Blockchain Developer Notes

Using the Hardhat IDE to Write Solidity Smart Contracts
and
React.js Decentralized Apps

(Version 0.2.2)

by Scott Weberg

Disclaimer:

Your use of these notes constitutes agreement that I am not responsible or in any way liable for any blockchain mis-haps resulting in loss of finances, reputation, or any other thing of value, whether directly or indirectly a result of using these notes.

This information is offered "as is" with no expressed warranty of any kind.

Some of this information was learned from Dapp University courses (https://www.youtube.com/@DappUniversity), with some information from other sources as well.

Find the latest official version of this document on Github:
https://github.com/sw-3/blockchain-dev-docs

Table of Contents

Developer Tools

The following free tools are used for Smart Contract and dApp development.

- **chaijs.com** – the Chai Assertion Library is included in Hardhat projects and used to write automated test scripts
- **ethers.js** – Javascript library which connects an app/script to the blockchain
  - See the github repo of examples for the basic things you would do with ethersjs: https://github.com/dappuniversity/ethers_examples
- **hardhat.org** – Hardhat is a complete Solidity IDE
- **React.js** – Javascript framework for building the UI for a web3 dapp
- **Redux** – a state container for Javascript apps; allows storing/handling larger amounts of blockchain data, analogous to a local pseudo-database
- **remix.ethereum.org** – a browser-based Solidity IDE, also used to “flatten” Solidity contracts
- **docs.openzeppelin.com/contracts** – free library for secure smart contract development. Implements common use cases which can be imported into your solidity projects.

TODO: Add more useful tools here.

Create a New Project

1. make a directory, and cd into it
2. (Optional) For a project with a React.js frontend:
   - `> npx create-react-app app-name` creates a new React app
   - Edit your package.json and add the 3 bold lines below with your info:
     ```
     "version": "0.1.0",
     "description": "One-sentence description of your project.",
     "author": "your name/email/github/etc",
     "license": "MIT",
     ```
3. (Optional) If you want to install any 3rd Party smart contract libraries with **npm**, it may be "safer" to install them at this point *before* creating a Hardhat development environment in the next step. The first time I tried some libraries, I installed them after installing Hardhat, and the React dependencies got out of whack. This may vary depending on what you are trying to install.
4. create a new Hardhat project
   - IF you did **not** do #2 above, (or do not yet have a package.json file in your project directory) then do this "npm init" step:
     `> npm init` (use defaults, and enter your email or github id for author)
     - this creates the initial package.json file
   - `> npm install --save-dev hardhat`
     - adds hardhat dependencies to project
   - `> npx hardhat`
     - enter the defaults; ("Create a JavaScript project")
       creates the contracts/scripts/test folders, hardhat.config.js
   - (If needed) Add any known dependencies to the package.json file, if your project has additional requirements. (See the Appendix for an example package.json file for a professional project, which contains React-bootstrap, Redux and Router.)

     If adding anything here, run:
     - `> npm install`
     - Installs everything in the package.json file for the project
5. Create a Git Repo for the project
   - `> git init`
   - `> git add .`
   - `> git commit -am "Initial commit."`

On Github.com:

- create a new Repo for the project on github
  - do not add a readme, gitignore, or license

From local terminal:

- `> git remote add origin git@github.com:user-name/proj-name.git`
- `> git branch -M main`
- `> git push -u origin main`

6. For contract (.sol) files, 1st 2 lines should be like this. (License can be “MIT” or whatever.)

```
// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;
```

7. For test (.js) files, 1st 2 lines should be like this.

```
const { expect } = require('chai');
const { ethers } = require('hardhat');
```

Manually Create a React App Inside a Hardhat Project

(**NOTE:** `> npx create-react-app appName` creates a brand new React app, but... If you already started coding your Hardhat contracts, use this manual method)

From main project directory:

```
> mkdir src
> touch src/index.js
> touch src/index.css
> touch src/reportWebVitals.js
> mkdir src/components
> touch src/components/App.js
> mkdir public
> touch public/index.html
> touch public/manifest.json
> touch public/robots.txt
```

Copy the contents of reportWebVitals.js, index.js, index.css, index.html, manifest.json, robots.txt into files.

Code the template App.js front end:

```
function App() {
      return(
            <div>Hello, world!</div>
      )
}

export default App;
```

Then from terminal:

```
> npm run start
```

Solidity Example

```
// SPDX-License-Identifier: Unlicense
pragma solidity 0.8.7;

contract MyContract {
    string value;

    constructor() {
        value = "Hello, world!";
    }

    function get() public view returns(string memory) {
        return value;
    }

    function set(string memory _value) public {
        value = _value;
    }
}
```

NOTES: **view** – read-only function
public – function is visible (accessible) from outside the contract
_value – use underscore as naming convention for local scope vars

Writing Tests with Hardhat

The below testing tools are included with the **hardhat-toolbox** installation, when a Hardhat project is created. (See hardhat.org/tutorial/testing-contracts.html for more info.)

- **Mocha** test runner
- Chai Assertion Library (see chaijs.com) – defines **expect**/**assert** constructs

Tests written in javascript, to simulate client-side interactions with smart contracts.

1. Write tests in **test** folder, with same name as the contract in **contracts** folder (ie. Token.sol and Token.js). All hardhat test files should begin with the same 2 lines:

   ```
   const { expect } = require('chai')
   const { ethers } = require('hardhat')
   ```

2. Run tests from the console (this will also compile all your smart contracts):

   ```
   > npx hardhat test
   ```

NOTE: *See the example test code in the Appendix.*

Launch a Local Blockchain & Deploy Contracts

1. Write your Solidity contracts in the **contracts** folder (ie. Token.sol)
2. Edit the **hardhat.config.js** file

   in **module.exports** set solidity version to “0.8.9”

   set networks to have localhost: {}
3. Compile the smart contracts (they should compile automatically when deployed)

```
> npx hardhat compile         // creates files in “artifacts” folder
```

4. Create a **migration script** in the **scripts** folder (named **1_deploy.js**, to indicate order)

   Needs an **async function main()**, then you call the **main()** function with the exit/catch

   The **ethers.getContractFactory** function fetches a contract for deployment

   Then call **await Contract.deploy()**, get the deployed contract and log anything desired
5. Use Hardhat to start a blockchain node (from terminal):

```
> npx hardhat node            // launches blockchain node on localhost
```

6. From a 2nd terminal, run the script to deploy the contracts to the blockchain

```
> npx hardhat run --network localhost ./scripts/1_deploy.js
```

Use Hardhat Console to Interact with Blockchain

1. Enter the console from a terminal

```
> npx hardhat console --network localhost
```

2. Fetch the contract to test into a variable:

```
> const token = await ethers.getContractAt("ContractName", "ContractAddress")
> token.address         // prints out the contract address
```

3. Other things to do in the console:

```
> const accounts = await ethers.getSigners()
> accounts[0].address
> const balance = await ethers.provider.getBalance(accounts[0].address)
> balance.toString()
> ethers.utils.formatEther(balance.toString())
> .exit
```

Launch the React Site for Development

If you are developing a React App as a front-end, launch it locally as follows.

1. Run the blockchain node: `npx hardhat node`
2. Deploy smart contracts: `npx hardhat run –network localhost scripts/1_deploy.js`
3. Start a local web server: `npm run start`
    - right-click select **inspect** and click **Console** tab to see console.log output
4. Make sure MetaMask is connected to your blockchain node (see next section)

Connecting Browser to Blockchain Via MetaMask

First make sure blockchain is running, and you have deployed your contracts to it.

Next add Hardhat network to Metamask:

> RPC URL is in the first line after running `> npx hardhat node`
> Chain ID is “31337”
> ETH
> Block explorer leave blank

Next add Accounts #0, #1, #2 to Metamask, using the private keys listed from launching the node (You can edit account name to Hardhat #0, Hardhat #1, Hardhat #2)

> NOTE: every time start the node, select each Account → Settings → Advanced → Reset Account

Next, convert website into a blockchain website:

> At top of App.js file add the line: `import { ethers } from ‘ethers’`
>
> Inside the function App:

```
const loadBlockchainData = async () => {
	const provider = new ethers.providers.Web3Provider(window.ethereum)
	console.log(provider)
}

useEffect(() => {
	loadBlockchainData()
})
```

Developing a Professional Level App/Front-end

A professional web frontend will consist of several pages and maintain the blockchain state across those pages. **React-Bootstrap** provides basic UI items, **React-Router** provides a mechanism for the frontend to have multiple pages, and **React-Redux** provides more complex state management of the blockchain.

Install the packages

If these were not already included in your packages.json file...

```
> npm install bootstrap react-bootstrap
> npm install react-router-dom react-router-bootstrap
> npm install react-redux
> npm install @reduxjs/toolkit
> npm install redux-thunk
```

Connect Redux to manage state

This section describes how to connect Redux for state management. We'll store the wallet provider, network chainId, and connected account into the state.

Create the initial Rudux application structure

- **src/store/store.js** – small file which configures the Redux store with the reducers reducers – write data into the store (**provider** is the 1st reducer definition)

```
import { configureStore } from '@reduxjs/toolkit'
import provider from './reducers/provider'

export const store = configureStore({
  reducer: {
    provider
  },
  middleware: getDefaultMiddleware =>
    getDefaultMiddleware({
      serializableCheck: false
    })
})
```

- **src/store/reducers/** - folder to contain all the reducer and initial state definitions
  - Will contain multiple definitions … provider.js is likely common to most apps.
- **src/store/reducers/provider.js** – defines the initial state of provider data, and associated reducer functions
  - Uses the **createSlice** thing from reduxjs/toolkit (See code on next page)

```
import { createSlice } from '@reduxjs/toolkit'

export const provider = createSlice({
  name: 'provider',
  initialState: {
    connection: null,
    chainId: null,
    account: null
  },
  reducers: {
    setProvider: (state, action) => {
      state.connection = action.payload
    },
    setNetwork: (state, action) => {
      state.chainId = action.payload
    },
    setAccount: (state, action) => {
      state.account = action.payload
    }
  }
})

export const {
  setProvider,
  setNetwork,
  setAccount
} = provider.actions

export default provider.reducer;
```

- **src/store/interactions.js** – collects all the functions which interact with the Redux store.
  - These functions will dispatch the actions defined and handled in the reducers.
  - These functions are called by frontend components as a clean way for the frontend to interact with the blockchain.

```
import { ethers } from 'ethers'

import {
  setProvider,
  setNetwork,
  setAccount
 } from './reducers/provider'

export const loadProvider = (dispatch) => {
  const provider = new ethers.providers.Web3Provider(window.ethereum)
  dispatch(setProvider(provider))

  return provider
}

export const loadNetwork = async (provider, dispatch) => {
  const { chainId } = await provider.getNetwork()
  dispatch(setNetwork(chainId))

  return chainId
}

export const loadAccount = async (dispatch) => {
  const accounts = await window.ethereum.request({ method: 'eth_requestAccounts' })
  const account = ethers.utils.getAddress(accounts[0])
```

```
  dispatch(setAccount(account))

  return account
}
```

- **src/components/App.js** – the main page component of the frontend
  - import the interactions; call them to connect the App the blockchain via the Redux store:

```
import { useDispatch } from 'react-redux'
import { ethers } from 'ethers'

import {
  loadProvider,
  loadNetwork,
  loadAccount
} from '../store/interactions'

function App() {

  const dispatch = useDispatch()

  const loadBlockchainData = async () => {
    // initiate provider
    const provider = await loadProvider(dispatch)

    // fetch chainId
    const chainId = await loadNetwork(provider, dispatch)

    // reload page when network changes
    window.ethereum.on('chainChanged', () => {
      window.location.reload()
    })

    // fetch current account from Metamask when changed
    window.ethereum.on('accountsChanged', async () => {
      await loadAccount(dispatch)
    })
  }

  useEffect(() => {
    loadBlockchainData()
  }, []);

  return ( … page rendering code here …
```

- **src/index.js** – this is the actual front page of the app .. it needs to render App.js.
  - Render App.js within the **React-Redux Provider**
    (**NOTE:** this is not the blockchain ‘provider’ – they are 2 different things.)

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'
import 'bootstrap/dist/css/bootstrap.css'
import App from './components/App'
import reportWebVitals from './reportWebVitals'

import { Provider } from 'react-redux'
import { store } from './store/store'

const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);

reportWebVitals();
```

When the above is all working correctly, the Redux DevTools icon (browser addon) will light up, and you will be able to see the provider in there. Under the provider, you will see ‘connection’, ‘chainId’ and ‘account’ all defined. (Note: get the Redux DevTools addon for Chrome, to easily view the contents in the Redux state.)

Using React-Router for Multi-Page Front End

The <HashRouter> tag allows addressing different components to appear on the page as a “hash URL” rather than an actual URL.

The URL for the front page would look like “localhost:3000”, and a hash url would look like “localhost:3000/#/view-diplomas” when displaying the ViewDiplomas component.

```
import { HashRouter, Routes, Route } from 'react-router-dom'
import { Container } from 'react-bootstrap'
…
…
function App() {
  …
  …
  return (
    <Container>
      <HashRouter>

        <Navigation />

        <hr />

        <Tabs />

        <Routes>
          <Route exact path="/" element={<ViewTranscripts />} />
          <Route path="/view-diplomas" element={<ViewDiplomas />} />
          <Route path="/issue-transcripts" element={<IssueTranscripts />} />
          <Route path="/issue-diplomas" element={<IssueDiplomas />} />
        </Routes>

      </HashRouter>
    </Container>
  )
}
```

Deploying to a Test Network

Configure for Deployment on Polygon's Mumbai Test Net (TODO – finish this section)

1. Need an account on alchemy.com, create an App there
2. Chain = Polygon, Network = Polygon Mumbai
3. in MetaMask, add Network Name = Polygon Mumbai, chain ID 80001, Symbol MATIC, copy http URL from "view key" on alchemy into Metamask URL
4. mumbaifaucet.com: Fund MetaMask accounts with MATIC on Polygon Mumbai network
5. .env file: add ALCHEMY_API_KEY (TODO: need an example, and explain .env usage with .gitignore)
6. hardhatconfig.js: add the mumbai info (TODO: need an example of this config!)
7. DO NOT RUN THE DEPLOY SCRIPT YET! Proceed to next section...

Flatten Your Contract Source Code

NOTE: FLATTEN BEFORE DEPLOYING TO the Blockchain!!! Remix may require a fix!

First, make sure your Solidity contracts are not including libraries that are not needed for production use.

> (An example of this would be to remove the import of "hardhat/console.sol" in your contracts, if you used console output during development/testing, but do not need it for production.)

Use remix.ethereum.org to "flatten" the contract.

- Open a new file under "Contracts". Name it the same as your .sol file.
- Copy-Paste your .sol file into this new file on Remix.
- Right-click on the file name in Remix, and select "Flatten".
    - This will create a new version of the file with "flattened" in the name.
    - *You will use this new version to verify your contract source code after deployment.*

Deploy Contracts

After you complete the previous 2 sections, you are ready to actually deploy your contracts.

1. Deploy contracts to Mumbai:
   `npx hardhat run --network mumbai scripts/1_deploy.js`
2. Validate txn and address on mumbai.polygonscan.com
3. Add contract addresses (from step 2) to your src/config.json for network “80001” if needed for your React application logic. (TODO: need example of this somewhere)

Verify Contract Source Code (on testnet or mainnet)

TODO: Need to finish this section better.

- Click in the flattened file code in Remix, and press Ctrl-A to select all, then copy-paste this file into the code field on Etherscan/Polygonscan when verifying your contract there.
- Make sure you select the right Solidity version that was used to compile the contract, select “single file”, and the desired license.

(TODO) Website Deployment Using Fleek

Even with your contracts deployed to a testnet or mainnet, you can run your React front-end locally to access them. However, if you wish to allow others to access your dApp, then you can use Fleek to deploy your front-end App to the internet for free.

TODO: Need to write this section.

Using IPFS for NFT images & metadata

TODO: This section could be more detailed...

- Create account on pinata.cloud
- upload your images folder
- copy the location hash for your folder
- paste that hash onto ipfs.io site, to see them on IPFS
- Edit the metadata for an image to use the IPFS folder hash, with .../1.jpg for image etc.
- then upload the metadata folder to IPFS after updated with the image locations

Using Timestamps

When working with date/time stamps, I found some useful tools.

- The site epochconverter.com will show the Unix **epoch time** for any date/time. You need to use the epoch time for date/time variables in javascript.
    - Using this site, you can see the exact epoch time for any date in the future. So if you want something to happen (like an NFT mint) at an exact day/time in the future, you can program it by getting the epoch time value from this site.
- Date.now() grabs the current time in milliseconds. (Divide by 1000 to get seconds.)
- You can add/subtract seconds from current time to get a future/past time etc.
    - ie. there are 3600 seconds per hour, so adding 3600 seconds to current time, would represent the time at 1 hour in the future. Subtracting 3600 seconds would represent the time at 1 hour in the past.
    - 86,400 seconds equals 1 day; 604,800 seconds equals 1 week
- Use the **day.js** library to format dates for display more easily.

Appendix

Example package.json for Professional Project

```
{
  "name": "project-name",
  "version": "0.1.0",
  "description": "Project using React.js and Hardhat. Includes Bootstrap, Redux, Router.",
  "author": "Your name/email/github/etc",
  "license": "MIT",
  "private": true,
  "dependencies": {
    "@openzeppelin/contracts": "^4.8.3",
    "@reduxjs/toolkit": "^1.9.5",
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "bootstrap": "^5.2.3",
    "react": "^18.2.0",
    "react-bootstrap": "^2.7.4",
    "react-dom": "^18.2.0",
    "react-redux": "^8.0.5",
    "react-router-bootstrap": "^0.26.2",
    "react-router-dom": "^6.11.1",
    "react-scripts": "5.0.1",
    "redux-thunk": "^2.4.2",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "@nomicfoundation/hardhat-toolbox": "^2.0.2",
    "hardhat": "^2.14.0"
  }
}
```

Example Test Code

This code was written as part of the automated tests for a Solidity ERC20 token. It is a sample only, and does not represent a complete set of tests.

```
const { expect } = require('chai');
const { ethers } = require('hardhat');

const tokens = (n) => {
    return ethers.parseUnits(n.toString(), 'ether')
}

describe('Token', ()=> {
    let token, accounts, deployer, receiver

    beforeEach(async () => {
        // fetch Token from blockchain
        const Token = await ethers.getContractFactory('Token')
        token = await Token.deploy('ScottW3 Token', 'SW3T', '100000000')

        // fetch accounts
        accounts = await ethers.getSigners()
        deployer = accounts[0]
        receiver = accounts[1]
    })

    describe('Deployment', () => {
        const name = 'ScottW3 Token'
        const symbol = 'SW3T'

        it('has correct name', async ()=> {
            expect(await token.name()).to.equal(name)
        })

        it('has correct symbol', async ()=> {
            expect(await token.symbol()).to.equal(symbol)
        })
    })

    describe('Sending Tokens', () => {
        let amount, transaction, filter, events

        describe('Success', () => {

            beforeEach(async () => {
                amount = tokens(100)
                transaction = await token.connect(deployer).transfer(receiver.address, amount)
                await transaction.wait()
            })

            it('transfers token balances', async () => {
                // ensure that tokens were transfered (balance changed)
                expect(await token.balanceOf(deployer.address)).to.equal(tokens(99999900))
                expect(await token.balanceOf(receiver.address)).to.equal(amount)
            })
```

```
            it('emits Transfer event', async () => {
                filter = token.filters.Transfer
                events = await token.queryFilter(filter, -1)
                const event = events[0]

                expect(event.fragment.name).to.equal('Transfer')

                const args = event.args
                expect(args.from).to.equal(deployer.address)
                expect(args.to).to.equal(receiver.address)
                expect(args.value).to.equal(amount)
            })
        })

        describe('Failure', () => {
            it('rejects insufficient balances', async () => {
                const invalidAmount = tokens(1000000000)        // 1 billion
                await expect(token.connect(deployer).transfer(receiver.address,
invalidAmount)).to.be.reverted
            })
        })
    })
})
```

TODO: Add more example code/files here