `File explorer` icon, and then select `index-MyToken.ts`. Right click and select `Run` from the pop-up menu.

Note the output of the program in the Remix Console area. Note the smart contract address. Use can use this address in the `Deploy & run transactions` pane to manually interact with the smart contract similar to in Lab 1.

Step 6. Click `Deploy & run transactions` icon. Change the `Environment` to `Injected Provider - Metamask`. Return back to the `File explorer`, and select the file `index-MyToken.ts`. Right click and select `Run` to deploy and execute our smart contract on the Sepolia testnet. Metamask should prompt you to sign and send transactions multiple times.

While the contract gets deployed and you can call the getter functions, other functions will fail as the Remix test accounts we use are not available on the testnet. We will see how these issues can be resolved in the next section.

4 Custom Project Setup

For this section, we will use the sample code given in the `Local` folder of `Lab2_Code.zip` file.

Step 7. Install following prerequisites (if not already installed from Tutorial 1):

- Download and install NodeJS
- Download and install one of the following:
  - Truffle Ganache CLI – Ganache-CLI is preferred over GUI as GUI version has compatibility issues with the latest Ethereum client code. Use `npm install ganache --global` and `ganache` commands to install and start the test client, respectively.
  - Truffle Ganache GUI – Start Ganache. Create a new Ethereum instance while setting the `Chain --> HARDFORK` option to `Berlin`. Start the client. You may need to do 2 small changes in the following Typescript code to work with the Ganache GUI.

Step 8. Chose a suitable folder to create your custom project, say `lab2`. Initialise the repository by executing the following commands using the the shell/terminal (part of the command starting with `#` can be ignored as it is a comment).

```
npm init -y # Initialise NodeJS project
```

```
mv Node.gitignore .gitignore # Move auto generated gitignore file
```

If the `gitignore` file is not auto-generated use one of the following commands to create one (this is useful when saving the project files in a version-controlled code repository like GitHub):

```
wget https://raw.githubusercontent.com/github/gitignore/master/Node.gitignore -O .gitignore # get gitignore
curl https://raw.githubusercontent.com/github/gitignore/master/Node.gitignore -o .gitignore # get gitignore
```

Step 9. Install all required modules using the following:

```
npm install typescript ts-node --save-dev # Add TypeScript developer dependencies
npm install @types/node --save-dev # Add TypeScript node developer dependencies
npm install web3 solc @openzeppelin/contracts fs-extra path # Add Web3.js and Solidity compiler
```

Initialise the project as follows (following is a single line command):

```
npx tsc --init --rootDir src --outDir build --esModuleInterop --resolveJsonModule --lib es6 --module
    commonjs --allowJs true --noImplicitAny true # Configure typescript
```

Instead of all `npm` commands above, you may also initialise the project using `yarn` as follows (make sure `yarn` is already installed on your machine or install it using `npm install --global yarn`):

```
yarn init -y
mv Node.gitignore .gitignore # Move auto generate gitignore file
yarn add -D typescript ts-node
yarn add -D @types/node
yarn add web3 solc @openzeppelin/contracts fs-extra path # Install dependencies
npx tsc --init --rootDir src --outDir build --esModuleInterop --resolveJsonModule --lib es6 --module
    commonjs --allowJs true --noImplicitAny true # Configure typescript
```

Create folders for source code files and binaries as follows:

```
mkdir src && touch src/index.ts # Create source code entry
mkdir contracts # Create folder to hold smart contracts
```

Create two other JSON files to keep track of the Web3 provider and private keys of accounts:

```
mkdir eth_providers && touch eth_providers/providers.json # Create provider entry
mkdir eth_accounts && touch eth_accounts/accounts.json # Create account entry
```

Populate the two files based on the given sample files. However, you will need to change the private keys to match the accounts in your Ganache blockchain node.

If you want to just install the sample files without going through the above project setup steps, run the following commands on the `Local` folder of after extracting the `Lab2_Code.zip` file.

```
npm install
```

Step 10. Create the following files and populate them based on the given sample code (it is highly recommend that you try to understand the code and type in your syntax):

- src/solc-lib.ts
- src/index.ts

`solc-lib.ts` is a new file that did not exist when we used Remix to develop the program. This is because, in Step 1, we used Remix to compile the smart contract and capture its metadata, such as the ABI (Application Binary Interface). However, when we use our program to deploy a smart contract, we must first compile it. This is achieved using lines 52-86, which utilise the Solidity compiler to compile the contract code (line 82). We save the contract metadata into a file using `writeOutput` function, as we need the ABI to invoke deployed smart contract functions.

Web3.js version 4.x was recently released and is not backward compatible with previous 1.x versions. Also, little documentation is available. Thus, we cannot use `web3-lib.ts` from the Remix IDE, which still supports Web3.js 1.x. Therefore, `index.ts` has some notable changes compared to `index-MyToken.ts`. Also, a few other changes need to be made as we are no longer running on Remix. Note that for Project 2 you may either use Web3.js 1.x or 4.x.

Lines 6 imports `compileSols` and `writeOutput` functions from `solc-lib.ts`. Next two lines import Web3 and several data type definitions from `Web3` and `web3-types` libraries.

The `GasHelper` is similar to the function from `web3-lib.ts`. However, it is updated to match new data types in Web 4.x. The `initProvider` function (lines 32-45) is used to initialise Web3 provider where we use a WebSocket to connect to Ganache blockchain node running at `ws://127.0.0.1:7545` (configured in `eth_providers/providers.json` file, and may need to be changed depending on the port your Ganache instance is running).

Given an account name, the `getAccount` function (lines 52-63) retrieves the corresponding private key from `eth_accounts/accounts.json` file. Make sure to update the private keys in `accounts.json` to match your Ganache instance's test accounts. It then generates an Ethereum account address. `getABI` function (lines 71-80) is used to load the contract ABI.

Compared to the Remix example, this program runs in two steps. It first deploys a smart contract and later interacts with an already deployed contract. Lines 105-160 are used to compile the contract, save its metadata to a file, and then deploy the contract. Lines 161-232 are used to transact by following the same steps as the Remix example.

Step 11. Use the following command to compile the TypeScript files:

```
npx tsc
```

This will generate a set of JavaScript files in the `build` folder. If there are any error carefully go through the error messages and fix the problem.

Step 12. Deploy the smart contract to the Ganache blockchain node using the following command (note the three constructor arguments while setting up the token contract):

```
node build/index.js deploy acc0 MyToken "Mint Token" MNT 20000
```

Note down the contract address, as well as transactions and blocks recorded on Ganache. If the command line did not automatically terminate after deploying the contract press `CTRL + C`.

Step 13. Transact with the MyToken contract by issuing the following command (you will need to replace the contract address with the address of the contract you just deployed):

```
node build/index.js transact MyToken 0xB2ac6929d9D07d806bfC34A36A1502eEF8955925
```

If the command did not automatically terminate after execting all the transactions, press `CTRL + C`.

Step 14. Use OpenZeppelin documentation to learn how `allowance`, `approve`, and `transferFrom` functions behave. Extend the code in `src/index.ts` to perform the following transactions:

1. Approve address 2 to spend half of those tokens owned by address 1.
2. Get address 2 to transfer 30% of the approved tokens from address 1 to address 3.

References

[1] Ethereum. *web3.js – Ethereum JavaScript API*. 2021. URL: https://web3js.readthedocs.io/.

[2] Gregory McCubbin. *Intro to Web3.js · Ethereum Blockchain Developer Crash Course*. Aug. 2022. URL: https://www.dappuniversity.com/articles/web3-js-intro.

[3] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum*. O'Reilly Media, 2018. URL: https://github.com/ethereumbook/ethereumbook.

[4] Mozilla Corporation. *async function*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.