Hotel Booking DApp: A Seamless Decentralized Solution

Created by: Nhlonipho Masuku 2022CSC1041

INTRODUCTION

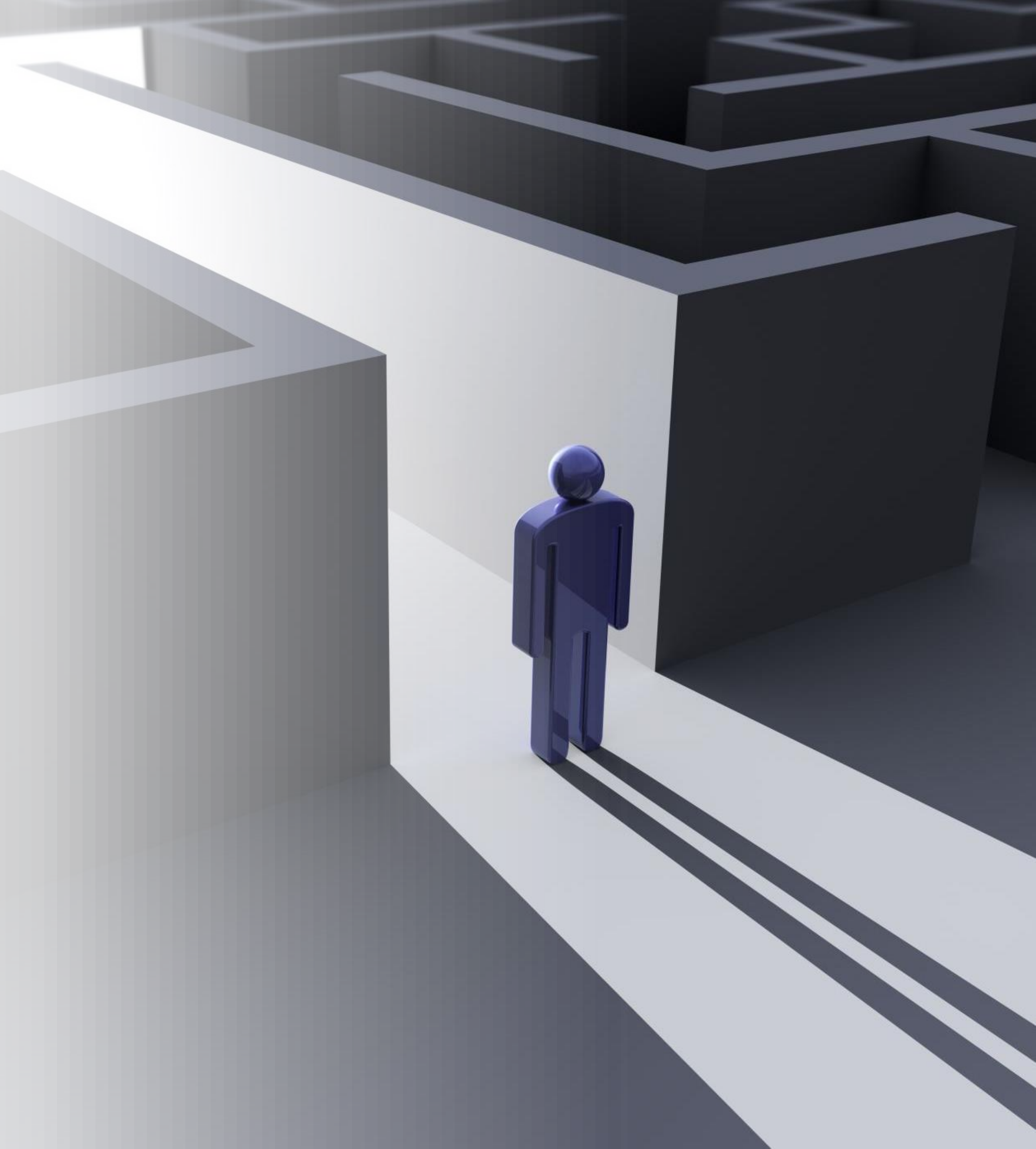
Welcome everyone to the presentation on my Hotel Booking DApp, which leverages the power of blockchain technology and smart contracts to provide a decentralized and secure platform for hotel bookings. In this presentation, I will walk you through the problem description, motivation, system architecture, tasks completed during the semester, limitations, and future enhancements.

PROBLEM DESCRIPTION

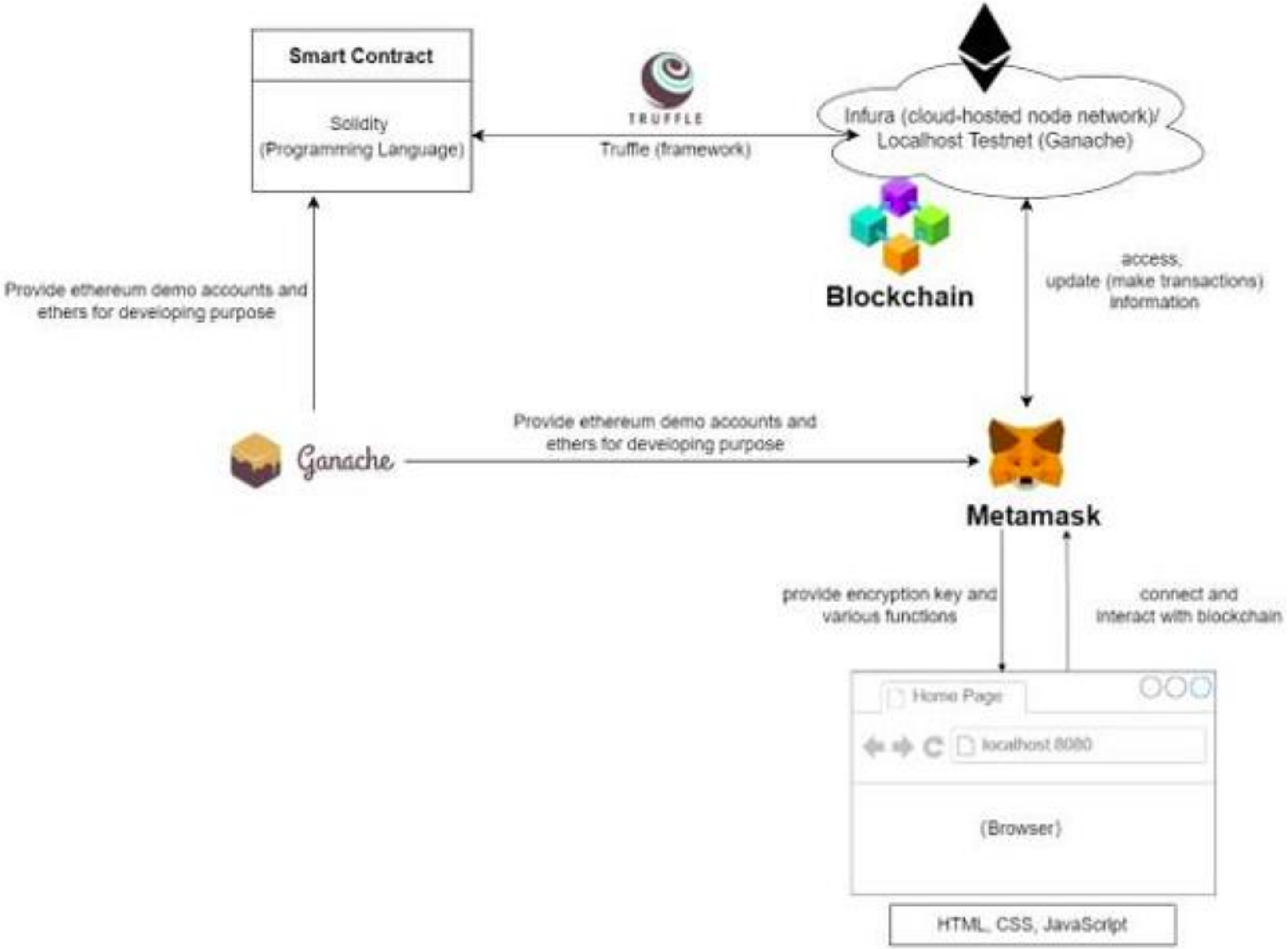
The traditional hotel booking process often involves intermediaries, high fees, and lack of transparency. It can also lead to double bookings or disputes. The aim of my Hotel Booking DApp is to overcome these challenges by providing a decentralized platform where users can directly interact with the hotel smart contract, ensuring trust, transparency, and cost-effectiveness.

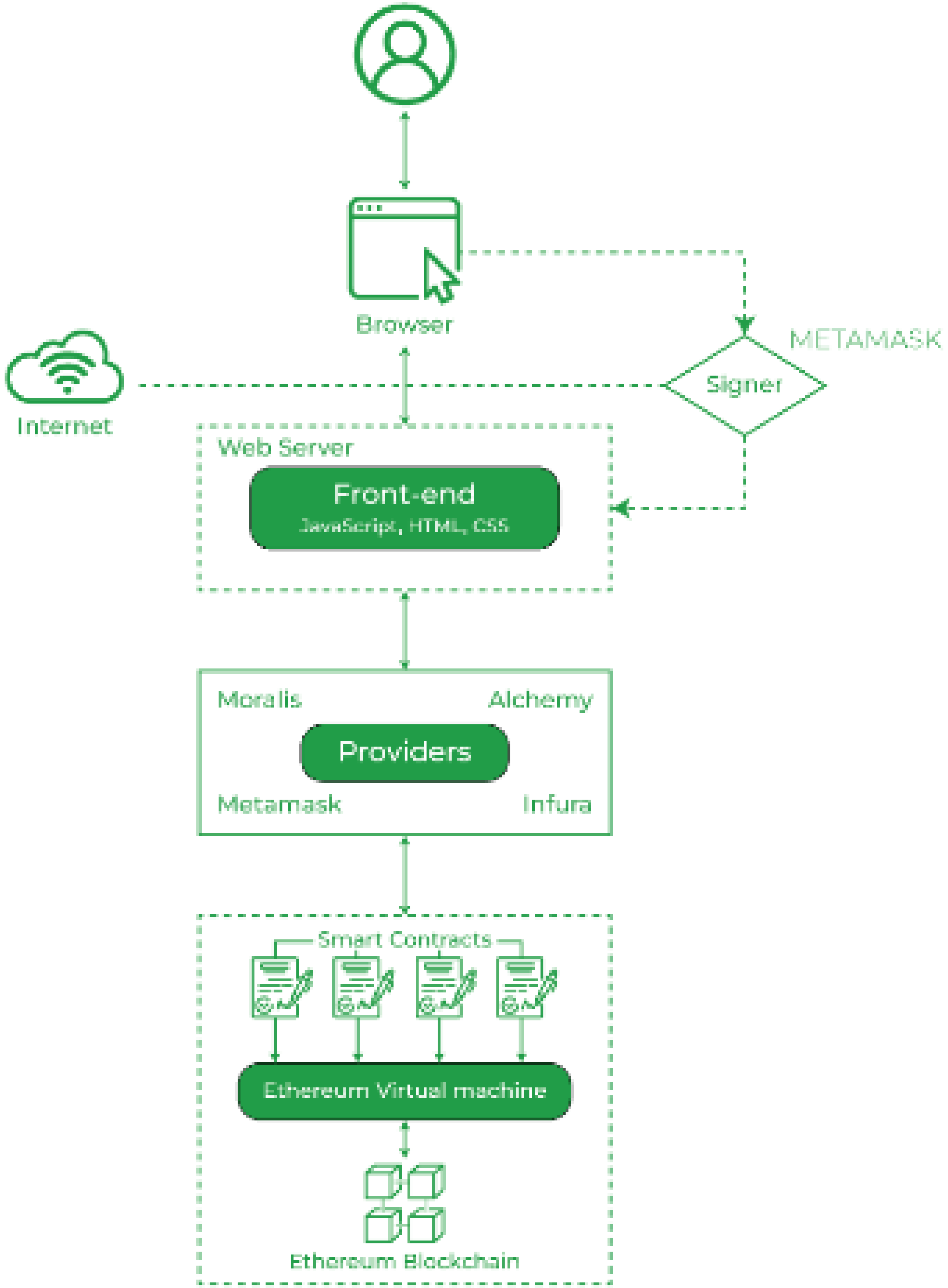
MOTIVATION

The motivation behind developing this DApp was to explore the potential of blockchain technology in the hospitality industry. By implementing a decentralized hotel booking system, we can eliminate the need for intermediaries, reduce costs, improve transparency, and provide a seamless user experience.

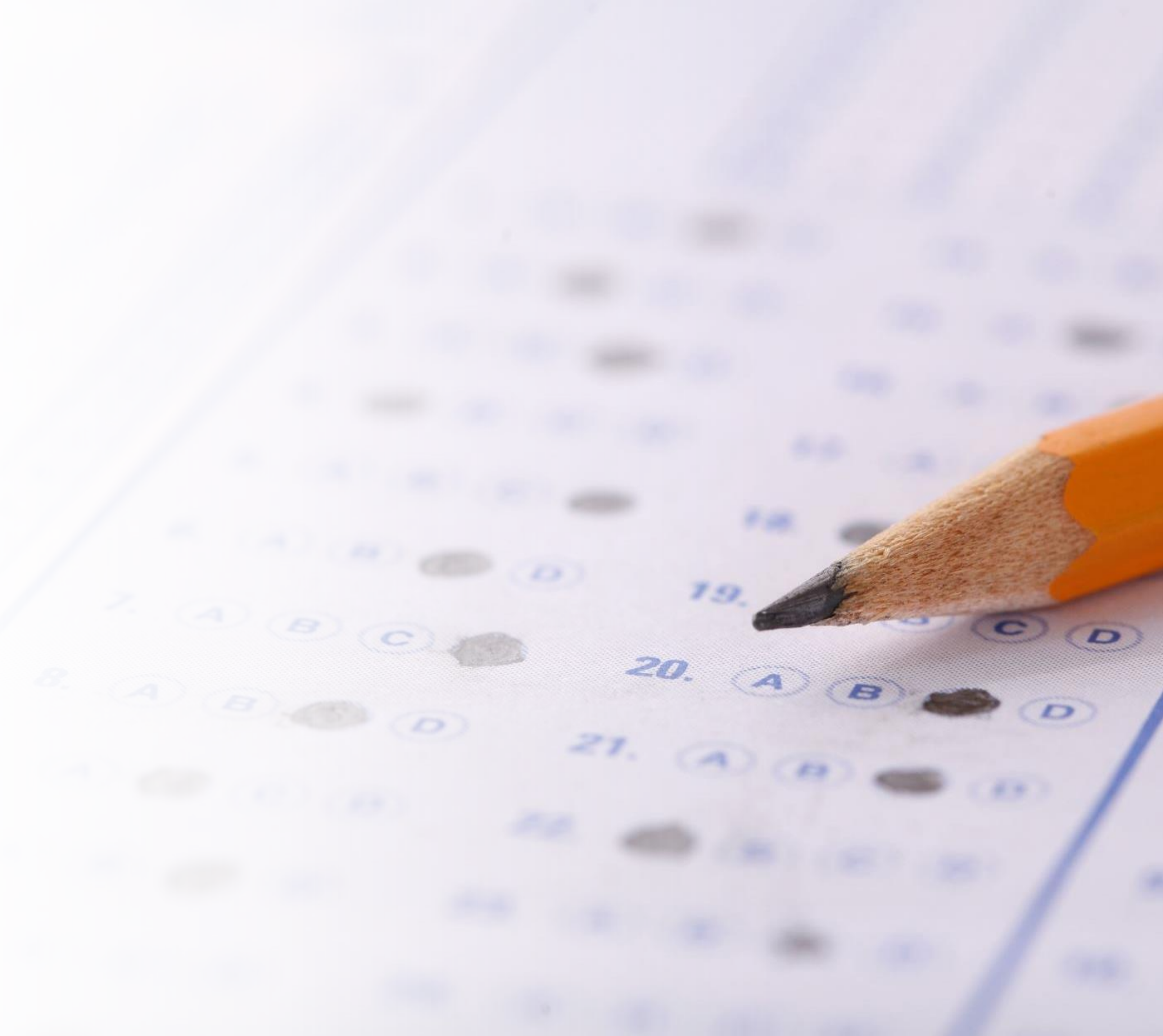


SYSTEM ARCHITECTURE:
dapp flowchart





TASKS COMPLETED IN THE SEMESTER



BLOCKCHAIN OVERVIEW:

WHAT IS IT?

- A decentralized, immutable, and transparent digital ledger.
- Consists of a chain of blocks containing transactional data.
- Operates on a peer-to-peer network without a central authority.

KEY CHARACTERISTICS

Decentralization:

- No central authority, eliminating the need for intermediaries.
- Transactions verified and stored across multiple nodes.

Transparency:

- Publicly accessible ledger for all participants.
- Enhanced trust and auditability of transactions.

Security:

- Utilizes cryptographic techniques to secure data.
- Immutable nature makes tampering nearly impossible.

Consensus Mechanism:

- Agreement among participants on the validity of transactions.
- Ensures trust and prevents double-spending.

Smart Contracts:

- Self-executing contracts with predefined rules and conditions.
- Automate processes and eliminate intermediaries.

USE CASES

•Financial Services: Payments, remittances, and asset tokenization.
•Supply Chain Management: Traceability, transparency, and anti-counterfeiting.
•Healthcare: Secure data sharing, medical records, and clinical trials.
•Voting Systems: Transparent and tamper-resistant elections.
•Decentralized Applications (DApps): Building decentralized applications.

CHALLENGES & OPPORTUNITIES

•Scalability: Addressing network capacity and transaction speed.
•Interoperability: Ensuring seamless interaction between different blockchains.
•Privacy and Data Protection: Protecting sensitive information on the blockchain.
•Regulation and Compliance: Navigating legal frameworks and industry standards.

FUTURE OUTLOOK

- Continuous innovation and adoption of blockchain technology.
- Exploration of hybrid solutions combining blockchain with other technologies.
- Integration with emerging technologies like AI, IoT, and decentralized finance.

INSTALLATION OF HYPERLEDGER AND PREREQUISITES

Prerequisites

Install	go (optional)
Install	python
Install	curl
Install	git
Install	nodejs and npm
Install	docker & docker compose

Hyperledger Fabric

mkdir	In a new working directory, clone fabric-samples repository to the folder
Install	fabric, fabric-ca, and third party docker image
Update	path i.e to fabric-samples (download location)
Kill	all docker stale and active containers

SMART CONTRACT DEVELOPMENT

- Line 1: Function declaration for makeReservation with parameters roomId, startDate, and endDate.
- Line 2: Declares the visibility of the function as public.
- Line 3: Applies the roomExists modifier to validate if the room exists.
- Line 4: Applies the roomAvailable modifier to validate if the room is available for the specified dates.
- Line 6: Checks if the endDate is greater than the startDate. Throws an error if it is not.
- Line 8: Creates a new Reservation struct in memory with the provided details and sets isPaid to false.
- Line 9: Adds the newly created Reservation struct to the reservations array.
- Line 11-13: Iterates over each day from startDate to endDate and marks the corresponding room as unavailable.
- Line 15: Emits the ReservationMade event, providing the details of the reservation.

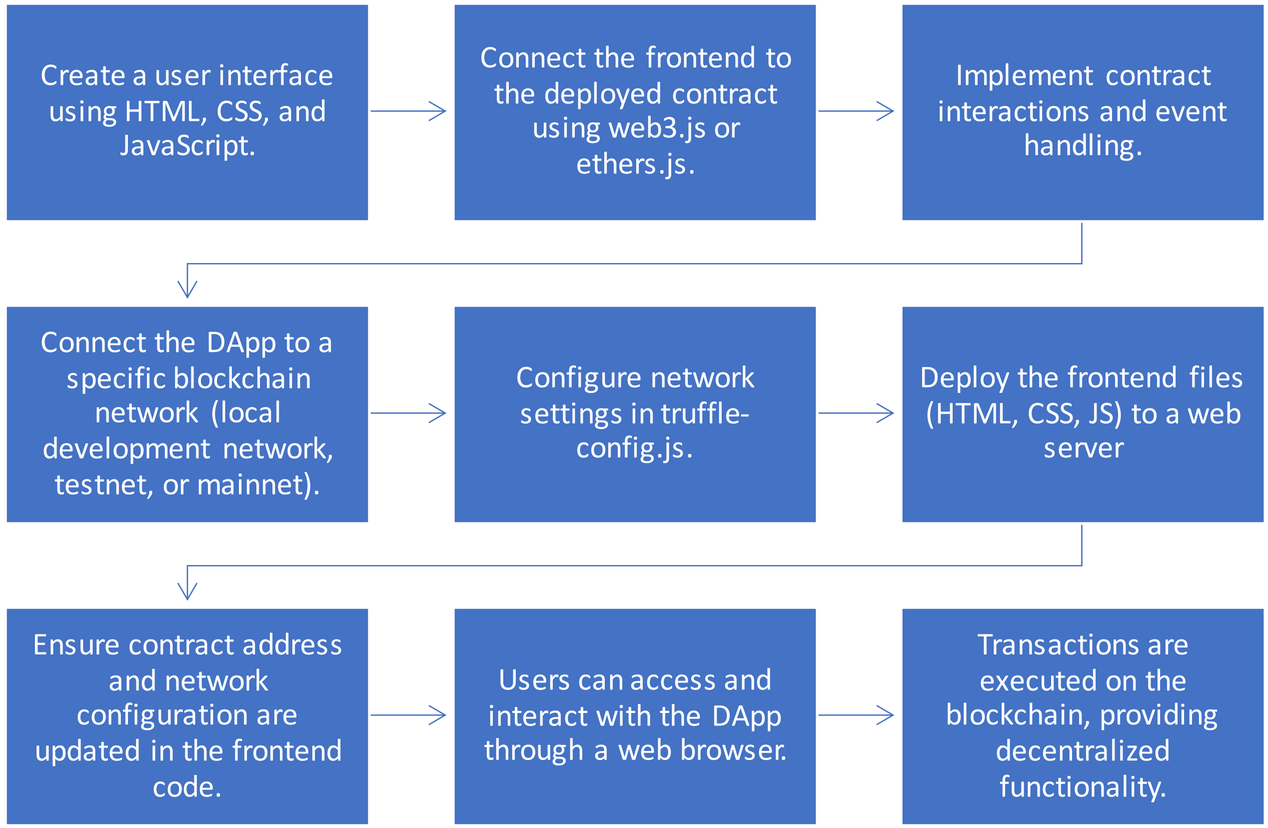
```
58      function makeReservation(uint256 roomId, uint256 startDate, uint256 endDate)
59          public
60          roomExists(roomId)
61          roomAvailable(roomId, startDate, endDate)
62      {
63          require(endDate > startDate, "End date should be greater than start date");
64
65          Reservation memory reservation = Reservation(msg.sender, roomId, startDate, endDate, false)
66          reservations.push(reservation);
67
68          for (uint256 i = startDate; i <= endDate; i++) {
69              rooms[roomId].isAvailable = false;
70          }
71
72          emit ReservationMade(reservations.length - 1, msg.sender, roomId, startDate, endDate);
73      }
74
75      function makePayment(uint256 reservationId, uint256 amount) public payable {
76          require(reservationId < reservations.length, "Invalid reservation ID");
77
78          Reservation storage reservation = reservations[reservationId];
79
80          require(!reservation.isPaid, "Payment has already been made");
81          require(msg.sender == reservation.guest, "Only the guest can make the payment");
82          require(msg.value >= amount, "Insufficient payment amount");
83
84          reservation.isPaid = true;
85
86          emit PaymentMade(reservationId, msg.sender, amount);
87      }
88
89
90      function cancelReservation(uint256 reservationId) public {
91          require(reservationId < reservations.length, "Invalid reservation ID");
92
93          Reservation storage reservation = reservations[reservationId];
94
95          require(!reservation.isPaid, "Payment has already been made");
96          require(msg.sender == reservation.guest, "Only the guest can cancel the reservation");
97
98          for (uint256 i = reservation.startDate; i <= reservation.endDate; i++) {
99              rooms[reservation.roomId].isAvailable = true;
100         }
101
102         emit ReservationCancelled(reservationId, msg.sender);
103     }
```

TRUFFLE SETUP & DAPP DEVELOPMENT

TRUFFLE:

- Install Node.js and npm
- Install Truffle globally: npm install -g truffle
- Initialize a new Truffle project: truffle init
- Create/paste contract in the correct folder
- Compile the smart contract: truffle compile
- Configure a deployment network in the truffle-config.js file.
- Migrate the smart contract to the network: truffle migrate

DAPP



WEB3 & METAMASK INTEGRATION

Install

Web3 Installation: Install Node.js and npm (if not already installed). Open the terminal/command prompt and run: npm install web3.

Install

Metamask Wallet Installation: Install the Metamask extension for your preferred browser. Create a new Metamask wallet or import an existing one. Set up a password and securely store the wallet seed phrase.

Connect

Metamask Integration in DApp: Connect DApp to the Metamask wallet

Import

Import the web3 library: import Web3 from 'web3'.

Create

Create an instance of the Web3 object: const web3 = new Web3(window.ethereum)

- To Interact with the smart contract:
- Load the contract using the contract's ABI and address: const contract = new web3.eth.Contract(abi, address)
- Call contract methods: contract.methods.methodName().call({ from: account }).
- Deploy the DApp on a local development network
- Ensure Metamask is connected and select the appropriate network.
- Interact with the DApp and verify smooth interaction with the blockchain network.
- Integrating Metamask with the DApp enables seamless user interaction with the blockchain network, allowing users to manage accounts, sign transactions, and securely interact with smart contracts.

APP CODE & STYLESHEET DEVELOPMENT

App.js (javascript)

```javascript
// Import necessary dependencies and styles for the React application
import React, { useState, useEffect } from "react";
import HotelBooking from "./contracts/HotelBooking.json";
import getWeb3 from "./getWeb3.js";
import "./App.css";

// App component
const App = () => {
  // State variables
  const [web3, setWeb3] = useState(null); // Web3 instance state
  const [accounts, setAccounts] = useState([]); // User accounts state
  const [contract, setContract] = useState(null); // Contract instance state

  // Use effect hook to initialize web3 and contract on component mount
  useEffect(() => {
    const init = async () => {
      try {
        // Get web3 instance
        const web3Instance = await getWeb3();
        setWeb3(web3Instance);

        // Get user accounts
        const userAccounts = await web3Instance.eth.getAccounts();
        setAccounts(userAccounts);

        // Get contract instance
        const networkId = await web3Instance.eth.net.getId();
        const deployedNetwork = HotelBooking.networks[networkId];
        const contractInstance = new web3Instance.eth.Contract(
          HotelBooking.abi,
          deployedNetwork && deployedNetwork.address
        );
```

App.css (cascading style sheet)

```css
body {
  font-family: "Arial", sans-serif;
  margin: 0;
  padding: 0;
  background: linear-gradient(90deg, #00c9ff 0%, #92fe9d 100%);
  background-image: url('./background.jpg');
  background-repeat: repeat;
  font-family: "Sansita Swashed", cursive;
  display: flex;
  align-items: center;
  justify-content: center;
  min-height: 100vh;
  position: relative;
}

h2, p {
  color: #fff;
  font-size: 24px;
  text-shadow: 2px 2px 6px rgba(0, 0, 0, 0.8);
}

h2 {
  font-size: 30px;
}

h2 {
  padding: 10px;
  border-radius: 5px;
}
```

HOTEL BOOKING DAPP USER INTERFACE

DEPLOYMENT & TESTING

ESURING CORRECTNESS & RELIABILITY:

Unit Testing:

Testing methodology for decentralized applications (DApps). It focuses on testing individual units of code to ensure they function correctly in isolation.

- After successful deployment (i.e truffle environment)
- Write Tests: Write individual test functions to verify the expected behavior of each unit. Test functions should simulate the necessary inputs, execute the unit being tested, and check the output against expected results.
- Run tests: truffle test
- Implementing a combination of testing methodologies ensures the correctness, reliability, and security of the DApp. A comprehensive testing approach is crucial for successful deployment and user satisfaction.

CHALLENGES & LIMITATIONS

RESPONSIVE UI: THE REACT DAPP IS CURRENTLY NOT FULLY RESPONSIVE, WHICH MAY LEAD TO USABILITY ISSUES ON DIFFERENT SCREEN SIZES OR DEVICES.

METAMASK DEPENDENCY: THE INTEGRATION WITH METAMASK ASSUMES USERS HAVE THE EXTENSION INSTALLED AND CONFIGURED. USERS WITHOUT METAMASK WILL NOT BE ABLE TO INTERACT WITH THE DAPP.

USER EXPERIENCE: THE DAPP MAY LACK USER-FRIENDLY FEATURES, INTUITIVE GUIDANCE, OR INFORMATIVE ERROR MESSAGES, LEADING TO A SUBOPTIMAL USER EXPERIENCE.

LIMITED BLOCKCHAIN SUPPORT: THE DAPP MAY BE DESIGNED FOR A SPECIFIC BLOCKCHAIN NETWORK OR REQUIRE MODIFICATIONS TO SUPPORT DIFFERENT NETWORKS.

UPGRADABILITY CHALLENGES: UPGRADING THE SMART CONTRACT MAY REQUIRE ADDITIONAL STEPS, SUCH AS DATA MIGRATION OR REDEPLOYMENT, WHICH CAN BE COMPLEX AND TIME-CONSUMING.

REGULATORY COMPLIANCE: COMPLIANCE WITH LEGAL AND REGULATORY REQUIREMENTS, SUCH AS DATA PROTECTION OR FINANCIAL REGULATIONS, MAY IMPOSE CONSTRAINTS ON THE DAPP'S FUNCTIONALITIES

FURTURE ENHANCEMENT

- Responsive Design: Implement a fully responsive user interface to ensure seamless user experience across various screen sizes and devices.
- Enhanced User Interface: Improve the visual design, layout, and user flow to provide an intuitive and engaging user experience.
- Mobile Support: Develop a mobile application version of the DApp to cater to users who prefer mobile devices for blockchain interactions.
- Metamask Alternatives: Explore integration with alternative wallets or on-chain authentication methods to provide users with more options for interacting with the DApp.
- Offline Functionality: Implement offline support to allow users to access and interact with the DApp even when they have limited or no internet connectivity.
- Smart Contract Upgradability: Design the smart contract with upgradability in mind, enabling easier contract upgrades without compromising data integrity or user security.
- Integration with Additional Blockchain Networks: Extend support for multiple blockchain networks, providing users with flexibility and choice in network selection.
- Advanced Functionality: Add advanced features such as user ratings and reviews, additional payment options, or integration with external APIs to enhance the DApp's functionality and utility.

APPENDIX: CODE SNIPPETS:

MIGRATION SCRIPT

```
const HotelBooking = artifacts.require("./HotelBooking.sol");
module.exports = function (deployer)
{
  deployer.deploy(HotelBooking);
};
```

DAPP CODE

:imports

```
import React, { useState, useEffect } from "react";
import HotelBooking from "./contracts/HotelBooking.json";
import getWeb3 from "./getWeb3.js";
import "./App.css";
```

:network, contract & accounts instances

```
        // Get network provider and web3 instance
        const web3 = await getWeb3();

        // Get the contract instance
        const networkId = await web3.eth.net.getId();
        const deployedNetwork = HotelBooking.networks[networkId];
        const contract = new web3.eth.Contract(HotelBooking.abi,
        deployedNetwork && deployedNetwork.address,
        "0x14919268E9115A99fD01E6ad67E857fff0f592d8");
        setContract(contract);
        // Get the user accounts
        const accounts = await web3.eth.getAccounts();
        setDefaultAccount(accounts[0]);
        // Check if the account is the manager/owner
        const manager = await contract.methods.owner().call();
        setIsManager(manager === accounts[0]);
```

:initialization and configuration of the web3 instance in the DApp

```
import Web3 from "web3";

const getWeb3 = () => {
  return new Promise((resolve, reject) => {
    // Wait for loading completion to avoid race conditions with web3 injection timing
    window.addEventListener("load", async () => {
      // Modern dapp browsers...
      if (window.ethereum) {
        const web3 = new Web3(window.ethereum);
        try {
          // Request account access if needed
          await window.ethereum.enable();
          // Acccounts now exposed
          resolve(web3);
        } catch (error) {
          reject(error);
        }
      }
      // Legacy dapp browsers...
      else if (window.web3) {
        // Use Mist/MetaMask's provider.
        const web3 = window.web3;
        console.log("Injected web3 detected.");
        resolve(web3);
      }
      // Fallback to localhost; use dev console port by default...
      else {
        const provider = new Web3.providers.HttpProvider("http://127.0.0.1:9545");
        const web3 = new Web3(provider);
        console.log("No web3 instance injected, using Local web3.");
        resolve(web3);
      }
    });
  });
};

export default getWeb3;
```

REFERENCES

- Solidity Documentation: https://docs.soliditylang.org/
- Truffle Documentation: https://www.trufflesuite.com/docs/truffle/overview
- React Documentation: https://reactjs.org/docs/getting-started.html
- Web3.js Documentation: https://web3js.readthedocs.io/
- Metamask Documentation: https://docs.metamask.io/
- Ethereum Developer Tools: https://ethereum.org/developers/tools/
- Blockchain Basics: https://www.ibm.com/blockchain/what-is-blockchain

THANK YOU FOR YOUR ATTENTION