h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbeit
FACULTY OF
ELECTRICAL ENGINEERING AND
INFORMATION TECHNOLOGY

M03 Team Project

Blockchain Based Order and Access Management

SS21/22

submitted by

Sachith Harshitha Liyanagama
Matr. Nr. 769617
Ken Freise
Matr. Nr. 746041
Shakil Ahammed
Matr. Nr. 769709

submitted to

Prof. Dr.-Ing. Markus Haid

Table of Contents

1. Project Scope

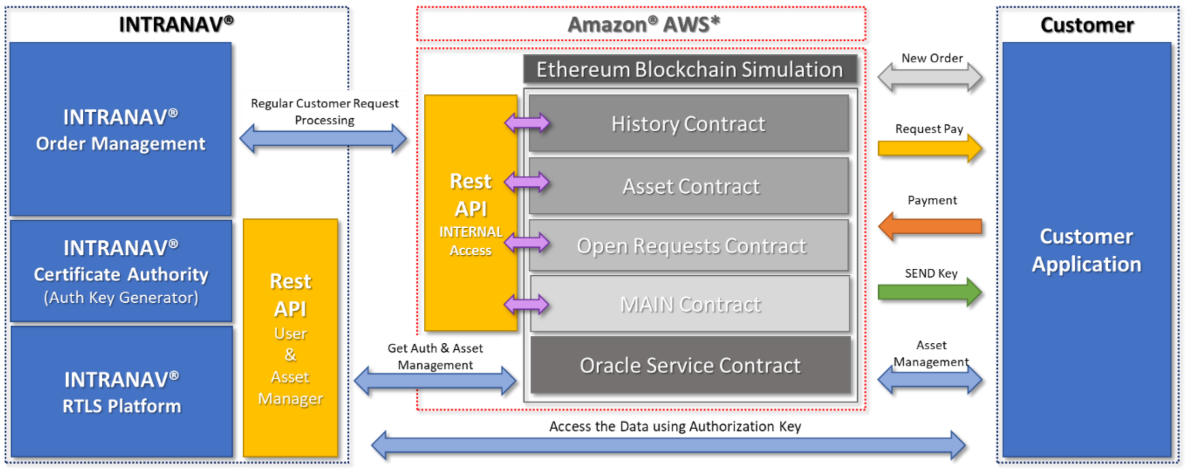

INTRANAV®
INTRANAV® Order Management
INTRANAV® Certificate Authority (Auth Key Generator)
INTRANAV® RTLS Platform
Rest API User & Asset Manager
Regular Customer Request Processing
Amazon® AWS*
Ethereum Blockchain Simulation
Rest API INTERNAL Access
History Contract
Asset Contract
Open Requests Contract
MAIN Contract
Oracle Service Contract
Get Auth & Asset Management
Access the Data using Authorization Key
New Order
Request Pay
Payment
SEND Key
Asset Management
Customer
Customer Application

Figure 1 : Project deliverable structure

The project scope was to develop client order management system integration concept using Ethereum Blockchain.

1.1. Requirements

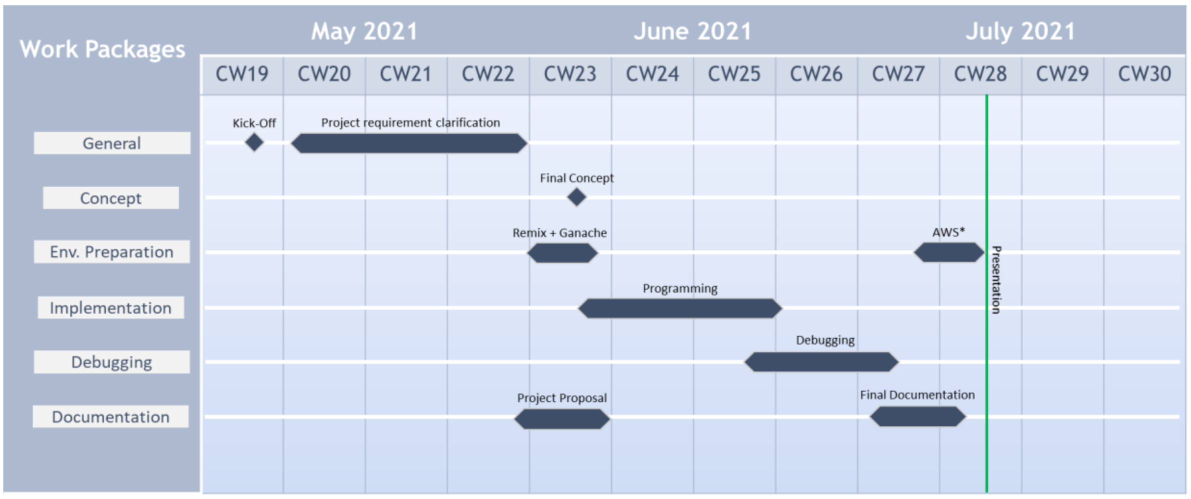
According to the project requirements, it was defined that, Ethereum based smart contracts to be used and based on the concept components other components such as web applications, Rest APIs to be developed as required in order to showcase the full application functionality.

1.2. Deliverables

Index	Component	Description	Deliverability
01	Ganache **Blockchain Simulator**	Implementing and documentation of the dockerized version of the Ganache CLI.	✓
02	Smart Contract(s) **Main, History, Open Requests, Assets & Oracle Service**	Implementation of the Smart contracts with required structure of the artifact and documentation.	✓
05	Rest API **For Internal Communication**	Swagger driven Java based Rest API (Optimized to work with the Main Smart Contract)	✓
06	Rest API **For External Communication**	Swagger driven Java based Rest API (Optimized to work with the Main Smart Contract)	✓

07	Rest API **Mock API to represent the INRANAV REST API**	A simple [Swagger]() driven Java based Rest API with a built-in AES Token simulator and asset manager.	✓
08	Amazon AWS Deployment	Deploy the above containerized applications in Amazon AWS separately and create the communication in between them.	?
09	Sample Customer Application	Simple Graphical User Interface Application to showcase the functionality.	✓
?	All the components which are meant to host on AWS are created in docker containers. If required, using a PAID AWS account these can be hosted in AWS.		

1.3. Timeline



2. Concept Overview

2.1. General Concept Overview

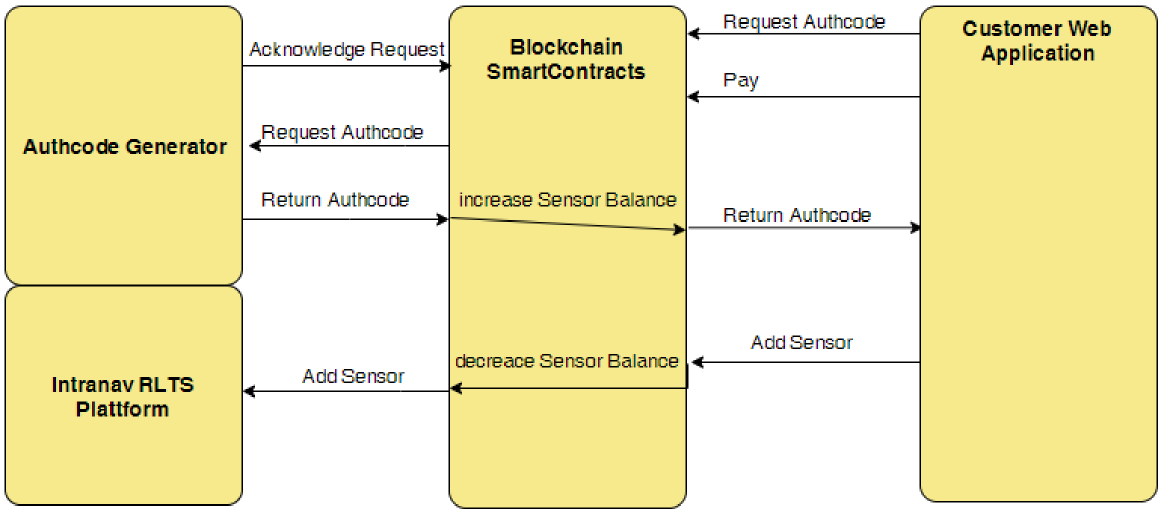


Figure 2 : General Concept Overview

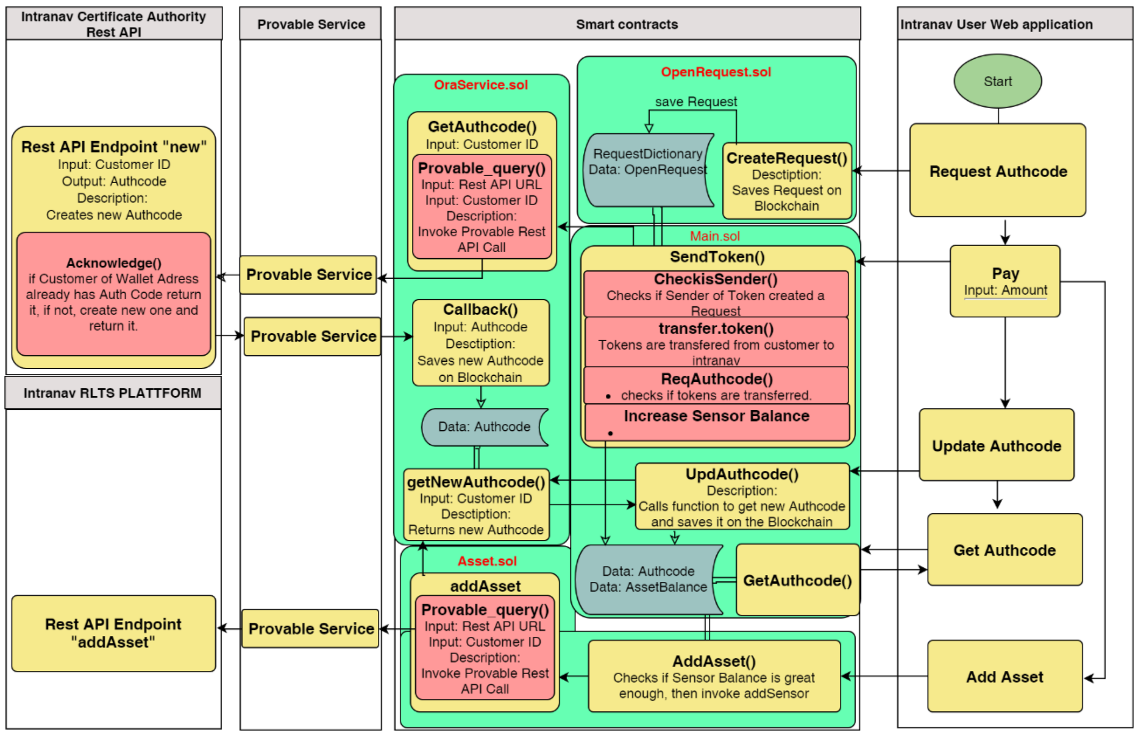


Figure 3 : Detailed Concept Overview

3. Blockchain

3.1. What is Blockchain?

Blockchain is a shared, immutable ledger that facilitates the process of recording transactions and tracking assets in a business network. An asset can be tangible (a house, car, cash, land) or intangible (intellectual property, patents, copyrights, branding). Virtually anything of value can be tracked and traded on a blockchain network, reducing risk and cutting costs for all involved.

3.2. What is Ethereum?

Ethereum is a technology that lets you send cryptocurrency to anyone for a small fee. It also powers applications that everyone can use and no one can take down. It's called the world's programmable blockchain. Ethereum builds on Bitcoin's innovation, with some big differences. Both let you use digital money without payment providers or banks. But Ethereum is programmable, so you can also use it for lots of different digital assets – even Bitcoin!
This also means Ethereum is for more than payments. It's a marketplace of financial services, games and apps that can't steal your data or censor you.

3.3. Smart Contract

3.3.1. What is it?

Smart contracts are simply programs stored on a blockchain that run when predetermined conditions are met. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss. They can also automate a workflow, triggering the next action when conditions are met.

3.3.2. How does it work

Smart contracts work by following simple "if/when…then…" statements that are written into code on a blockchain. A network of computers executes the actions when predetermined conditions have been met and verified. These actions could include releasing funds to the appropriate parties, registering a vehicle, sending notifications, or issuing a ticket. The blockchain is then updated when the transaction is completed. That means the transaction cannot be changed, and only parties who have been granted permission can see the results.

Within a smart contract, there can be as many stipulations as needed to satisfy the participants that the task will be completed satisfactorily. To establish the terms, participants must determine how transactions and their data are represented on the

blockchain, agree on the "if/when...then…" rules that govern those transactions, explore all possible exceptions, and define a framework for resolving disputes.
Then the smart contract can be programmed by a developer – although increasingly, organizations that use blockchain for business provide templates, web interfaces, and other online tools to simplify structuring smart contracts.

3.3.3. Types of Smart Contracts

Assuming the reader has a basic understanding of contracts and computer programming, and building on from our definition of smart contracts, we can roughly classify smart contracts and protocols into the following major categories.

- Smart legal contracts

These are presumably the most obvious kind. Most, if not, all contracts are legally enforceable. Without going into much technicalities, a smart legal contact is one that involves strict legal recourses in case parties involved in the same were to not fulfill their end of the bargain. As previously mentioned, the current legal framework in different countries and contexts lack sufficient support for smart and automated contracts on the blockchain and their legal status is unclear. However, once the laws are made, smart contracts can be made to simplify processes which currently involve strict regulatory oversight such as transactions in the financial and real estate market, government subsidies, international trade etc.

- DAOs

Decentralized Autonomous Organizations, shortly DAO, can be loosely defined as communities that exist on the blockchain. The community may be defined by a set of rules arrived at and put into code via smart contracts. Every action by every participant would then be subject to these sets of rules with the task of enforcing and reaching at recourse in case of a break being left to the program. Multitudes of smart contracts make up these rules and they work in tandem policing and watching over participants.
A DAO called the Genesis DAO was created by Ethereum participants in may of 2016. The community was meant to be a crowdfunding and venture capital platform. In a surprisingly short period of time they managed to raise an astounding $150 million. However, hacker(s) found loopholes in the system and managed to steal about $50 million dollars' worth of Ethers from the crowdfund investors. The hack and its fallout resulted in a fork of the Ethereum blockchain into two, Ethereum and Ethereum Classic.

- Application logic contracts (ALCs)

If you've heard about the internet of things in conjunction with the blockchain, chances are that the matter talked about Application logic contacts, shortly ALC. Such smart contracts contain application specific code that work in conjunction with other smart contracts and programs on the blockchain. They aid in communicating with and validating communication between devices (while in the domain of IoT). ALCs are a pivotal piece of every multi-function smart contract and mostly always work under a managing program. They find applications everywhere in most examples cited here.

3.4. Blockchain Oracle

3.4.1. The Oracle Problem

The Oracle Problem is defined as the security, authenticity, and trust conflict between third-party oracles and the trustless execution of smart contracts. The digital world needs to know about the physical world.
Oracles retain an enormous amount of power over smart contracts in how they are executed because the data they provide determines how the smart contracts execute. Therefore, data feeds from third-party sources give that data substantial influence over the execution of a smart contract, removing its trustless nature as part of a decentralized network.

Each node needs to calculate(Mine) same result for same input for a contract method. Because of this you cant use an external api from inside a smart contract to get external data. Because for example with an API to get the time, each node would get another time. To get external data inside a smart contract a blockchain Oracle is used.

3.4.2. How does it work?

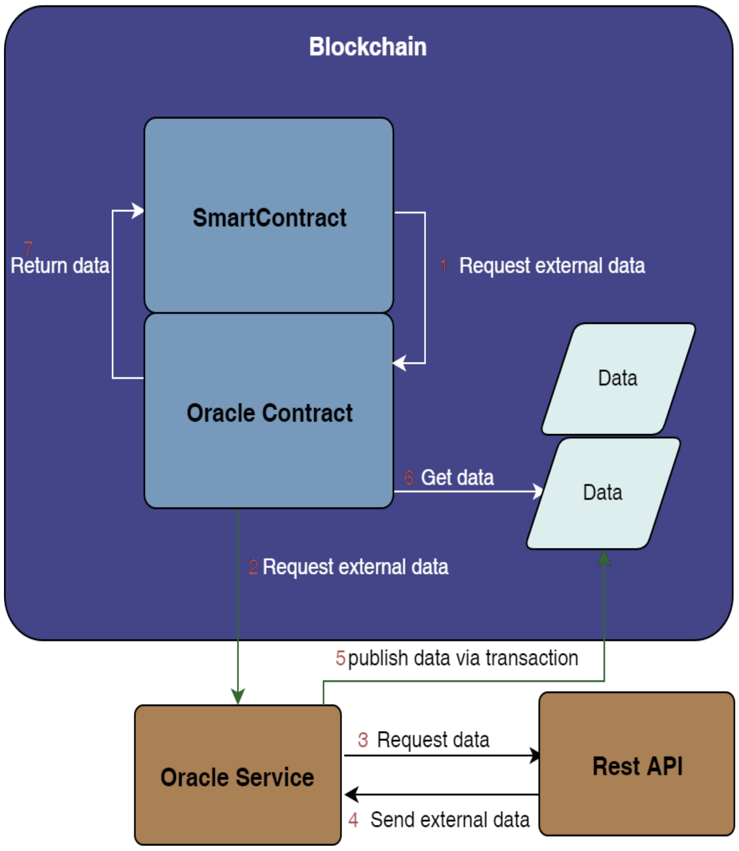


Figure 4 : Blockchain Oracle Communication

1. Inside Smart Contract call oracle function to get external data.
2. The Oracle contract requests the data from Oracle Service
3. The Oracle Service invoke a Rest API
4. The Oracle Service gets the data
5. The Oracle Service publishs the data on the blockchain
6. Oracle Contract callback function gets the data
7. The Oracle Contract returns the data
8. All nodes can mine the same result, since the data is published

3.4.3. Oracles used in Ethereum

There are multiple Oracles exist and it is also possible to create your own modular oracle networks to get any customized data into the blockchain. In example Chainlink and Provable (Formerly known as Oraclize) are two most prominent ones. In our applicaton we used Provable, since the Provable Oracle is created to use external REST API.

3.5. How to Communicate with blockchain

Generally, the Blockchain communication happens via different ways and each of them are unique to the blockchain architecture. Ethereum supports JSON RPC API communication and there are available API libraries developed for this task as well.

- JSON RPC API

Ethereum JSON-RPC APIs use a namespace system. RPC methods are clustered into several categories, depending on their usage. All method names are composed of the namespace, an underscore, and the actual method name within the namespace.

For example, the eth_call method resides in the eth namespace.
It will request data either from the node, execute an EVM function and return a response, or transmit data to the Ethereum network.

- Different Languages and available API libraries for Ethereum Communication

Language	API Library
.Net Framework (C#, F#)	Nethereum
Java, JavaScript	web3J, web3.js & Ethers.js
Golang	Geth (Go-Ethereum)
Others (If no API library exists)	JSON-RPC

Table 1 : Different Programming Languages and the available API libraries

4. Components

4.1. Ganache CLI

Ganache is a personal blockchain for rapid Ethereum and Corda distributed application development. You can use Ganache across the entire development cycle; enabling you to develop, deploy, and test your dApps in a safe and deterministic environment.
The Ganache CLI is the Command line version of Ganache. We will be using a modified version of the Ganache CLI docker with a built in Ethereum Bridge in order to be able to use external oracle services.

4.2. Smart Contracts

We have four Smart Contracts: Main.sol, Assets.sol, OpenRequests.sol and OraService.sol. They have different functionalities. The overview call structure is shown below.

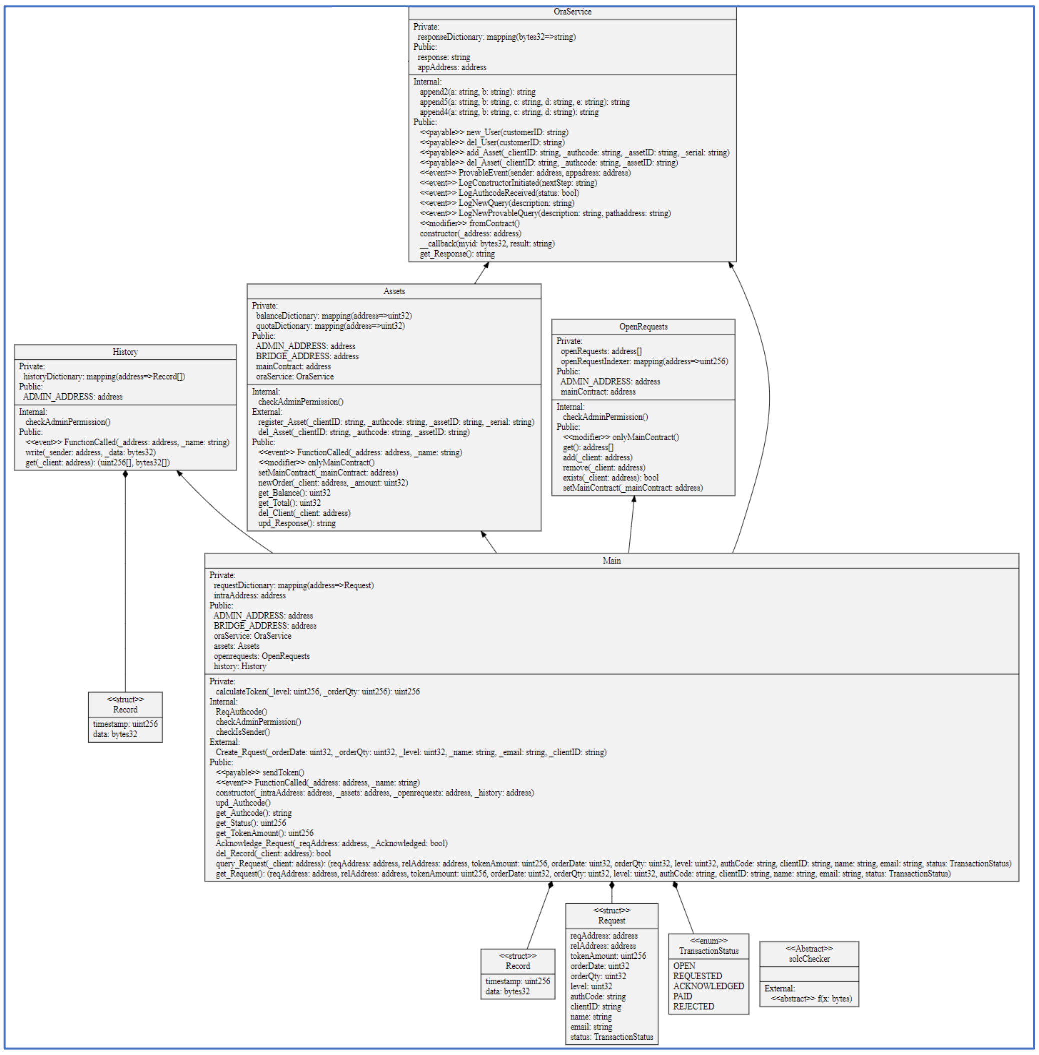

OraService
Private:
responseDictionary: mapping(bytes32=>string)
Public:
response: string
appAddress: address
Internal:
append2(a: string, b: string): string
append5(a: string, b: string, c: string, d: string, e: string): string
append4(a: string, b: string, c: string, d: string): string
Public:
<<payable>> new_User(customerID: string)
<<payable>> del_User(customerID: string)
<<payable>> add_Asset(_clientID: string, _authcode: string, _assetID: string, _serial: string)
<<payable>> del_Asset(_clientID: string, _authcode: string, _assetID: string)
<<event>> ProvableEvent(sender: address, appadress: address)
<<event>> LogConstructorInitiated(nextStep: string)
<<event>> LogAuthcodeReceived(status: bool)
<<event>> LogNewQuery(description: string)
<<event>> LogNewProvableQuery(description: string, pathaddress: string)
<<modifier>> fromContract()
constructor(_address: address)
__callback(myid: bytes32, result: string)
get_Response(): string
History
Private:
historyDictionary: mapping(address=>Record[])
Public:
ADMIN_ADDRESS: address
Internal:
checkAdminPermission()
Public:
<<event>> FunctionCalled(_address: address, _name: string)
write(_sender: address, _data: bytes32)
get(_client: address): (uint256[], bytes32[])
Assets
Private:
balanceDictionary: mapping(address=>uint32)
quotaDictionary: mapping(address=>uint32)
Public:
ADMIN_ADDRESS: address
BRIDGE_ADDRESS: address
mainContract: address
oraService: OraService
Internal:
checkAdminPermission()
External:
register_Asset(_clientID: string, _authcode: string, _assetID: string, _serial: string)
del_Asset(_clientID: string, _authcode: string, _assetID: string)
Public:
<<event>> FunctionCalled(_address: address, _name: string)
<<modifier>> onlyMainContract()
setMainContract(_mainContract: address)
newOrder(_client: address, _amount: uint32)
get_Balance(): uint32
get_Total(): uint32
del_Client(_client: address)
upd_Response(): string
OpenRequests
Private:
openRequests: address[]
openRequestIndexer: mapping(address=>uint256)
Public:
ADMIN_ADDRESS: address
mainContract: address
Internal:
checkAdminPermission()
Public:
<<modifier>> onlyMainContract()
get(): address[]
add(_client: address)
remove(_client: address)
exists(_client: address): bool
setMainContract(_mainContract: address)
Main
Private:
requestDictionary: mapping(address=>Request)
intraAddress: address
Public:
ADMIN_ADDRESS: address
BRIDGE_ADDRESS: address
oraService: OraService
assets: Assets
openrequests: OpenRequests
history: History
Private:
calculateToken(_level: uint256, _orderQty: uint256): uint256
Internal:
ReqAuthcode()
checkAdminPermission()
checkIsSender()
External:
Create_Rquest(_orderDate: uint32, _orderQty: uint32, _level: uint32, _name: string, _email: string, _clientID: string)
Public:
<<payable>> sendToken()
<<event>> FunctionCalled(_address: address, _name: string)
constructor(_intraAddress: address, _assets: address, _openrequests: address, _history: address)
upd_Authcode()
get_Authcode(): string
get_Status(): uint256
get_TokenAmount(): uint256
Acknowledge_Request(_reqAddress: address, _Acknowledged: bool)
del_Record(_client: address): bool
query_Request(_client: address): (reqAddress: address, relAddress: address, tokenAmount: uint256, orderDate: uint32, orderQty: uint32, level: uint32, authCode: string, clientID: string, name: string, email: string, status: TransactionStatus)
get_Request(): (reqAddress: address, relAddress: address, tokenAmount: uint256, orderDate: uint32, orderQty: uint32, level: uint32, authCode: string, clientID: string, name: string, email: string, status: TransactionStatus)
<<struct>>
Record
timestamp: uint256
data: bytes32
<<struct>>
Record
timestamp: uint256
data: bytes32
<<struct>>
Request
reqAddress: address
relAddress: address
tokenAmount: uint256
orderDate: uint32
orderQty: uint32
level: uint32
authCode: string
clientID: string
name: string
email: string
status: TransactionStatus
<<enum>>
TransactionStatus
OPEN
REQUESTED
ACKNOWLEDGED
PAID
REJECTED
<<Abstract>>
solcChecker
External:
<<abstract>> f(x: bytes)

Figure 5 : UML Diagram of the Smart Contract Communication

4.2.1. Main Contract

Functionality of Main Contract:

- Save the Request a customer created to get the authentication code
- Calculate the price to process request
- Pay the required tokens for the request
- Invoke function of the OraService smart contract to generate and get new authentication code
- Update the new Authcode on Blockchain
- Return the new Authcode to Web Application
- increase Sensor balance if payment was successful
- decrease Sensor balance if Sensor is added

4.2.2. Assets Contract

Functionality of Asset Contract:

- Adds or deletes an asset
- Update the asset quota and balance with the order amount

4.2.3. History Contract

- Records history of the order process

4.2.4. OpenRequests Contract

- Provides a list where open requests can be saved to or deleted from

4.2.5. Oracle Service Contract

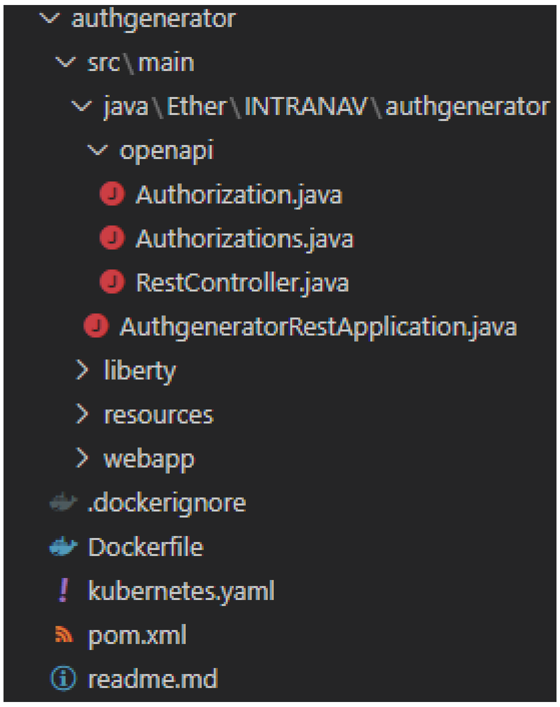
Functionality of Oracle Service Contrace:

- Use Provable Oracle Service to get the Authcode from Rest API
- Save the Authcode on the Blockchain
- Return the new Authcode to Main.sol Contract

4.3. Authentication Code Generator API

The Authentication Code Generator Rest API provides two endpoints. The first one: “/new/{authToken}/{reqID}". This endpoint gets an authToken and reqID as input and is returning the authentication code for the requester id. If the request id is not existing, a new authentication code is created and returned. The second endpoint is: “delete/{authToken}/{reqID}". There the authentication code of the request id is deleted.

4.3.1. Project Structure



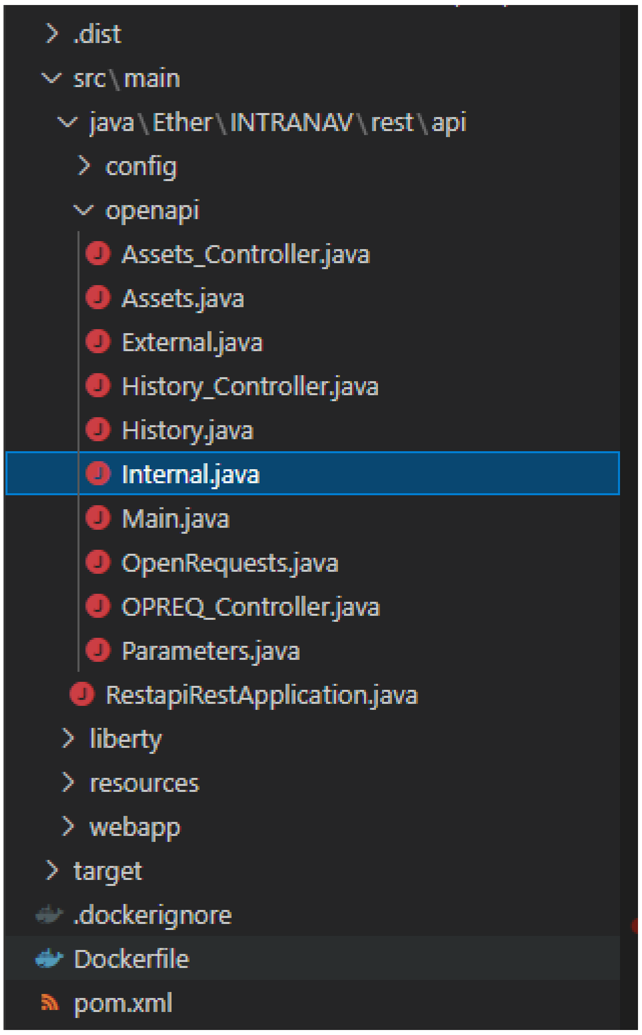
```
authgenerator
  src\main
    java\Ether\INTRANAV\authgenerator
      openapi
        Authorization.java
        Authorizations.java
        RestController.java
      AuthgeneratorRestApplication.java
    liberty
    resources
    webapp
  .dockerignore
  Dockerfile
  kubernetes.yaml
  pom.xml
  readme.md
```

The most important file is RestController.java. This File contains the implementation of the 2 Endpoints the Rest API provides.

4.4. Internal Communication Rest API

We used the Openliberty Framework to create this REST API. The source is in 04_InternalRestAPI. To communicate with the blockchain web3j library is used. This API acknowledges the Open Requests.

4.4.1. Project structure



```
> .dist
v src\main
  v java\Ether\INTRANAV\rest\api
    > config
    v openapi
        Assets_Controller.java
        Assets.java
        External.java
        History_Controller.java
        History.java
        Internal.java
        Main.java
        OpenRequests.java
        OPREQ_Controller.java
        Parameters.java
      RestapiRestApplication.java
  > liberty
  > resources
  > webapp
> target
  .dockerignore
  Dockerfile
  pom.xml
```

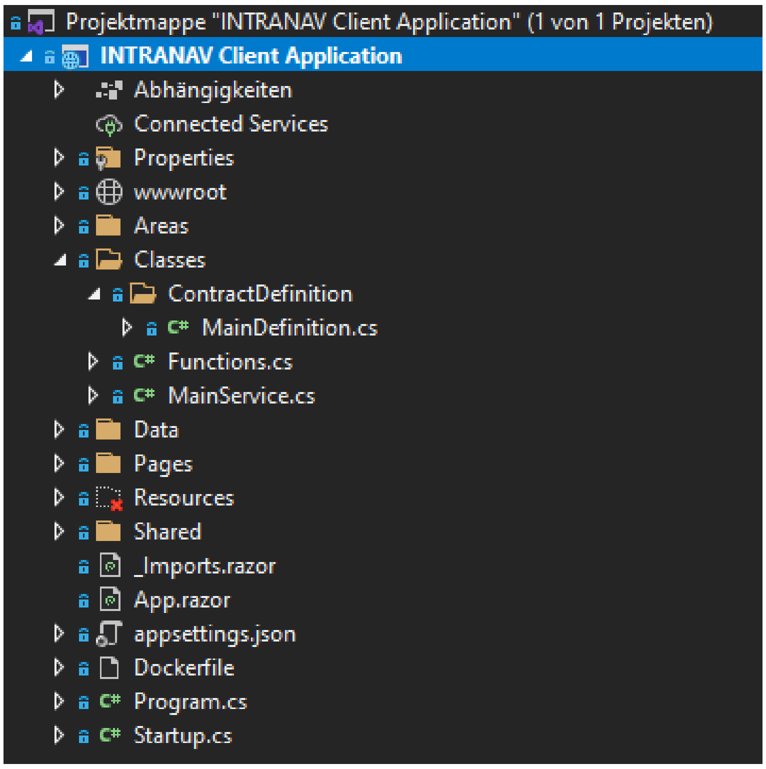
In internal.java the REST API endpoints are defined.

4.5. Customer Portal Application

The Customer Application provides an Interface for the Customer to:

- create and send an Authentication Code request
- pay the required tokens for the request
- get the Authentication code and the Sensor Balance
- add and delete a Sensor.

4.5.1. Project Structure



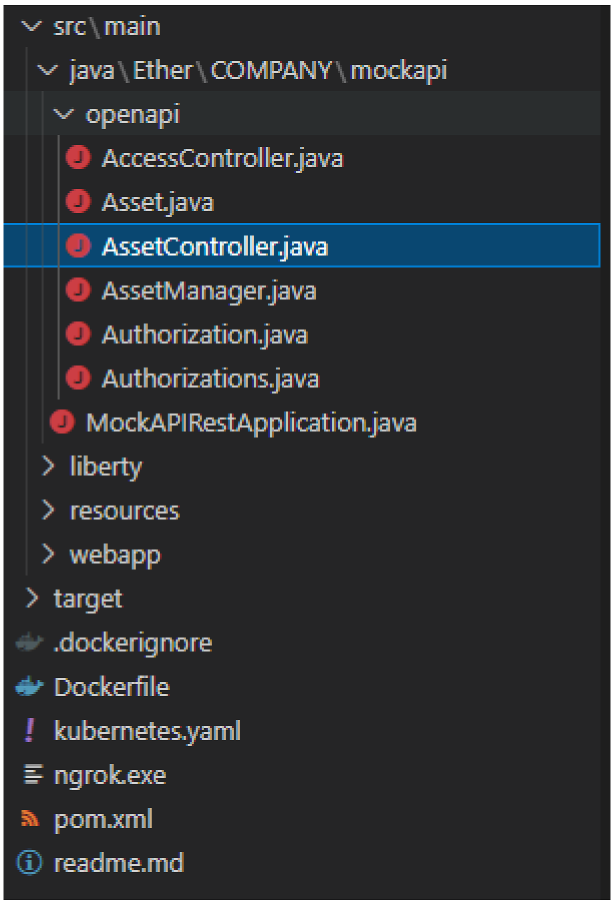
```
Projektmappe "INTRANAV Client Application" (1 von 1 Projekten)
  INTRANAV Client Application
    Abhängigkeiten
    Connected Services
    Properties
    wwwroot
    Areas
    Classes
      ContractDefinition
        MainDefinition.cs
      Functions.cs
      MainService.cs
    Data
    Pages
    Resources
    Shared
    _Imports.razor
    App.razor
    appsettings.json
    Dockerfile
    Program.cs
    Startup.cs
```

In "Functions.cs" the main functions like newRequest() and transfertoken() are implemented. The main functionality of the User Application can be modified here. These functions invoke nethereum functions which are implemented in "MainService.cs". Netherum is the interface between the Web Application and Ethereum Blockchain/Smart Contracts. In MainDefinitions.cs the names of the SmartContract Functions are defined which nethereum is calling.

4.6. INTRANAV RTLS Rest API

We emulated the Intranav RTLS API because the existing one needs too much informations to register an asset, this would exceed the size of the smart contract. Our simulated REST API is shown here.

4.6.1. Project Structure



```
src\main
  java\Ether\COMPANY\mockapi
    openapi
      AccessController.java
      Asset.java
      AssetController.java
      AssetManager.java
      Authorization.java
      Authorizations.java
    MockAPIRestApplication.java
  liberty
  resources
  webapp
target
.dockerignore
Dockerfile
kubernetes.yaml
ngrok.exe
pom.xml
readme.md
```

The REST API Endpoints to register an Asset are defined in AssetControler.java

5. How to Run

5.1. Pre-Requisites

5.1.1. Docker

In order to proceed with setup of the project it is required to configure docker on your computer.

Please refer: https://docs.docker.com/get-docker/

5.1.2. Ethereum Remix

Given its ability to debug and test the smart contracts as well as extensive support with the built-in solidity compilers, Ethereum Remix is the suitable Solidity programming IDE for this project.

Please refer: https://remix-project.org/

There are other IDEs available for Solidity Smart Contract development. Most used ones are Atom, Truffle and VS code with solidity extension. A simple comparison can be seen below.

Remix IDE	VS Code with Plugins
All-in-One solution	Require multiple plugin installations and configurations
Can import any source libraries locally or over the network.	Does not support Direct import of supporting contracts over external sources. (Ex. Github)
Automation is not currently possible.	Some testing can be automated.
Good for small-mid ranged projects.	Enterprise grade big solidity projects with multiple connections would benefit.
Do not support code formatting.	There are plugins available to support code formatting.
Ethereum Specific development with multiple developers backing the process.	Single developers developing the plugins separately and therefore takes time for bug fixing.

Table 2 : Remix IDE vs VS-Code for Solidity Programming

5.1.3. NGROK

In order to for the blockchain to access external data using the oracle service, the external data source must be exist in the internet. Therefore we will be using the NGROK local tunneling platform to expose the Company Rest API system to the internet so that we could obtain data from it.

Please refer: https://dashboard.ngrok.com/get-started/setup

5.1.4. Visual Studio Code (DEV only)

Since the REST API microservices are programmed in JAVA language, Visual Studio Code can be the go to method for programming the APIs. Similarly with the plugins available for open liberty the API can be debugged live without the need of restart.

Please refer: https://code.visualstudio.com/
In order to proceed you will also require to install below plugins

a. Java Extension Pack
b. Open Liberty Tools

5.1.5. JAVA

Java JDK or JDE is necessary to run .java files.

Please refer: https://www.java.com/de/download/manual.jsp

5.1.6. MAVEN

Maven is a build automation tool used primarily for Java projects. Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages.
Please refer: https://maven.apache.org/install.html

5.1.7. GIT

Since the source codes are all stored in GIT repositories it would be convenient to have git installed on the local computer.

Please refer: https://git-scm.com/

5.1.8. Web3j (DEV only)

Web3j is a lightweight, highly modular, reactive, type safe Java and Android library for working with Smart Contracts and integrating with Ethereum blockchains. This allows you to work with Ethereum blockchains, without the additional overhead of having to write your own integration code for the platform.

Please refer: http://docs.web3j.io/latest/quickstart/

5.1.9. Nethereum Generator (DEV only)

Nethereum is the .Net integration library for Ethereum, simplifying smart contract management and interaction with Ethereum nodes whether they are public, like Geth , Parity or private, like Quorum and Besu.

Please refer: http://docs.nethereum.com/en/latest/getting-started/

In order to generate the API libraries, we will be using the Console CLI version of the Nethereum.
Please refer : https://docs.nethereum.com/en/latest/nethereum-codegen-console/

5.2. Clone the GIT project repository

Step 1: Clone the project repo to your computer.

```
git clone
https://github.com/SachiHarshitha/HDa_TeamProject_SS21.git
```

5.3. Start Ganache CLI Docker

Step 1: Clone the Original Ganache CLI docker source into the path ".\01_GanacheCLI". Should look like this : ".\01_GanacheCLI\ganach-cli"

```
git clone https://github.com/trufflesuite/ganache-cli.git
```

Step 2: Clone the Ganache CLI, Ethereum Bridge integration project into the same path. Should look like this : ".\01_GanacheCLI\GanacheCLI-with-EthBridge"

```
git  clone  https://github.com/SachiHarshitha/GanacheCLI-with-
EthBridge.git
```

Step 3: Copy "1entrypoint.sh", "args.js" and "Dockerfile" from ".\01_GanacheCLI\GanacheCLI-with-EthBridge\" folder and replace into ".\01_GanacheCLI\ganache-cli\" .

Info: In args.js are wallet addresses defined. The first wallet address is (0xb8B7…). In the smart contracts we use this address as Admin_Adress. You need this information later when you deploy the smart contracts. **Some of the next steps are only necessary, if you change the args.js file.**

Step 4: Build the docker container.

```
cd .\ganache-cli\
docker build --tag trufflesuite/ganache-cli-test .
```

Step 5: Run the docker container.

```
docker run -d --name ganache -p 8545:8545 trufflesuite/ganache-
cli-test  -l  80000000  -m  "argue  liberty  sock  desert  drift
peasant vivid fox hint document author circle" -v
```

** If you have a specific mnemonic code for your wallet replace the string value above with your own mnemonic string.

```
!!! Only important when you use an other wallet address
(different args.js used), if not ignore this step!!!
```

Step 6: Open the docker CLI for the ganache docker and copy out the Ethereum Bridge Address. You will need it to deploy the smart contracts.

How to : Open Docker Software, and double click on the running ganache container.

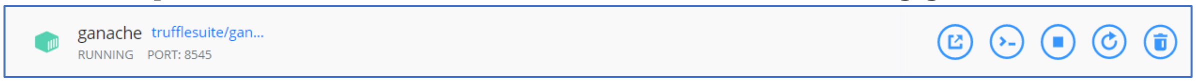
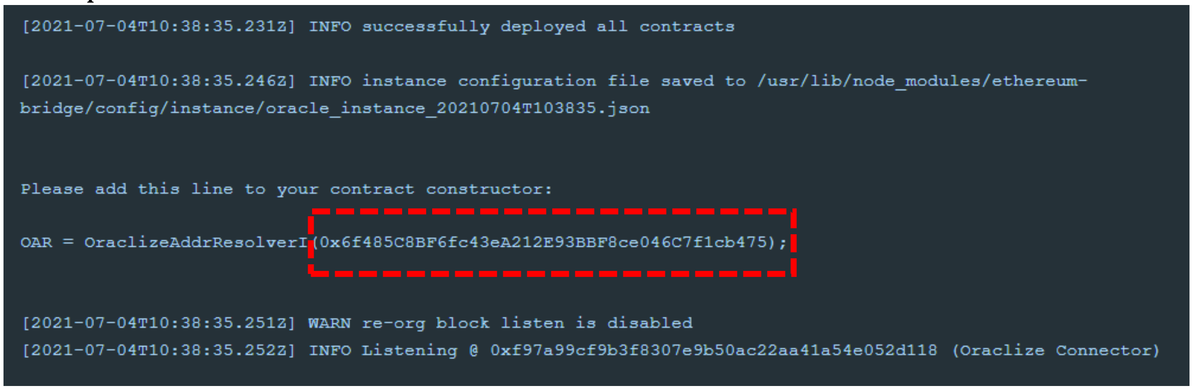


Figure 6 : Ganache Docker Container in Docker GUI.

Example :



```
[2021-07-04T10:38:35.231Z] INFO successfully deployed all contracts

[2021-07-04T10:38:35.246Z] INFO instance configuration file saved to /usr/lib/node_modules/ethereum-
bridge/config/instance/oracle_instance_20210704T103835.json


Please add this line to your contract constructor:

OAR = OraclizeAddrResolverI(0x6f485C8BF6fc43eA212E93BBF8ce046C7f1cb475);


[2021-07-04T10:38:35.251Z] WARN re-org block listen is disabled
[2021-07-04T10:38:35.252Z] INFO Listening @ 0xf97a99cf9b3f8307e9b50ac22aa41a54e052d118 (Oraclize Connector)
```

Figure 7 : Docker CLI log from Ganache Docker with Ethereum Bridge Address

5.4. Deploy Smart Contracts

Step 1: Open Remix IDE and set workspace to “localhost”

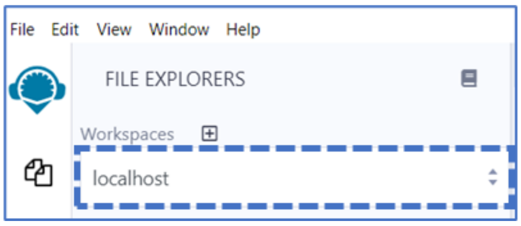


Figure 8 : Set Workspace

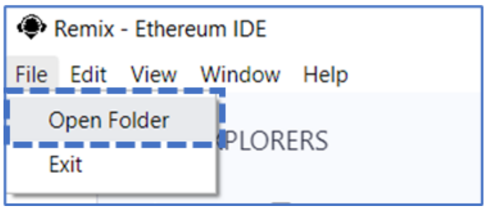


Figure 9 : Open Folder view

Step 2: Open the "\02_SmartContract" folder from File Menu.

You will be able to see five *.sol files inside the folder as shown.

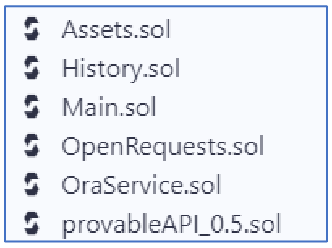
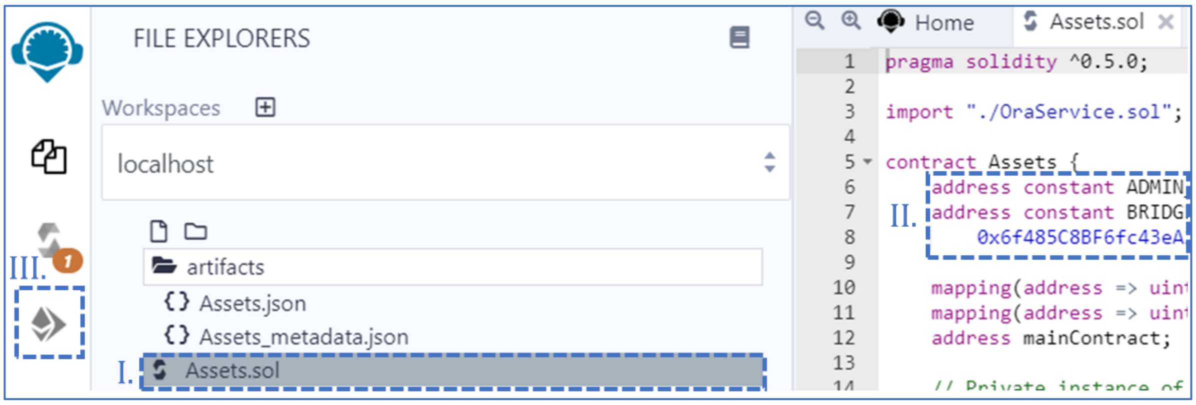


Figure 10 : Smart Contracts

Step 3: Open "Assets.sol" smart contract



```
pragma solidity ^0.5.0;

import "./OraService.sol";

contract Assets {
    address constant ADMIN
    address constant BRIDG
        0x6f485C8BF6fc43eA

    mapping(address => uin
    mapping(address => uin
    address mainContract;

    // Private instance of
```

Figure 11 : Assets Smart Contract Deployment – Step 1

!!! Only important when you use other wallet address (different args.js used), if not ignore this step!!!

Step 4:

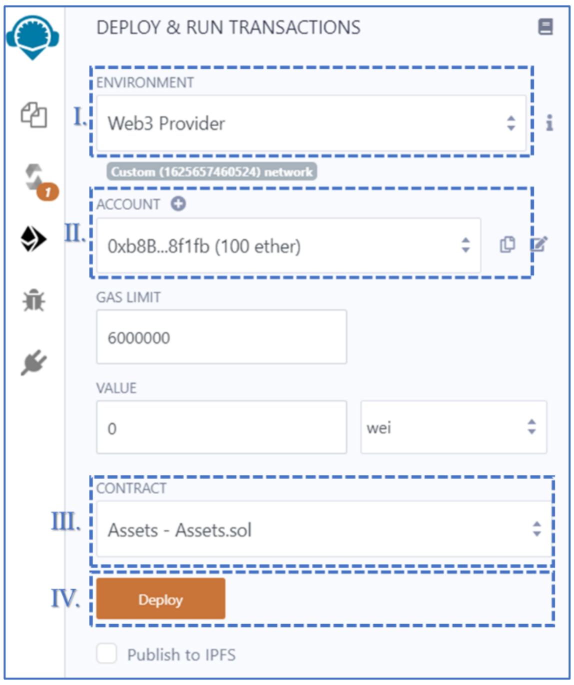
please update the below information according to your instance as shown in "Item II".

```
address constant ADMIN_ADDRESS =
0x79bD771Dc8CaBdaFf066aFfeB86455f9d5602E65;
//ADMIN Account which the Internal API uses

address constant BRIDGE_ADDRESS =
0x6f485C8BF6fc43eA212E93BBF8ce046C7f1cb475;
// Ethereum-Bridge Account address
```

Step 5:

Now go to the compile tab and compile the contract, then deploy the contract using the "Deploy & Run Transactions" tab in REMIX IDE.



ENVIRONMENT: Web3 Provider

Custom (1625657460524) network

ACCOUNT: 0xb8B...8f1fb (100 ether)

GAS LIMIT: 6000000

VALUE: 0 wei

CONTRACT: Assets - Assets.sol

Figure 12 : Deploy Contract

I. Set Environment

Select the "**Web3 Provider**" as the environment.

In the coming up "**External Node Connection** "window, click "**OK** ".

II. Set Account

Select the first account in the blockchain. (it's the (0xbb87) account, which is defined as admin address in the smart contracts.

III. Check the contract

Make sure the "**Assets.sol**" contract is selected.

IV. Deploy the contract

Click "Deploy" to deploy the contract into the blockchain.

Step 4: Repeat the above "Step 3" for "History.sol" and "OpenRequests.sol" smart contracts.

Step 5: Deploy the Main Contract.

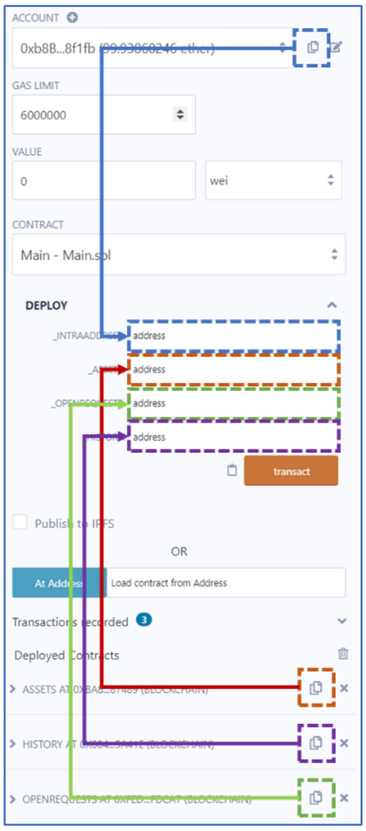


Figure 13 : Main Contract Deployment

Since the Main Contract is larger in size, please Enable "Contract Optimization" ✓ Enable optimization in Compile Tab.

Repeat the above "Step 3" according to the address values in your instance.

When you are required to deploy the contract, it will require certain parameters to be passed into the constructor as shown below. Copy the address values from below "Deployed Contracts" section and provide it into the appropriate address parameter.

Please note that use the First Account in the "Accounts" drop down list.

AVOID using the second account since **it** is fixed to **the** Ethereum Bridge.

Step 6: Set the Main Contract address in "Assets.sol" and "OpenRequests.sol" contract instances.

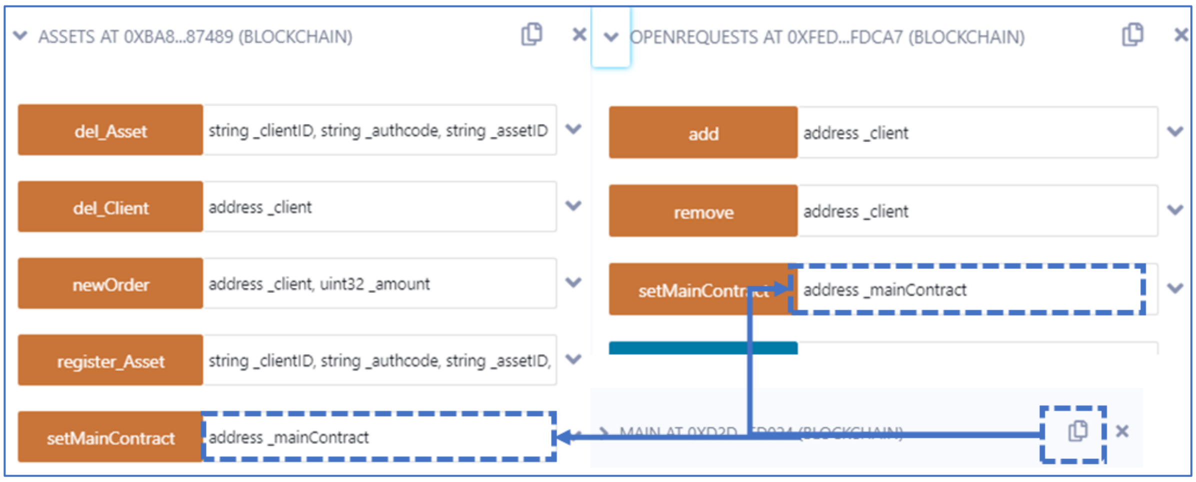

ASSETS AT 0XBA8...87489 (BLOCKCHAIN)
del_Asset
string _clientID, string _authcode, string _assetID
del_Client
address _client
newOrder
address _client, uint32 _amount
register_Asset
string _clientID, string _authcode, string _assetID,
setMainContract
address _mainContract
OPENREQUESTS AT 0XFED...FDCA7 (BLOCKCHAIN)
add
address _client
remove
address _client
setMainContract
address _mainContract
MAIN AT 0XD2D...FD024 (BLOCKCHAIN)

Figure 14 : Set Main Contract address into Assets and OpenRequests.

5.5. Start the Internal Communication Rest API

Step 1: Redirect to "\04_InternalRestAPI" path and edit the contract addresses.
THIS STEP SHOULD NOT BE NECCECCARY, BUT CHECK IF THE ADRESSES ARE CORRECT.

```
The “Parameters.java” JAVA file can be found here under the
path,
“.\04_InternalRestAPI\src\main\java\Ether\INTRANAV\rest\api\o
penapi”.
```

Open the file in your text editor and replace the below information.

```
/**
 * Set the Parameters below according to the Simulation Envir
onment
 */
public class Parameters {

// Main Smart Contract Address
public static final String MAIN_CONTRACT_ADDRESS = "0xd2DF0bc
9B2e9eb28c260daDf7D2ACC6db23fD024";
// ASSETS Smart Contract Address
public static final String ASSETS_CONTRACT_ADDRESS = "0xbA89f
50904FeB82245C9c238dDb8A88420b87489";
// Open Requests Smart Contract Address
public static final String OPREQUESTS_CONTRACT_ADDRESS = "0xf
EdC7803a40c39ae4bfef1430677B5f3B6cFDca7";
// History Smart Contract Address
public static final String HISTORY_CONTRACT_ADDRESS = "0x6048
65472362ba89C115dB5a7e01A31e8E75a41E";
// Private KEY of the Client Account used in the project
public static final String CLIENT_PRIVATE_KEY = "0xbf562bcbb6
792187dc5cdcf645ff07595d68ce9f234f173785ca1eae097c61e1";
// Private KEY of the ADMIN Account used in the project
public static final String ADMIN_PRIVATE_KEY = "0x77f4d804740
5cc8dbeed5c831a6a8123c3102948b235b5aca7a94be529cd743e";
// Address of the Ganache Docker. In order to access another
docker value must be "http://host.docker.internal:PORT/"
public static final String GANACHE = "http://host.docker.inte
rnal:8545/";
}
```

Step 2 : Compile the maven project.

```
cd .\ 04_InternalRestAPI \
mvn install -f ".\pom.xml"
```

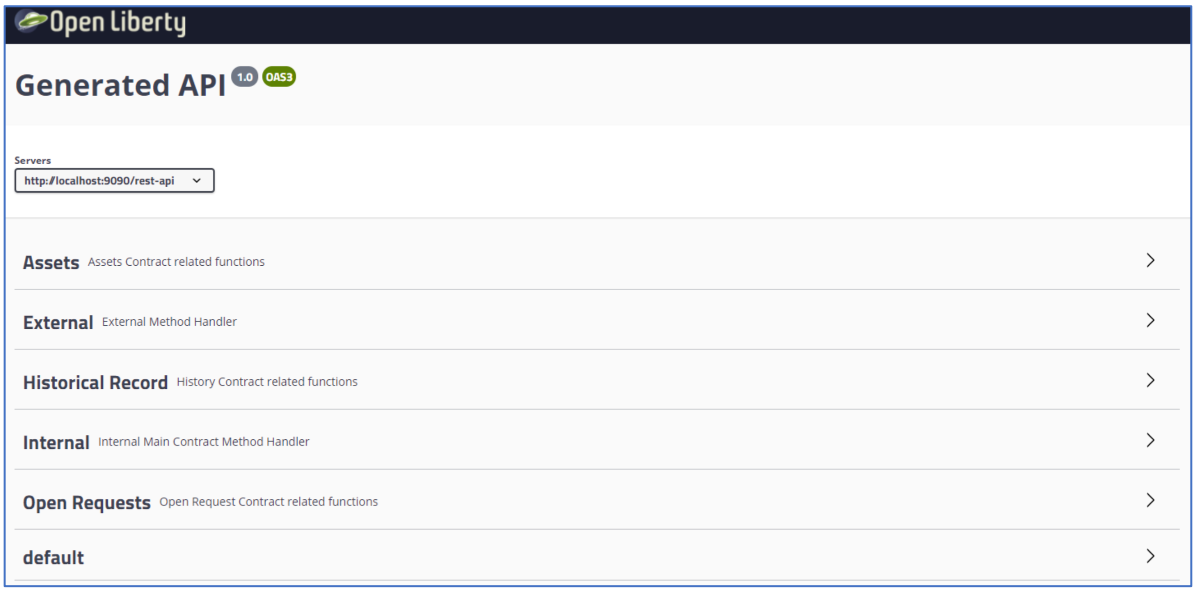
Step 3: Build the docker container

```
docker build -t rest-api:1.0-SNAPSHOT .
```

Step 4: Start the Docker Container

```
docker run -d --name InternalAPI -p 9090:9090 rest-api:1.0-
SNAPSHOT
```

Now if you open the path http://localhost:9090/openapi/ui/ you will be able to see the Internal Communication API is running.



Generated API 1.0 OAS3

Servers
http://localhost:9090/rest-api

Assets Assets Contract related functions

External External Method Handler

Historical Record History Contract related functions

Internal Internal Main Contract Method Handler

Open Requests Open Request Contract related functions

default

Figure 15 : Internal Communication Rest API

5.6. Start the Company Rest API

Step 1: Redirect to "\03_CompanyRestAPI" path and compile the maven project.

```
cd .\03_CompanyRestAPI\
mvn install -f ".\pom.xml"
```

Step 2: Build the docker container

```
docker build -t company-mock-api:1.0-SNAPSHOT .
```

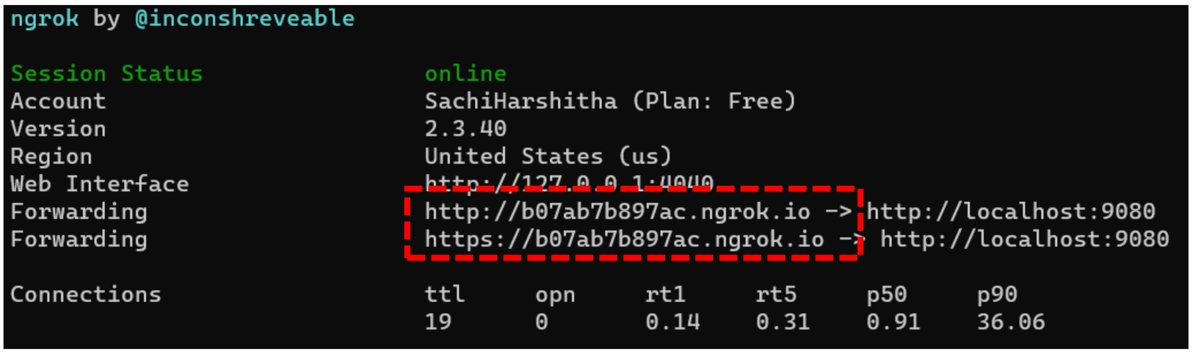
Step 3: Start the Docker Container

```
docker run -d --name system -p 9080:9080 company-mock-api:1.0-
SNAPSHOT
```

Step 4: Expose the port 9080 and generate an external web address for the Rest API.

```
.\ngrok http 9080
```

Step 5: Check if above steps succeed by opening the link generated in the ngrok console window.
Example :



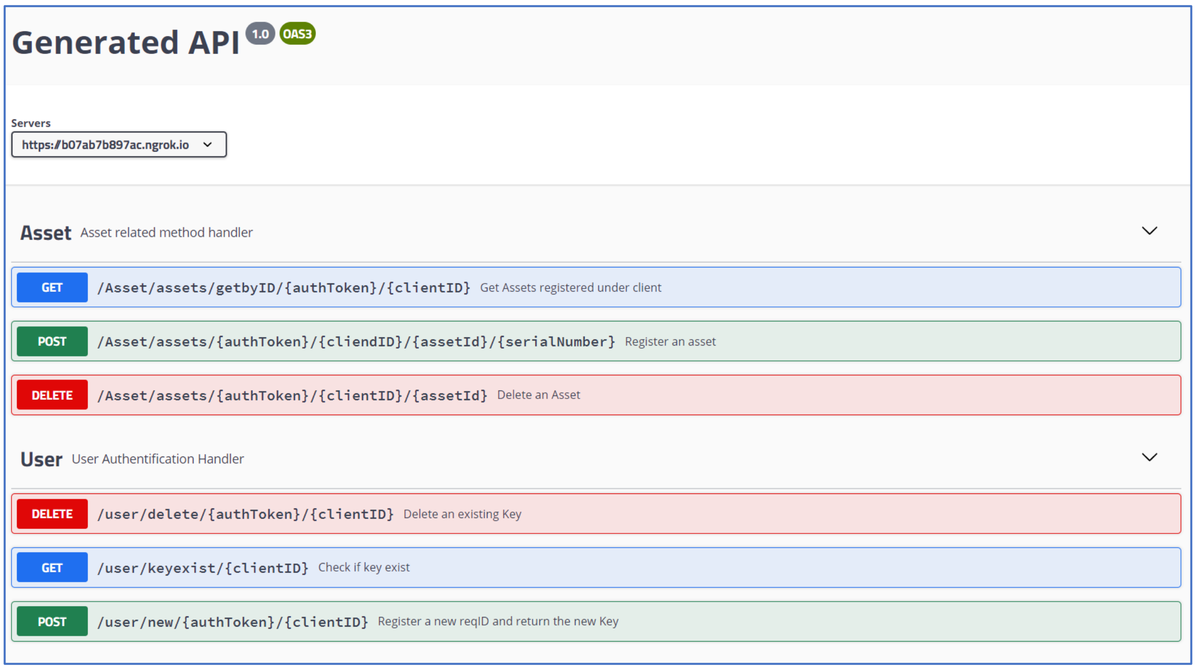
```
ngrok by @inconshreveable

Session Status                online
Account                       SachiHarshitha (Plan: Free)
Version                       2.3.40
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://b07ab7b897ac.ngrok.io -> http://localhost:9080
Forwarding                    https://b07ab7b897ac.ngrok.io -> http://localhost:9080

Connections                   ttl     opn     rt1     rt5     p50     p90
                              19      0       0.14    0.31    0.91    36.06
```

Figure 16 : NGROK local tunneling CLI window.

The address mentioned above MUST be append with "/openapi/ui"
Ex. https://b07ab7b897ac.ngrok.io/openapi/ui/



Generated API 1.0 OAS3

Servers
https://b07ab7b897ac.ngrok.io

Asset Asset related method handler

GET `/Asset/assets/getbyID/{authToken}/{clientID}` Get Assets registered under client

POST `/Asset/assets/{authToken}/{cliendID}/{assetId}/{serialNumber}` Register an asset

DELETE `/Asset/assets/{authToken}/{clientID}/{assetId}` Delete an Asset

User User Authentification Handler

DELETE `/user/delete/{authToken}/{clientID}` Delete an existing Key

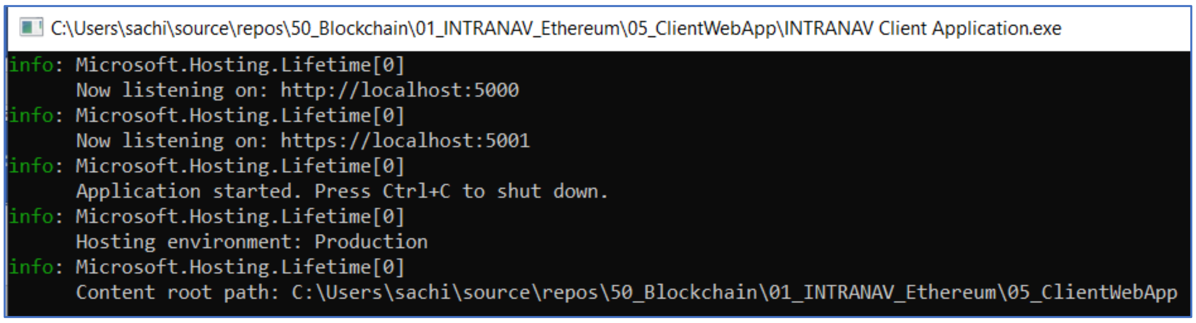
GET `/user/keyexist/{clientID}` Check if key exist

POST `/user/new/{authToken}/{clientID}` Register a new reqID and return the new Key

Figure 17 : Sample COMPANY Rest API

5.7. Run the Client Application Web Service

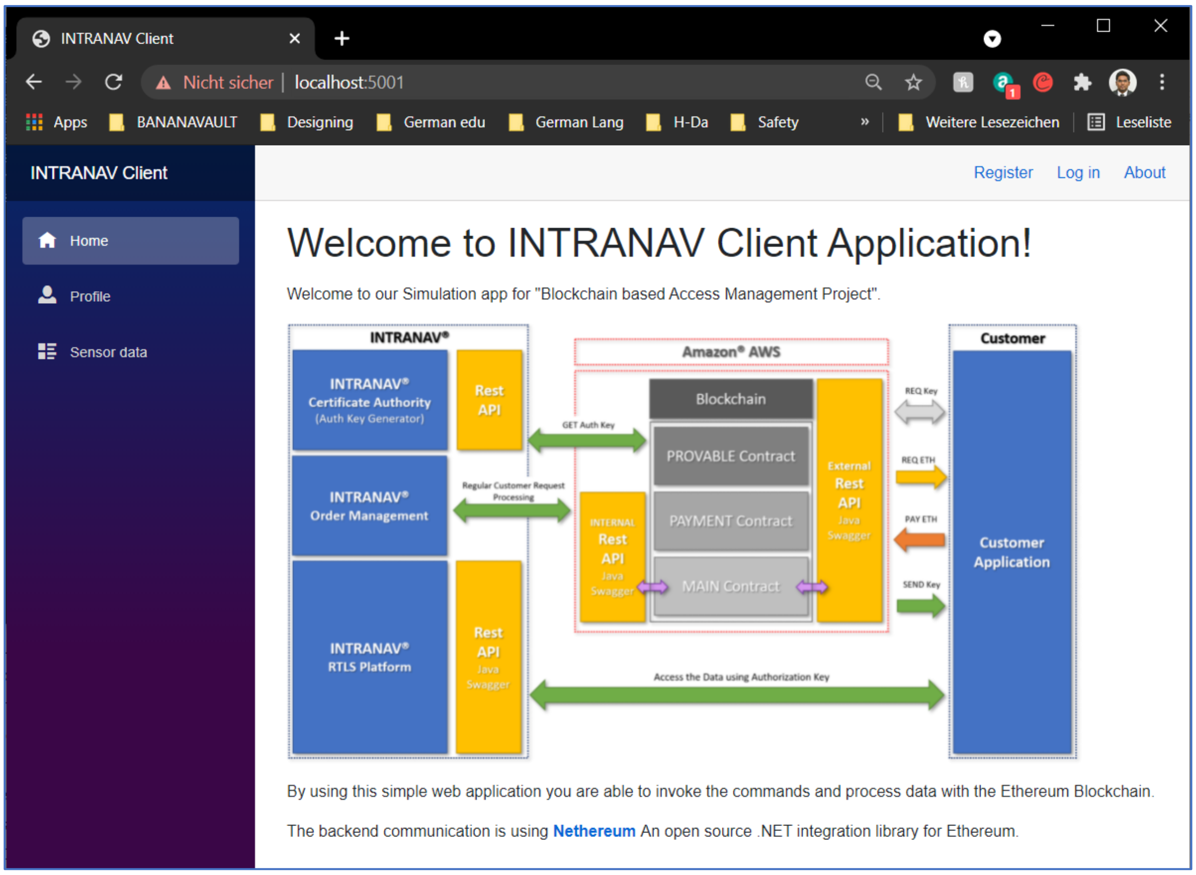
Step 1: Redirect to "\05_ClientWebApp\20_Release\" path and run the "INTRANAV Client Application.exe" file.
Example:



```
C:\Users\sachi\source\repos\50_Blockchain\01_INTRANAV_Ethereum\05_ClientWebApp\INTRANAV Client Application.exe
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\sachi\source\repos\50_Blockchain\01_INTRANAV_Ethereum\05_ClientWebApp
```

Figure 18 : CMD window of Blazor server app

Now you if you open the address in your web browser https://localhost:5001/ you will be proceeding to the web application as shown below.



Welcome to INTRANAV Client Application!

Welcome to our Simulation app for "Blockchain based Access Management Project".

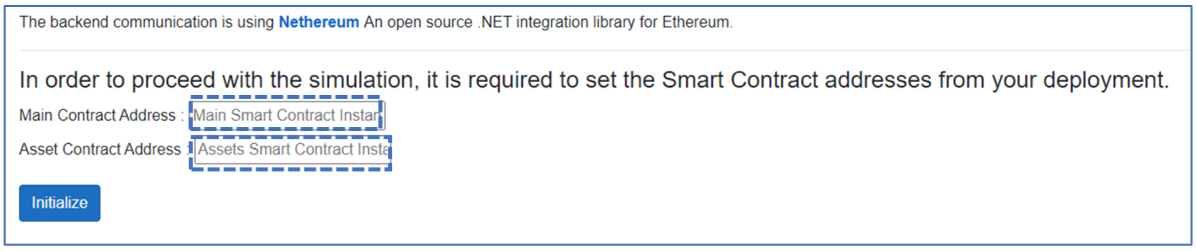
INTRANAV®
INTRANAV® Certificate Authority (Auth Key Generator)
Rest API
INTRANAV® Order Management
INTRANAV® RTLS Platform
Rest API Java Swagger
GET Auth Key
Regular Customer Request Processing
Amazon® AWS
Blockchain
PROVABLE Contract
PAYMENT Contract
MAIN Contract
INTERNAL Rest API Java Swagger
External Rest API Java Swagger
REQ Key
REQ ETH
PAY ETH
SEND Key
Access the Data using Authorization Key
Customer
Customer Application

By using this simple web application you are able to invoke the commands and process data with the Ethereum Blockchain.

The backend communication is using **Nethereum** An open source .NET integration library for Ethereum.

Figure 19 : Client App User Interface

Step 2: Initialize the application



The backend communication is using **Nethereum** An open source .NET integration library for Ethereum.

In order to proceed with the simulation, it is required to set the Smart Contract addresses from your deployment.

Main Contract Address : Main Smart Contract Instar

Asset Contract Address : Assets Smart Contract Insta

Figure 20 : Client Application Smart Contract address definition.

Copy and paste the Main and Assets smart contract instance address from above section "5.4 Deploy Smart Contracts". Then click "Initialize".

6. Amazon AWS Integration

In order to be able to deploy the blockchain and the application into the Amazon AWS platform, we have created the applications with the ability to be containerized. Which then can be easily deployed on AWS using Kubernetes cluster or docker containers directly.

Amazon AWS integration sometimes would be costly. Therefore, it is important to categorize the application based on resource usage as well as available support in order to select the matching AWS system.
Given the size and resource usage we have categorized the components as below.

Component	AWS system
Internal Communication API	Kubernetes Cluster
INTRANAV REST API Simulation	Fargate Cluster
Ganache CLI environment	Kubernetes Cluster
Client Application	Elastic Beanstalk

Table 3 : Application Components and the AWS end points.

6.1. How to deploy

Based on the specific AWS systems used in different applications the deployment do differ from each other.

For Fargate Cluster please refer to:
https://aws.amazon.com/getting-started/hands-on/deploy-docker-containers/

For Kubernetes Cluster with Docker Container please refer to:
https://openliberty.io/guides/cloud-aws.html#configuring-the-aws-cli

For Elastic Beanstalk with dot net Blazor application please refer to:
https://aws.amazon.com/blogs/developer/run-blazor-based-net-web-applications-on-aws-serverless/

7. Source

Ganache CLI, Ethereum bridge modification:
https://github.com/SachiHarshitha/GanacheCLI-with-EthBridge

Project Data:
https://github.com/SachiHarshitha/HDa_TeamProject_SS21

8. Bibliography

1. https://www.ibm.com/topics/what-is-blockchain
2. https://ethereum.org/en/what-is-ethereum/
3. https://www.ibm.com/topics/smart-contracts
4. https://www.ibm.com/topics/smart-contracts
5. https://ethereum.org/en/developers/docs/smart-contracts/
6. https://medium.com/@teexofficial/what-are-oracles-smart-contracts-the-oracle-problem-911f16821b53
7. https://ostechnix.com/blockchain-2-0-explaining-smart-contracts-and-its-types/

9. Appendix I - Figures

10. Appendix II - Tables