Installing the system

The system can be found on Github at `https://github.com/Litolo/VC-DLT-Honours`. Clone the repo and use at your leisure.

Requirements & Dependencies

1. This system requires Node.js and node package manager (npm) to be installed. Instructions for installation can be found here for both - `https://docs.npmjs.com/downloading-and-installing-node-js-and-npm`

2. This system has been developed and tested with the latest stable version of Node and npm which was version 20.12.2 and 10.5.0 respectively. Information on node version support is available here - `https://nodejs.org/en/about/previous-releases`.

3. To run the demonstration of the project, users must have python and the package manager pip installed - found here `https://www.python.org/downloads/`. Users must also be able to run the demonstration jupyter notebook file by installing Jupyter - found here `https://jupyter.org/`.

4. Install node dependencies with command:

```
$ npm install
```

5. OPTIONAL - configure your wallet and API keys to allow the use of the interplanetary file system (IPFS) and external networks. Instructions on how to do so are included in the *API Keys* section.

API Keys - OPTIONAL

This section is optional. While technically not necessary for core functionality, it is needed if you wish to connected to a different external network or upload credential documents to the IPFS.

If you want to be able to upload your credential documents to the IPFS or use other networks apart from localhost you will need to configure you API keys. Keys are stored in an environment file called .env in the root directory of the project.

If you wish to access these features so, create a .env file in the root directory of the project containing the keys for API access. This project uses Pinata[1] for IPFS access and Infura[2] for the Sepolia endpoint. You may wish to use other providers but you will need to modify your network endpoint url attribute in the config file, called hardhat.config.ts located in the root directory of the project, shown below.

[1] `https://www.pinata.cloud/`
[2] `https://www.infura.io/`

```
import "@nomicfoundation/hardhat-toolbox";

require('hardhat-abi-exporter');
require('dotenv').config({ path: './.env' })

module.exports = {
  solidity: {
    version:  "0.8.20",
    settings: {
      optimizer: {
        enabled: true,
        runs: 2000,
      }
    }
  },
  networks: {
    sepolia: {
      url: `https://sepolia.infura.io/v3/${process.env.INFURA_API_KEY}`,
      accounts: [process.env.SEPOLIA_PRIVATE_KEY]
    }
  },
  abiExporter: {
    path: './abi',
    runOnCompile: true,
    clear: true,
    flat: false,
    only: [],
    spacing: 2,
    format: "json",
  }
};
```

You will also have to change urls in IPFS upload and fetch scripts appropriately for your provider(s), located at scripts/generics/upload.ts and scripts/generics/upload.ts in the project structure. You should refer to the official documentation from your provider on how to handle API responses and format requests. Here is the documentation for pinata, the provider used for the current implementation `https://docs.pinata.cloud/api-reference/introduction`

You may find it useful to have a digital wallet to manage your private keys for different networks, as this project does for the Sepolia network. Some common wallet providers are Coinbase[3] or Metamask[4]. You can find you Sepolia account private key from your wallet provider. Shown below is the .env format used, in case you want to configure your project to work with no additional edits.

[3] `https://www.coinbase.com/wallet`
[4] `https://metamask.io/`

```
GATEWAY_URL=[your dedicated IPFS gateway URL]
GATEWAY_TOKEN=[your dedicated IPFS gateway token]
PINATA_API_KEY=[your Pinata API key]
INFURA_API_KEY=[your Infura API key]
SEPOLIA_PRIVATE_KEY=[your Sepolia private key from your wallet]
```

Space & Memory Requirements

The only space requirement for the system is to have enough storage to hold the source code files of the project, about 330 MB. If you wish to run the demonstration, you should ensure there is enough space of your system to store the json files it creates (a few KBs at most per run).

There are no memory requirements for the system. Systems with lower memory may find that computation takes longer.

Compiling & running the system

1. The following command compiles the smart contracts:

   ```
   $ npx hardhat compile
   ```

2. Using the compiled contracts, you can deploy to a network of your choosing with command:

   ```
   $ npx hardhat run scripts/generics/deploy.ts --network <network>
   ```

3. On the deployed smart contracts, you can now call a variety of methods. Functions for interaction using each method are already created in the scripts/generics/ folder. If you struggle to understand how to use these scripts, refer to the demonstration shown in the demo.ipynb file in the root of the directory and variety of supplied demo scripts found in the scripts/demo/ folder. Other methods on the issuance smart contract exist in accordance with ERC721, which you can view the source code from at node_modules/@openzeppelin/contracts/token/ERC721/ERC721.sol or the API at `https://docs.openzeppelin.com/contracts/2.x/api/token/erc721`.

4. To run the demo, simply run the demo.ipynb jupyter notebook file and run each cell in order. Visit `https://docs.jupyter.org/en/latest/running.html` for help on how to run a jupyter notebook file.

5. To run tests, run the following command in the console:

   ```
   $ npx hardhat test
   ```

Key file paths

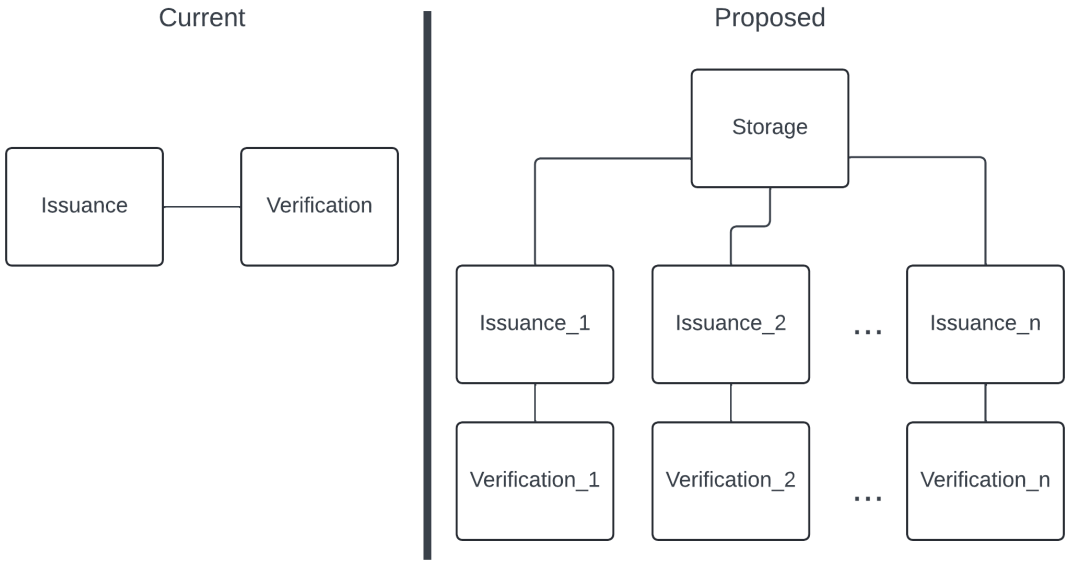
1. Smart contract source code : contracts/
2. Scripts used for common interaction with the system : scripts/generics/
3. Scripts used in the demonstration, use cases for common interaction : scripts/demo/
4. Automated tests : test/

Future improvements

The system is built to be easily modifiable and adaptable to a variety of use cases. Some future improvements which will benefit the system are listed below.

Split NFT storage and issuance behaviour

To enforce modularity, the issuance contract should be split into two. One for NFT storage and the other with only the methods for token issuance. Currently the issuance contract is not modifiable, since it also stores tokens, which if a new version of the contract was deployed would require to re-mint every token (or link to the old contract). This also means that multiple different issuance contracts can use the same storage contract. A quick sketch is shown below to illustrate this design.



Multiple issuers access to the same contract instance

Mainly to split the costs of deployment, it would be useful to implement a mechanism to allow multiple issuing organisations to access the same deployed smart contract. This is can be done in a few ways. One way can be to keep a mapping of allowed addresses stored within the contract. Only an allowed address can add another address. This opens up issues relating to malicious issuers inviting others who should not have access, which could be mitigated by requiring majority approval of allowed issuers to provide elevated access. If choosing to implement this it is recommended you read about proper smart contract access control patterns. OpenZeppelin has a good solution using roles `https://docs.openzeppelin.com/contracts/4.x/access-control`.

Batching credentials

To decrease cost, it would benefit if the system was capable of batching credential issuances/verifications into one smart contract call, instead of multiple. It is unclear from a theoretical perspective if this would actually be computational faster, so implementers should evaluate the performance by using/adapting existing evaluation script in the scripts/demo/evaluation.ts file.

Support for other blockchain networks

Since the current implementation is written in Solidity, it is only runnable on those networks which virtual machines support solidity code. Smart contract source code can be written in different languages to support the running on different virtual machines.

Bug solving

When modifying or adding to the system, remember to write tests into the test/ folder. Test the system when modifying it any way.

When debugging, if possible for the language you are using, use a suitable debugger with breakpoints. This will allow you to view the state of the program at a specific point in it's execution. Remix IDE provides many other useful features for debugging smart contracts which you can read about at `https://remix-ide.readthedocs.io/en/latest/debugger.html`.

For smart contracts, with Hardhat you can create a local node to query the state of transactions and calls to contracts when connected through it. To spin up a local node run the following command in the terminal:

```
$ npx hardhat node
```

You can then connect through this node using the localhost option for the network flag in your commands. An example is shown below. Doing this also allows you to view stack traces when a transaction/call fails.

```
$ npx hardhat run scripts/generics/deploy.ts --network localhost
```