Wafa ben slama souei
2023/2024

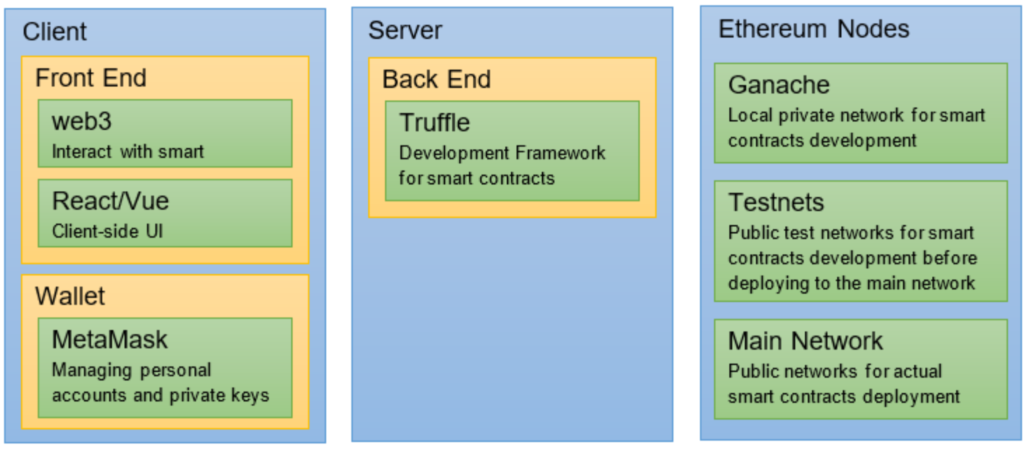
Implementation of a Dapp application

PREPARATION OF THE DEVELOPMENT ENVIRONMENT

To prepare the development environment, follow these general instructions:

	Description
PC	✓ Asus personal computer of 8 GB of RAM, equipped with an intel core i7-7500U CPU @2.70GHz processor, under windows8.1, 64bit system.
Smart contract	✓ Front end is developed using truffle, react, drizzle. ✓ Back-end is developed using solidity.
Blockchain	✓ Ganache local blockchain ✓ MetaMask is a software cryptocurrency wallet used to interact with the blockchain.

Our application will follow the architecture of Dapp application illustrated bellow.


Client
Front End
web3
Interact with smart
React/Vue
Client-side UI
Wallet
MetaMask
Managing personal accounts and private keys
Server
Back End
Truffle
Development Framework for smart contracts
Ethereum Nodes
Ganache
Local private network for smart contracts development
Testnets
Public test networks for smart contracts development before deploying to the main network
Main Network
Public networks for actual smart contracts deployment

Ethereum Decentralized Application (Dapp) Architecture and Frameworks

1. Dependencies Installation:

To install dependencies for a decentralized application (dapp) on Ethereum, you'll typically be using tools and frameworks specific to Ethereum development. Here's a general outline of the process:

a. **Ganache Personal Blockchain:** downloaded and installed from https://github.com/trufflesuite/ganache/releases
b. **Install Node.js:** Ethereum development often relies on Node.js and its package manager, npm. Install Node.js by downloading it from the official website (https://nodejs.org) and following the installation instructions.
c. **Truffle:** Truffle is a popular development framework for Ethereum. It provides a suite of tools for smart contract compilation, deployment, and testing. Install Truffle globally by running the command **npm install -g truffle.**

d. Metamask Ethereum Wallet: the extension is downloaded and installed from https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn?hl=en

2. Creation of the DSR project

a. Project setup:

// to create a project directory

- `$ mkdir DSR_Registry`
- `$ cd DSR_Registry`

// Now we initialize a new truffle project to develop our project like this:

- `$ truffle init`

//Now let's create a package.json file to install some development dependencies that will need for the project. You can do that from the command line like this:

- `$ touch package.json`

//install the dependencies

- `$ npm install`

//to create a smart contract and edit it

- `$ touch ./contracts/DSR.sol`

//compile the smart contract and ensure that there are no errors:

- `$ truffle compile`

//Configure and connect to the blockchain network : edite truffle_config.js file:

-
  ```
  module.exports = { networks: {

  development: {
          host: "127.0.0.1",     // Localhost (default: none)
          port: 7545,            // Standard Ethereum port (default:
          none)
          network_id: "*",       // Any network (default: none) }
  },solc: {optimizer:{
                enable: true,
                runs:200 }}}
  ```

//Next, we'll create a migration script inside the migrations directory to deploy the smart contract to the personal blockchain network. From your project root, create a new file from the command line like this:

- `$ touch migrations/2_deploy_contracts.js :`

```
Add this code  :
var DSR_registry = artifacts.require("./DSR_registry.sol");
module.exports = function(deployer) {
  deployer.deploy(DSR_registry);
};
```

- `$ truffle migrate`

Configure and connect to the blockchain network:

3. Creating our React.js project

//This should create a client directory in your Truffle project and bootstrap a barebones React.js project for you to start building your front-end with.

- `npx create-react-app client`

In the `truffle-config.js` file, replace the contents with the following:

```
b. const path = require("path");
c.
d. module.exports = {
e.   contracts_build_directory: path.join(__dirname, "client/src/contracts")
f. };
```

//This will make sure to output the contract build artifacts directory inside your React project.

4. Install Drizzle

// Make sure you are in the **client** directory and then run the following:

- `npm install @drizzle/store`

5. Wire up the React app with Drizzle

//Before we go further, let's start our React app by running the follow command inside our **client directory:**

- `npm start`

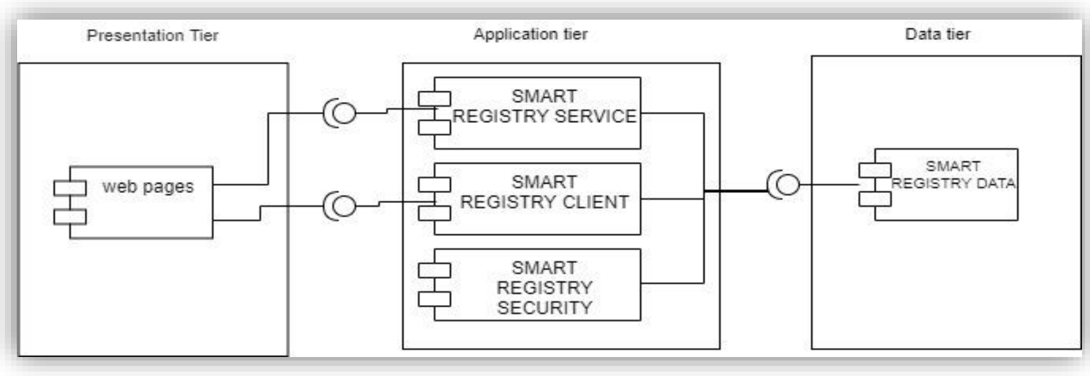
Upon completion, you will witness the launch of the Webpack server, accompanied by a visually appealing DApp web interface within your browser. The webpage will not only display the Drizzle logo but also present several forms that enable interaction with the sample contracts included in the box.

IMPLEMENTATION OF THE DAPP APPLICATION: DSR

Our DSR application is composed from four smart contract and two user interfaces.

1. Smart contracts (Smart registry service, Smart registry client, Smart registry security, Smart registry data)
2. Users' interfaces: The first is for the provider of the DSR application, the second is for the DSR clients. Bellow the architecture of the application based on the three tiers architecture.

We will show in this tutorial only the client users' interface and the smart registry client smart contract.



a. Smart registry client

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.5.16;
import "./requestActionsInterface.sol";
import "./SmartRegistryData.sol";
contract SmartRegistryClient is requestActionsInterface{
SmartRegistryData.SCDATA[]  SRD;
address proprietaire;

constructor (SmartRegistryData srd) public{
proprietaire = msg.sender;
SRD = srd.getsrd();
}

function searchDesc(string calldata category,string calldata security,string calldata legal,uint gas) external view returns(address[] memory){
address[] storage contracts = new address[](500);
uint j=0;
for(uint i = 0;i<10000;i++){
if (SRD[i].getname(i) == category){
if(filterBylegal(SRD[i])== true) {
if(filterBySecurity(SRD[i])==true){
if(filterbygas(gas)){
contracts[j]==SRD[i].addreess(i);
}
}} }}
return contracts;
}

function filterBylegal(SmartRegistryData.SCDATA memory desc) public returns (bool){
if (desc[i].getlegalvalue(i) != ""){
return true;
}else{
return false;
}}
function filterBySecurity ()public returns (bool){
if (desc[i].getsecurityvalue(i) != ""){
return true;
}else{
return false;
}}
function filterbygas(uint value) public returns (bool){
if (desc[i].getgasvalue(i) == value){
return true;
}else{
return false;
}}}
```

b. Customizing the Drizzle box to our context

The initial unboxing of Drizzle created a Truffle workspace alongside a ReactJS project, which can be found under the src/ folder. To customize this example according to our requirements, follow these steps:

Wafa ben slama souei
2023/2024

1. Copy the SmartRegistryClient.sol file into the contracts/ folder. Remove any other contracts present in the folder, except for Migration.sol and SmartRegistryClient.sol.

2. Next, open the 2_deploy_contract and Replace the existing deployment script with the following code snippet:

```
var SRC = artifacts.require("SmartRegistryClient ");

deployer.deploy(SRC).then(function() {
return deployer.deploy(SRC.address);
}).then(function() { })
```

Next, edit drizzleOptions.js, which is located under the src/ folder, as follows:
import SRC from './../build/contracts/ SmartRegistryClient..json'

```
const drizzleOptions = { web3: {
block: false, fallback: {
type: 'ws',
url: 'ws://127.0.0.1:7545'
}
},
contracts: [ SRC ], events: {,
},
polls: { accounts: 1500 }
}
```

The `drizzleOptions` object is exported as the default in this code snippet.

Within the `drizzleOptions` object, we define the following parameters:

- The `fallback` attribute specifies the URL to be used for the web3 connection in the absence of a web3 provider like MetaMask.

- The `contracts` parameter represents an array of contract artifacts that we can interact with.

- In the `events` option, we set an object that consists of contract names along with an array of event names we want to listen for. Additionally, event names can be replaced with an object containing both `eventName` and `eventOptions`, where the `eventOptions` field defines an event filter.

- Finally, the `polls` parameter determines the frequency at which Drizzle will check the blockchain for state changes. The value of `polls` is set in milliseconds, with the default being 3000 (3 seconds). In this case, Drizzle will poll every 1.5 seconds.

Once you have finished editing the `drizzleoptions.js` file, proceed to the `src/layout/home` directory.

c. Client user interface

Using the default homepage shipped with the Drizzle box, we will set up a web page for our DAPP.

```
import React, { useState, useEffect } from "react";
import { DrizzleContext } from "@drizzle/react-plugin";

// Component for displaying search results
const SearchResults = ({ results }) => {
return (
<div>
<h2>Search Results</h2>
{results.map((address, index) => (
<p key={index}>{address}</p>
))}
</div>
);
};

// Main App component
const App = () => {
const [results, setResults] = useState([]);

// Invoke the searchDesc function on component mount
useEffect(() => {
searchDescription();
}, []);

// Function invocation example
const searchDescription = async () => {
// Retrieve the Drizzle context
const { drizzle, drizzleState } = useContext(DrizzleContext.Context);
const contractInstance = drizzle.contracts. SmartRegistryClient;

try {
const result = await contractInstance.methods
.searchDesc("category", "security", "legal", 100)
.call({}, { from: drizzleState.accounts[0] });

setResults(result);
} catch (error) {
// Handle any errors
console.error(error);
}
};

return (
<div>
<h1>DApp Example</h1>
<SearchResults results={results} />
</div>
);
};

export default App;
```

Wafa ben slama souei
2023/2024

TRYING THE DAPP :

Congratulations on successfully creating your first Drizzle app using the Drizzle box! Not only have you accomplished that, but you have also established a robust development and deployment environment that will greatly simplify DApp development and testing.

Connecting Ganache to MetaMask

1. To interact with Ganache from the browser, we need to configure MetaMask. For that, in the top-left of MetaMask, you can select Custom RPC. A new dialogue box, titled New RPC URL will show up; here, you should enter http://127.0.0.1:7545 (Ganache's IP and port) and click Save.
2. After successfully establishing the connection between MetaMask and Ganache, you may have noticed that your previous MetaMask accounts have been loaded with 0 ether, which is not ideal for testing our DApp. However, there's good news! Ganache-cli has generated a set of virtual accounts, each preloaded with 100 ether. To interact with the contract, we need to import some of these accounts into MetaMask. Follow these steps to import the corresponding accounts:
    i. Open the ganache-cli output and locate the collection of private keys.
    ii. Copy a few private keys from the ganache-cli output.
    iii. Open MetaMask and click on the account avatar or account icon.
    iv. Select "Import Account" or "Import Account with Private Key" (depending on your MetaMask version).
    v. Paste one of the private keys you copied from ganache-cli into the provided field.
    vi. Click on the "Import" or "Add" button to import the account into MetaMask.
    vii. Repeat steps 4-6 for any additional accounts you want to import.

By importing these accounts into MetaMask, we will have the necessary funds to interact with the contract and fully experience our DApp.