RQF LEVEL 5

SWDFB501

SOFTWARE DEVELOPMENT

Fundamental of Blockchain Application

TRAINEE'S MANUAL

October, 2024

FUNDAMENTAL OF BLOCKCHAIN APPLICATION

2024

AUTHOR’S NOTE PAGE (COPYRIGHT)

© Rwanda TVET Board

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer’s and trainee’s manuals for the TVET Certificate V in Software Development specifically for the module "**SWDFB501: Fundamental of Blockchain Application**."

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Asher Emmanuel

Production Team

Authoring and Review

MUKARUGWIZA Annonciatha

IRADUKUNDA Chance Desire

NTEZIRIZAZA Jean Pierre

Validation

NTIGURIRWA Jules

MALAYIKA SYLVIE

IRARORA JONAH

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

MUKANGWIJE Yvonne

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

NDAYISABA Olivier

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

ABI: Application Binary Interface
AML: Anti-Money Laundering
BTC: Bitcoin
DAO: Decentralized Autonomous Organization
DApp: Decentralized Application
DeFi: Decentralized Finance
DEX: Decentralized Exchange
DPoS : Delegated Proof of Stake
ECDSA: Elliptic Curve Digital Signature Algorithm
ETH: Ethereum
EVM: Ethereum Virtual Machine
FOMO: Fear of Missing Out
FUD: Fear, Uncertainty, and Doubt
ICO: Initial Coin Offering
IEO: Initial Exchange Offering
KOICA: Korean International Cooperation Agency
KYC: Know Your Customer
L1: Layer 1
L2: Layer 2
LP: Liquidity Pool
NFT: Non-Fungible Token
P2P: Peer-to-Peer
PoA: Proof of Authority
PoS : Proof of Stake
PoW : Proof of Work
PoW: Proof of Weight
RTB: Rwanda TVET Board
SHA : Secure Hash Algorithm
TPS: Transactions Per Second
TPS: Transactions Per Second
TQUM Project: TVET Quality Management Project
TX: Transaction

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in Software Development specifically for the module of **"Fundamental of Blockchain Application".** Students enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

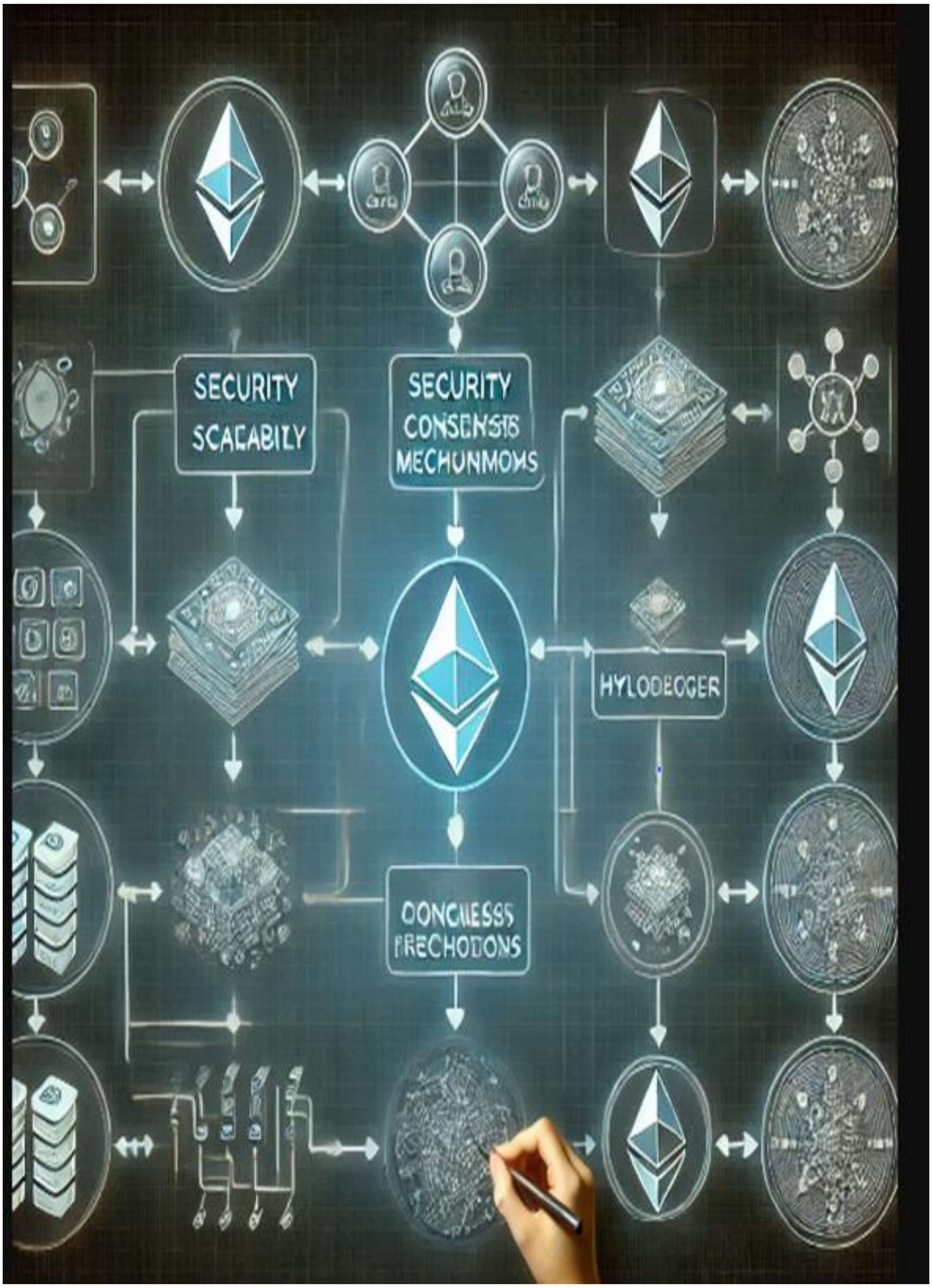
The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labor market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: SWDFB501 FNDAMENTAL OF BLOCKCHAIN APPLICATION

Learning Outcome 1: Design Blockchain System Architecture
Learning Outcome 2: Apply Solidity Basics
Learning Outcome 3: Develop Smart Contracts System
Learning Outcome 4: Apply Frontend Integration


SECURITY
SCALABILY
SECURITY
CONSENSIS
MECHUNMOMS
HYLODEGGER
OONCNESSS
PRECHODONS

Indicative Contents
1.1 Identification of Blockchain Requirements
1.2 Selecting Blockchain Technologies
1.3 Designing the Architecture of Blockchain Application

Key Competencies for Learning Outcome 1: Design Blockchain System Architecture

Knowledge	Skills	Attitudes
● Description of Blockchain requirements ● Description of blockchain layers ● Description of blockchain principles ● Description of types of Consensus mechanism ● Identification of the types of attacks and vulnerabilities of blockchain ● Description of blockchain architecture	● Applying blockchain use cases ● Selecting Blockchain Technologies ● Using Consensus Mechanism ● Designing system architecture ● Drawing blockchain architecture	● Being creative in designing blockchain project ● Being updated on the latest technologies of blockchain ● Being Innovative in blockchain technology ● Being collaborative with developers, designers, and stakeholders in block chain project

Duration: 10 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe clearly Blockchain requirements based on project purpose.

2. Describe Accurately the different layers of blockchain
3. Describe clearly core blockchain principles
4. Describe clearly different types of consensus mechanisms
5. Identify clearly types of blockchain attacks and vulnerabilities
6. Describe clearly blockchain architecture
7. Apply properly blockchain use cases based on specific requirements
8. Use properly blockchain technologies
9. Design properly blockchain system architecture

Resources

Equipment	Tools	Materials
● Computer	● EdrawMax ● Figma ● Browser	● Internet ● Electricity

Indicative content 1.1: Identification of Blockchain requirement

Duration: 3 hrs

Theoretical Activity 1.1.1: Description of Blockchain Requirements

Tasks:

1: Answer the following questions:

i. What do you understand by the following terms:

   a. Blockchain
   b. Cryptography
   c. Private Keys
   d. Public keys
   e. Wallet
   f. Addresses
   g. Blocks
   h. Blockchain transaction
   i. Merkle Trees
   j. Hierarchical Deterministic Wallets
   k. Mnemonic Seeds
   l. Smart Contracts

ii. Explain how blockchain transaction works.

iii. Give a brief History of blockchain

iv. Suggest any Types of Blockchain

v. Explain Blockchain Principles

vi. Identify Functionalities of blockchain

vii. What are Pros and Cons of blockchain

2: Write your findings on paper or flipchart

3: Present your findings to the trainer or colleagues

4: For more clarification read the key readings 1.1.1.

5: In addition, ask questions where necessary.

Key readings 1.1.1.: Description of Blockchain Requirements

- **Description of blockchain key concepts**

✓ **Blockchain**

A **blockchain** is a distributed digital ledger technology that records transactions across a network of computers in a secure, immutable, and transparent way. It consists of blocks, where each block contains a list of transactions.

Once a block is completed, it links to the next block, forming a chain. This chain of blocks is decentralized, meaning no single entity controls it.

Blocks: is a collection of data (transactions) that is permanently recorded on the blockchain. Each "block" is like a page in a notebook. It contains information like transaction details, the time, and a special code called a "hash" that makes it unique.

Chain: These blocks are connected to each other in a series, like a chain. Each block has a reference to the block before it. This connection is what makes the information safe because if someone tries to change one block, it will affect the entire chain

The key features of blockchain include:

- **Decentralization**: No central authority controls the blockchain; all participants share control.
- **Transparency**: All transactions are visible to participants within the network.
- **Immutability**: Once data is recorded in a block, it cannot be altered without altering all subsequent blocks.
- **Security**: Uses cryptographic methods to secure data and transactions.

✓ **Cryptography**

Cryptography is the practice of securing information by converting it into unreadable formats for unauthorized users. In blockchain, cryptography is crucial for ensuring the privacy, security, and integrity of data.

The two main cryptographic techniques used in blockchain are:

- **Hashing**: Converts data into a fixed-size string of characters, which is unique to the input data.
- **Public-Key Cryptography**: Uses pairs of cryptographic keys (public and private) to encrypt and decrypt data. This ensures secure communication and transaction validation within the blockchain.

✓ **Public Keys**

A **public key** is an openly shareable cryptographic key that corresponds to a private key. Public keys are used to receive blockchain transactions, acting as an address for others to send assets. While a public key can be shared openly, it does not provide access to the owner's assets, which are controlled by the private key.

Example: If you want to receive cryptocurrency, you would share your public key (address) with the sender.

✓ **Wallet**

A **blockchain wallet** is a software application or hardware device that stores public and private keys and interacts with blockchain networks to enable users to send, receive, and monitor their cryptocurrency or digital assets.

Types of Wallet :

- **Hot Wallets**: Connected to the internet (e.g., software wallets like MetaMask or Trust Wallet).
- **Cold Wallets**: Not connected to the internet, offering greater security (e.g., hardware wallets like Ledger or Trezor).

✓ **Addresses**

A **blockchain address** is a unique string of characters derived from a public key. It represents the destination for a transaction, similar to an email address for sending and receiving messages. Blockchain addresses are used to receive cryptocurrency or other digital assets.

- **Example**: A Bitcoin address looks something like 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa.

✓ **Blocks**

A **block** is a collection of data or transactions that are bundled together and added to the blockchain.

Components of block:

- **Transaction data**: Details of all the transactions included in the block.
- **Timestamp**: When the block was added.
- **Previous block hash**: A reference to the previous block, ensuring immutability and creating the "chain."
- **Merkle Root**: A cryptographic summary of all transactions in the block.

✓ **Blockchain Transaction**

A **blockchain transaction** refers to the transfer of data or assets between participants on a blockchain network. This could be a transfer of cryptocurrency, digital tokens, or data verification. Transactions are verified by network nodes through consensus mechanisms (e.g., Proof of Work, Proof of Stake) and then grouped into a block.

Example: Sending 1 Bitcoin from one wallet address to another is a blockchain transaction.

✓ **Merkle Trees**

A **Merkle Tree** is a data structure used in blockchain to efficiently verify the integrity and consistency of data. It is composed of **leaves** (individual transaction hashes) and **branches** that merge into a single **Merkle Root**.

The root represents the cryptographic summary of all transactions in the block, allowing quick verification without checking every individual transaction.

Example: If someone needs to verify that a specific transaction was included in a block, they can check the transaction hash against the Merkle Root.

✓ **Hierarchical Deterministic Wallets (HD Wallets)**

An **HD Wallet** is a type of blockchain wallet that generates a hierarchical tree of key pairs from a single seed (the master key). This allows users to create an unlimited number of public and private keys from one source, improving wallet management. HD wallets use **BIP 32** protocol for deriving keys.

- **Benefit**: Users can manage multiple addresses while backing up only one seed phrase.

✓ **Mnemonic Seeds**

A **Mnemonic Seed** (or seed phrase) is a human-readable series of words generated by a blockchain wallet, used to derive a user's private keys and restore access to their wallet. Typically, it consists of 12 to 24 words.

Example: A seed phrase might look like "apple dolphin umbrella coffee tree cloud banana." This phrase allows a user to recover their wallet if they lose access.

✓ **Smart Contracts**

A **Smart Contract** is a self-executing contract with the terms of the agreement directly written into code. The contract automatically enforces and executes actions (e.g., payment transfers) once pre-defined conditions are met. Smart contracts run on blockchain networks like Ethereum, ensuring transparency, security, and immutability.

Example: A smart contract might release payment to a freelancer only after the project is marked complete in the system.

✓ **History of blockchain**

- **Early Foundations (1970s–1990s)**

Blockchain's roots come from cryptography and digital signatures. In 1991, Stuart Haber and Scott Stornetta developed a system for timestamping documents using cryptographic methods, which formed the basis of blockchain.

- **Creation of Bitcoin (2008)**

In 2008, an anonymous person or group called **Satoshi Nakamoto** introduced **Bitcoin**, the first successful blockchain application. Nakamoto solved the **double-spending problem** by creating a decentralized system where transactions are grouped in blocks, secured by cryptographic proofs. Bitcoin's blockchain officially launched in **2009**.

- **Beyond Bitcoin: Ethereum and Smart Contracts (2015)**

In **2015**, **Vitalik Buterin** launched **Ethereum**, a new blockchain that could run **smart contracts**.

Smart contracts are programs that automatically execute actions (like payments) when certain conditions are met.

This made Ethereum the foundation for decentralized applications (dApps) and expanded blockchain's uses beyond cryptocurrency.

- **Blockchain for Business and Industry (2016–2020)**

From 2016, industries like finance, supply chains, and healthcare began exploring blockchain for secure data sharing, traceability, and efficiency. Platforms like **Hyperledger** and **Corda** helped businesses use blockchain in private, permissioned networks.

- **New Innovations (2021–Present)**

  Blockchain continues to grow with new trends such as:

- **NFTs (Non-Fungible Tokens)**: Digital collectibles and assets recorded on blockchains.
- **DeFi (Decentralized Finance)**: Financial services without middlemen.
- **Layer 2 Solutions**: Technologies like **Polygon** that make blockchains faster and cheaper.
- **Proof of Stake (PoS)**: A new energy-efficient way to secure blockchains, now adopted by **Ethereum 2.0**.

  Blockchain has evolved from a technology for digital money into a tool that can transform many industries, offering decentralized solutions for secure transactions, data integrity, and trust.

✓ **Types of Blockchain**

- **Public Blockchain**

  A **public blockchain** is open to everyone. Anyone can join the network, view transactions, and participate in validating transactions (mining or staking). It is decentralized, meaning no single entity controls it. Public blockchains are highly secure due to their transparency and community-driven nature, but they can be slower due to the number of participants.

  Examples: Bitcoin, Ethereum.

- **Private Blockchain**

  A **private blockchain** is restricted to specific users. Only selected participants have permission to view, add, or validate transactions. It is controlled by an organization, making it faster but less decentralized. Private blockchains are often used by companies for internal processes.

  Examples: Hyperledger, Corda.

- **Consortium Blockchain (Federated Blockchain)**

  A **consortium blockchain** is a hybrid between public and private blockchains. It is partially decentralized, where multiple organizations jointly manage the network. Only approved participants can validate transactions, but it allows collaboration between businesses or entities while maintaining security and control.

  Examples: Quorum, R3 Corda.

- **Hybrid Blockchain**

  A **hybrid blockchain** combines features of both public and private blockchains. It allows certain data to be open to the public while keeping sensitive information private.

This makes it flexible, allowing companies to decide which parts of the blockchain are public or private based on their needs.

Examples: Dragonchain.

- ✓ **Blockchain Principles**
- **Decentralization**: No single entity controls the network.
- **Immutability**: Data once recorded cannot be changed.
- **Transparency**: Transactions are open for all to see (in public blockchains).
- **Security**: Cryptographic protection ensures the network's safety.
- **Consensus Mechanism**: Agreement among nodes ensures valid transactions.
- **Peer-to-Peer Network**: Direct communication between participants, no intermediaries.
- **Tokenization**: Digital representation of assets for easy transfer and trade.
- **Anonymity/Pseudonymity**: Users can remain anonymous, protecting their privacy.
- ✓ **Functionalities of blockchain**
- **Decentralized Ledger**: No central authority, data stored across multiple nodes.
- **Transaction Validation**: Consensus mechanisms ensure valid and secure transactions.
- **Immutable Records**: Once recorded, data cannot be altered or deleted.
- **Smart Contracts**: Automated, self-executing contracts without intermediaries.
- **Transparency**: Public access to transaction history, while preserving user privacy.
- **Security**: Strong cryptographic protections prevent fraud and tampering.
- **Peer-to-Peer Transactions**: Direct transactions without middlemen.
- **Tokenization**: Real-world assets can be represented and traded as tokens.
- **Traceability**: Complete visibility of transactions, ideal for auditing and supply chain management.
- ✓ **Pros and Cons of blockchain**
- **Pros of blockchain**
  - **Decentralization**: No need for intermediaries
  - **Transparency**: Public and verifiable transactions
  - **Transparency**: Public and verifiable transactions
  - **Efficiency**: Faster, peer-to-peer transactions
  - **Cost Reduction**: Eliminates middlemen and fees
  - **Immutability**: Records cannot be tampered with
  - **Tokenization**: Enables asset ownership and transfer
- **Cons of blockchain**
  - **Scalability**: Networks can slow down with increased use
  - **Energy Consumption:** Proof of Work is energy-intensive
  - **Lack of Regulation:** Unclear legal frameworks
  - **Privacy Concerns**: Transactions are traceable

- **Irreversibility:** Mistakes cannot be undone
- **Regulatory Uncertainty**: Inconsistent laws and regulations globally

✓ **Blockchain Company solutions**

- Blockchain companies offer a wide range of solutions tailored to different industries and use cases.
- **key solutions provided by blockchain companies include:**

- **Supply Chain Management**

  Blockchain ensures transparency and traceability in supply chains, allowing for real-time tracking of goods, reducing fraud, and improving inventory management.

  Example of companies use supply chain:

  - **IBM Blockchain**: Helps companies like Walmart manage and trace food supply chains.
  - **VeChain**: Provides blockchain solutions to track products in various industries, including agriculture and pharmaceuticals.

- **Decentralized Finance (DeFi)**

  DeFi platforms use blockchain to offer financial services such as lending, borrowing, trading, and earning interest without intermediaries.

  Example of companies use Decentralized Finance:

  - **Aave**: A decentralized finance protocol for lending and borrowing cryptocurrencies.
  - **Uniswap**: A decentralized exchange (DEX) allowing peer-to-peer crypto asset swaps.

- **Digital identity**

  Blockchain provides secure, tamper-proof digital identity solutions that empower individuals to control their personal data and enhance authentication.

  Example of Companies use Digital identity:

  - **Civic**: Provides decentralized identity verification services for secure user authentication.
  - **Sovrin**: Offers self-sovereign identity solutions where individuals manage their identities without central authorities.

- **Healthcare Solutions**

  Blockchain secures patient data, ensures interoperability, and improves the accuracy of medical records.

  Example of Companies use Blockchain in healthcare solutions:

  - **Medicalchain**: Uses blockchain to create secure health records that can be shared between doctors and patients.
  - **BurstIQ**: Provides blockchain-based solutions for healthcare data management and secure data sharing.

- **Smart Contracts and Decentralized Applications (dApps)**

Smart contracts automate business processes and enforce agreements without intermediaries. Blockchain platforms enable developers to create dApps that run on decentralized networks.

Example of platforms used in smart contract and decentralization of applications:

- o **Ethereum**: The most widely used blockchain platform for developing dApps and smart contracts.
- o **Solana**: A high-performance blockchain that supports dApps with faster transactions and lower fees.

➢ **Tokenization of Assets**

Tokenization allows physical and digital assets to be represented as blockchain-based tokens, enabling fractional ownership, trading, and liquidity.

Example of Platforms used in Tokenization of Assets:

- o **Securitize**: Specializes in tokenizing real-world assets like real estate, stocks, and bonds.
- o **Polymath**: A platform for creating and managing security tokens, enabling regulatory-compliant tokenization of assets.

➢ **Voting Systems**

Blockchain-based voting systems ensure secure, transparent, and tamper-proof elections.

Example of platforms use Voting systems:

- o **Voatz**: A mobile voting platform that uses blockchain for secure and anonymous voting.
- o **FollowMyVote**: Focuses on blockchain-based voting solutions for transparent and auditable elections.

➢ **Energy Trading**

Blockchain enables peer-to-peer energy trading platforms where individuals can buy and sell renewable energy directly.

Example of Platform used in energy trading:

- o **Power Ledger**: A platform for decentralized energy trading and management using blockchain technology.
- o **WePower**: Uses blockchain to allow consumers to buy and trade renewable energy directly from producers.

➢ **Cross-Border Payments and Remittances**

Blockchain reduces the cost and time required for cross-border payments by eliminating intermediaries and providing faster settlement times.

Example Platform used in Cross-Border Payments and Remittances

- o **Ripple**: Focuses on cross-border payments using blockchain and the XRP token.

- **Stellar**: Provides blockchain solutions for fast, low-cost cross-border payments.

➢ **Data Storage and Sharing**

Decentralized data storage systems on blockchain provide secure and immutable data sharing and storage solutions.

Example of platform used in Data storage and sharing:

- **Filecoin**: A decentralized storage network where users can rent out spare storage capacity.
- **Storj**: Offers encrypted, decentralized cloud storage using blockchain.

✓ **Essential components of wallet (Private Keys, Public Keys, Addresses)**

blockchain wallet is a digital tool used to manage cryptocurrencies or digital assets. It allows users to send, receive, and store these assets securely.Three critical components of a blockchain wallet are **Private Keys**, **Public Keys**, and **Addresses**.

Private Keys

A private key is a randomly generated string of characters used to access and control the digital assets stored in a wallet. It's like a password or a PIN code.

➢ **Function of private keys** :

The private key grants full control over the funds in the wallet. It is used to **sign transactions**, proving the ownership of the assets and authorizing the transfer of funds.

N.B: Private keys must be kept **secret** and **never shared**. If someone gets access to your private key, they can steal all your funds.

Format example of private:

5J3mBbAH58C9pQqPzpTQR2p2MzC6vqLfMRknmD8T5uZdYd5dMZ1

Public Keys

A public key is mathematically derived from the private key. While it's linked to the private key, it cannot be used to reveal the private key.

➢ **Function of public keys**:

- The public key is used to **encrypt** information and verify that a transaction was signed with the corresponding private key. It serves as the wallet's unique identification.
- In any transaction, the public key helps the network verify that the sender's private key was used to sign the transaction, ensuring the sender is the rightful owner.

Format example of public key: 04bfcab9d9bbf5a15d2c6ab604014a7f8a9a...

Addresses

A blockchain address is a shorter, compressed version of the public key. It is a unique identifier where funds can be sent and received.

It functions like an account number in traditional banking.

- **Function**: The address is the **public-facing** part of a wallet. You share your wallet address with others to receive funds.

  N.B: Unlike private keys, addresses can be shared openly. However, all transactions made to and from an address are visible on the blockchain.

  Format Example: Bitcoin address: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa

✓ **Working of Blockchain Transaction**

- **Initiating**: The sender signs the transaction with a private key; the receiver provides their public key.
- **Broadcasting**: The transaction is broadcast to the network and shared with nodes.
- **Validation**: The network validates the transaction using a consensus mechanism (e.g., PoW or PoS).
- **Block Creation**: The transaction is grouped with others in a block and added to the blockchain.
- **Recording**: The transaction is recorded permanently on the blockchain, ensuring transparency and security.
- **Completion**: The transaction is confirmed, and the receiver can access the assets.

Practical Activity 1.1.2: Description of Blockchain Use Cases

Task:

1: You are requested to go in computer lab to apply blockchain use cases.

2: Read the key readings 1.1.2

3: Apply safety precaution

4: Apply blockchain use case

5: Present your work to trainer.

6: Perform the task provided in application of learning 1.1

Key readings 1.1.2: Description of Blockchain Usecases

✓ **The following are different blockchain usecases:**

- **Finance**: Cryptocurrency, decentralized finance (DeFi).
- **Supply Chain**: Transparency and traceability.
- **Healthcare**: Secure management of medical records.
- **Voting**: Secure and tamper-proof elections.

- **Digital Identity**: Decentralized control of personal data.
- **Smart Contracts**: Automated, trustless agreements.
- **Asset Tokenization**: Fractional ownership of real-world assets.
- **Intellectual Property**: Protection and verification of ownership.
- **Energy**: Peer-to-peer energy trading and renewable tracking.
- Gaming/NFTs: Digital ownership of unique assets and collectibles.

✓ **The application of blockchain in KB-Chain's supply chain, here are steps they should follow**:

- **Understand the Use case Requirement**s

➢ **Company Overview:** KB-Chain is an industrial company handling the import of drinks.

➢ **Problem:** The company aims to integrate blockchain into its supply chain to improve transparency, traceability, and security in importing goods.

➢ **Objective:** Trainees need to understand how blockchain can be used in logistics, inventory tracking, and real-time data sharing among suppliers, warehouses, and distributors.

- **Identify Blockchain Concepts to Apply**

➢ **Blockchain Fundamentals:** Explain key concepts such as distributed ledgers, immutability, consensus mechanisms, and smart contracts.

➢ **Supply Chain Integration:** Demonstrate how blockchain offers traceability, enhances data security, and reduces fraud.

➢ **Decentralization:** Highlight how different participants (suppliers, distributors, retailers) access the blockchain, ensuring transparent communication.

➢ **Select a Suitable Blockchain Platform**

➢ **Choose a Platform:** Identify blockchain platforms like Ethereum, Hyperledger, or VeChain that are suited for supply chain management.

➢ **Justify the Choice:** Explain why the selected platform is suitable (e.g., smart contract functionality, business-focused features).

- **Design Blockchain Architecture for Supply Chain**

➢ **Blockchain Layers:** Illustrate how the blockchain technology stack layers (network, consensus, protocol, and application layer) will work together to manage the supply chain.

➢ **Create an Architecture Diagram:** Include blockchain nodes representing suppliers, warehouses, shipping companies, and retailers. Show how data is shared among participants.

➢ **Define Roles:** Identify who will be the participants (nodes) in the network and their roles (validators, users, etc.).

- **Identify Consensus Mechanism**
  - **Select a Consensus Mechanism:** For instance, Proof of Stake (PoS) or Proof of Authority (PoA) could be used for managing transactions within the network.
  - **Explain Mechanism:** Provide a rationale for why the chosen mechanism ensures security and transparency.

- **Develop a Smart Contract for Supply Chain Transactions**
  - **Define Contract Terms:** Explain how smart contracts will automate payment processes, goods handovers, and ensure accountability at each step of the import process.
  - **Key Functions of the Contract:** Demonstrate how to create functions like:
  - **Order Placement:** A supplier creates a record when goods are ordered.
  - **Shipment:** Records when the shipment is dispatched and received.
  - **Payment:** Triggers payment upon verification of delivery.
- **Data Flow and Transaction Management**
  - **Track and Trace:** Show how blockchain can provide real-time tracking of drinks as they move through the supply chain, recording each stage immutably.
  - **Supply Chain Transparency:** Demonstrate how every party can see the transaction history but only authorized nodes can add new data.
- **Simulate or Prototype the Supply Chain on Blockchain**
  - **Prototype on a Test Network:** Have trainees develop a small prototype, deploying the smart contracts on a blockchain testnet (e.g., Ethereum's Ropsten or Hyperledger).
  - **Perform Transactions:** Simulate transactions like ordering drinks, shipping, and payment to test the solution.

Points to Remember

- Blockchain is a decentralized digital ledger that records transactions across multiple computers. Each block in the chain contains a list of transactions, and once recorded, the data cannot be altered retroactively.
- Cryptography plays a critical role in blockchain technology by providing security and integrity.
- **Types of Blockchains are** Public Blockchains, Private Blockchains, Consortium Blockchains, and Hybrid Blockchains.
- **Principles of Blockchain** includes Decentralization, Immutability, and Consensus Mechanisms.
- Blockchain layers are Application Layer , Execution Layer , Consensus Layer, Network Layer ,Data Layer and Infrastructure Layer

- **Solving Industry-Specific Challenges:** Blockchain use cases are most effective when they address specific industry challenges, such as enhancing transparency, ensuring data security, or automating processes through decentralized systems.
- **Decentralization and Trust:** Blockchain eliminates the need for intermediaries by creating a trustless system where all participants can verify and access the same data.
- While applying use cases, first identify the problem and assess if blockchain can provide a suitable solution. Choose the appropriate blockchain platform, design the system architecture, and develop smart contracts.
- Finally, test, deploy, and integrate the solution while ensuring regulatory compliance.

Application of learning 1.1.

KB-Chain is an industrial company handling importing of drinks, currently they want to integrate blockchain into their supply chain. Based on the introduction to blockchain and blockchain key concepts ask the trainees to demonstrate the application of this use case

Indicative content 1.2: Selecting Blockchain Technologies

Duration: 3hrs

Theoretical Activity 1.2.1: Description of Blockchain Technology Stack Principles

Tasks:

1: Answer the following questions:

i. What is blockchain technology stack principles?

ii. Explain the following layers in blockchain technology stack

a. Consensus Layer (e.g., Proof of Work, Proof of Stake)
b. Network Layer (e.g., Ethereum's Peer-to-Peer Network)
c. Protocol Layer (e.g., Ethereum's EVM - Ethereum Virtual Machine)
d. Smart Contracts Layer (e.g., Decentralized Finance (DeFi) Platforms)
e. Application Layer (e.g., CryptoKitties)
f. Storage Layer (e.g., IPFS - InterPlanetary File System)
g. Identity and Access Management (e.g., SelfKey)
h. Security and Encryption (e.g., Public and Private Key Encryption)
i. Interoperability Layer (e.g., Polkadot)
j. Scalability Solutions (e.g., Lightning Network for Bitcoin)
k. Governance Mechanisms (e.g., Tezos)
l. User Interfaces (e.g., MetaMask)

2: Write your findings on paper or flipchart

3: Present your findings to the trainer or colleagues

4: For more clarification read the key readings 1.2.1.

5: In addition, ask questions where necessary.

Key readings 1.2.1.: description of blockchain technology stack plinciples

✓ **Consensus Layer** (e.g., Proof of Work, Proof of Stake)

- **Consensus mechanisms** ensure that all nodes in the blockchain network agree on the validity of transactions.
- In **Proof of Work (PoW)**, miners solve complex puzzles to add blocks.
- In **Proof of Stake (PoS)**, validators are selected based on their stake in the network.
- **Role**: Prevents double-spending, ensures data integrity, and secures the blockchain.

✓ **Network Layer** (e.g., Ethereum's Peer-to-Peer Network)
The network layer facilitates communication between nodes.

- **Blockchain networks** use a decentralized, **peer-to-peer (P2P)** system where nodes communicate directly without a central server.
- Each node stores a copy of the blockchain and shares transaction data with other nodes.Enhances transparency, security, and fault tolerance, as there is no single point of failure.

✓ **Protocol Layer** (e.g., Ethereum's EVM - Ethereum Virtual Machine)

- The **protocol layer** establishes the rules for blockchain operations and ensures compatibility across the network.
- The **Ethereum Virtual Machine (EVM)** is an example, providing an environment where smart contracts can be executed.
- **Role**: Defines the rules for transaction validation, smart contract execution, and security measures.

✓ **Smart Contracts Layer** (e.g., Decentralized Finance (DeFi) Platforms)

- **Smart contracts** are self-executing contracts with terms directly written into code on the blockchain.
- Once conditions are met, the contract executes automatically.
- **Role**: Eliminates the need for intermediaries, reduces costs, increases speed, and ensures trust less execution of transactions.

✓ **Application Layer** (e.g., CryptoKitties)

- The **application layer** provides the interface through which users interact with blockchain via **decentralized apps (dApps)**.
- Examples: Users can buy and trade digital assets on apps like **CryptoKitties**.
- **Role**: Facilitates user access to blockchain functionality, such as financial services, games, and marketplaces, without intermediaries.

✓ **Storage Layer** (e.g., IPFS - InterPlanetary File System)

The storage layer handles how data is stored in a decentralized manner.

- **Decentralized storage** (e.g., IPFS) allows data to be stored across a network of nodes rather than on a central server.
- This ensures that data remains accessible even if some nodes fail.
- **Role**: Enhances data security, availability, and fault tolerance while reducing the risks associated with centralization.

✓ **Identity and Access Management** (e.g., SelfKey)

- **Blockchain-based identity management** allows users to control their own digital identities without relying on central authorities.
- Platforms like **SelfKey** allow individuals to securely manage and share personal data.
- **Role**: Ensures secure and tamper-proof identity verification, giving users control over who accesses their data.

✓ **Security and Encryption** (e.g., Public and Private Key Encryption)

- **Encryption** uses public and private keys to secure transactions and data.
  - **Public keys** are used to encrypt data, while **private keys** decrypt it.
  - Digital signatures verify the sender's identity.
- **Role**: Protects transaction data from unauthorized access and ensures that only intended recipients can access sensitive information.

✓ **Interoperability Layer** (e.g., Polkadot)**.**

- **Interoperability** enables different blockchain platforms to share data and interact.
- Solutions like **Polkadot** and **Cosmos** allow for cross-chain communication and asset transfers.
- **Role**: Increases flexibility by allowing diverse blockchain ecosystems to work together, fostering innovation and integration.

✓ **Scalability Solutions** (e.g., Lightning Network for Bitcoin)

- **Scalability solutions** address the challenges of handling a large volume of transactions efficiently.
- **Lightning Network** is an off-chain scaling solution for Bitcoin that allows for faster, cheaper transactions.
- **Role**: Enhances transaction throughput, reduces costs, and improves the user experience by addressing blockchain's limitations in speed and scalability.

✓ **Governance Mechanisms** (e.g., Tezos)

- **Governance mechanisms** allow stakeholders to make decisions regarding blockchain upgrades, changes, and management.
- **On-chain governance** (e.g., Tezos) involves voting by token holders to determine the future of the blockchain.
- **Role**: Ensures decentralized and democratic decision-making, allowing communities to manage protocol changes transparently.

✓ **User Interfaces** (e.g., MetaMask)**.**

- **User interfaces (UI)**, such as **MetaMask**, simplify interactions with blockchain applications.
- Users can manage digital assets, execute transactions, and interact with dApps through these interfaces.
- **Role**: Makes blockchain technology accessible to non-technical users, improving adoption and user experience.

Theoretical Activity 1.2.2: Description of Types of Consensus Mechanism

Tasks:

1: Answer the following questions :

i. What is consensus mechanism?
ii. Explain the functionality of the following types of consensus mechanism in blockchain:
   a) Proof of Work
   b) Proof of Stake
   c) Delegated Proof of Stake
   d) Proof of Authority
   e) Proof of Weight
iii. Describe other types of consensuses mechanism
iv. Give the blockchain network and their used consensus mechanism.

2: Write your findings on paper or flipchart
3**:** Present your findings to the trainer or colleagues
4: For more clarification read the key readings 1.2.2.
5: In addition, ask questions where necessary.

Key readings 1.2.2.: Description of Types of Consensus Mechanism

✓ **Consensus mechanism**

- A **consensus mechanism** is a fundamental process used in blockchain networks to ensure that all participants (or nodes) agree on the validity of transactions and the state of the blockchain. Since blockchain is decentralized and lacks a central authority, consensus mechanisms are vital for maintaining trust, data integrity, and preventing fraud, such as **double-spending** (when the same asset is spent more than once).

In a blockchain, transactions are grouped into blocks, and these blocks need to be validated and added to the chain. Consensus mechanisms determine how the network agrees on which block is valid and should be added. Different consensus mechanisms use various methods for achieving this agreement, balancing security, efficiency, and decentralization.

✓ **Types of Consensus Mechanisms in Blockchain**

- **Proof of Work (PoW):**

PoW is based on miners competing to solve complex mathematical puzzles. The first one to solve the puzzle gets to add the next block to the blockchain and receives a reward.

➢ **Security**: High, but energy-intensive and requires significant computational power.
➢ **Pros**: Highly secure and decentralized.
➢ **Cons**: Slow transaction processing, high energy consumption, and expensive hardware requirements.

Example Networks: Bitcoin, Litecoin, and Ethereum (before the transition to PoS).

Proof of Stake (PoS):

- **How it works**: Validators are chosen to validate transactions based on the number of coins they hold (stake) and are willing to lock as collateral. Validators earn rewards for correctly validating blocks.
- **Security**: Less energy-intensive compared to PoW, but some argue it could lead to centralization.
- **Pros**: Energy-efficient, scalable, and faster transactions.
- **Cons**: Wealthier participants may have more influence over block validation.
- **Example Networks**: Ethereum 2.0, Cardano, Polkadot.

Delegated Proof of Stake (DPoS):

- Stakeholders vote to elect a small number of delegates who are responsible for validating transactions and maintaining the blockchain. These delegates are rotated periodically based on votes.
- **Security**: Efficient and scalable but can be less decentralized due to the small number of validators.
- **Pros**: Fast, scalable, and reduces transaction costs.
- **Cons**: More prone to centralization since only a few validators are elected.

Example Networks: EOS, TRON, Steemit.

Proof of Authority (PoA):

- Validators are pre-approved and must be trusted to validate blocks. Their authority or reputation is at stake, ensuring they behave honestly.
- **Security**: Fast and energy-efficient but more centralized.
- **Pros**: High efficiency and fast transaction speeds.
- **Cons**: Centralized since only trusted validators can participate.
- **Example Networks**: VeChain, Ethereum’s test networks (Rinkeby, Kovan).

Proof of Weight (PoWeight):

- Validators are selected based on various factors like the amount of stake, storage, or reputation within the network, with the "weight" of their contribution determining their selection.
- **Security**: Can provide both scalability and security depending on the factor used for selection.
- **Pros**: More flexible since multiple factors determine validation power.
- **Cons**: Could potentially lead to centralization depending on the factor chosen.
- **Example Networks**: Algorand (Proof of Stake with weight based on holdings).

✓ **Other Types of Consensus Mechanisms**

Proof of Burn (PoB):

- Participants "burn" or destroy coins to show commitment, allowing them to validate the next block.

- The more coins burned, the greater the chances of adding blocks.
- **Used by**: Slimcoin, Counterparty.
- **Proof of Elapsed Time (PoET)**:
- Participants wait for a randomly selected amount of time before becoming eligible to add a block. The first to finish the wait adds the block.
- **Used by**: Hyperledger Sawtooth.
- **Proof of Space (PoSpace)** or **Proof of Capacity**:
- Validators allocate storage space, and the more space they allocate, the better their chances of being selected to validate blocks.
- **Used by**: Chia, Burstcoin.
- **Proof of History (PoH)**:
- Transactions are timestamped to prove the order of events, allowing for faster consensus and greater throughput.
- **Used by**: Solana.
- **Practical Byzantine Fault Tolerance (PBFT)**:
- A consensus algorithm where nodes work together to reach agreement on the validity of transactions, even if some nodes act maliciously.
- **Used by**: Hyperledger Fabric, Zilliqa.

Practical Activity 1.2.3: use appropriate consensus mechanism

Task:

1: You are requested to go in computer lab to lab to apply blockchain use cases by using appropriate consensus mechanism

2: Read the key readings 1.2.3

3: Apply safety precaution

4: Apply blockchain use case

5: Present your work to trainer.

6: Perform the task provided in application of learning 1.2

Key readings 1.2.3: Use appropriate consensus mechanism

To use an appropriate consensus mechanism in a blockchain system, follow these key steps:

Step 1: Identify the Requirements

- **Transaction Volume:** Determine how many transactions need to be processed per second.
- **Security Needs:** Assess the level of security required, especially for preventing fraud or attacks.
- **Scalability Needs:** Consider future growth and the need to scale the system without performance loss.
- **Decentralization Requirements:** Decide the degree of decentralization, whether the network needs full decentralization or partial trust in some entities.
- **Energy Efficiency:** Evaluate how energy-efficient the consensus mechanism needs to be, especially for eco-friendly solutions.

Step 2: Evaluate Available Consensus Mechanisms

- **Proof of Work (PoW):** Best for security but less efficient and slow.
- **Proof of Stake (PoS):** More efficient and faster, with moderate decentralization.
- **Delegated Proof of Stake (DPoS):** Highly scalable, faster, but involves partial centralization.
- **Proof of Authority (PoA):** High efficiency and security for permissioned blockchains but less decentralized.
- **Proof of Weight (PoWeight):** Can provide custom consensus based on factors such as reputation or stake.

Step 3: Match the Consensus Mechanism to the Use Case

- **For High Security and Low Volume:** Use **Proof of Work (PoW)** for applications requiring the highest security (e.g., Bitcoin).
- **For High Scalability and Efficiency:** Use **Delegated Proof of Stake (DPoS)** or **Proof of Stake (PoS)** to handle large transaction volumes, especially in supply chain or financial applications.
- **For Permissioned Networks:** Use **Proof of Authority (PoA)** where a limited number of trusted validators are sufficient.

Step 4: Design the Network Architecture

✓ Set up the network according to the chosen consensus mechanism:

- **PoW:** Miners need computational power.
- **PoS/DPoS:** Validators need to stake tokens, and delegates are elected.
- **PoA:** Validators are pre-approved entities.

Step 5: Test the System

- Simulate real-world conditions to test the consensus mechanism’s ability to handle transaction volumes, speed, and security needs.

Step 6: Monitor and Adjust

- Continuously monitor the system performance, including transaction speed, security, and energy usage.
- Adjust or upgrade the consensus mechanism if the system's needs change over time.

Theoretical Activity 1.2.4: identify the types of attacks and vulnerabilities of block chain

Tasks:

1: Answer the following questions:

i.Explain the following attacks of blockchain technology:

a. Attack in consensus mechanism
b. Sybil Attack (spamming the network, disrupt communication among nodes)
c. Double Spending
d. Eclipse Attack
e. Smart Contract Vulnerabilities (re-entrancy attacks, integer overflow/underflow).
f. DDoS Attack (Distributed Denial of Service)
g. Blockchain Spamming
h. Long-Range Attack
i. Selfish Mining
j. Routing Attacks
k. Transaction Malleability
l. Consensus Manipulation

2: Write your findings on paper or flipchart
3**:** Present your findings to the trainer or colleagues
4: For more clarification read the key readings 1.2.4.
5: In addition, ask questions where necessary.

Key readings 1.2.4: Identify the types of attacks and vulnerabilities of blockchain

- ✓ **Attack in Consensus Mechanism**:
- Manipulation of the consensus process, like the **51% attack**, where a malicious entity control most of the mining power (PoW) or stakes (PoS) to validate fraudulent transactions or disrupt the network.
- ✓ **Sybil Attack**:
- The attacker creates multiple fake nodes or identities to gain influence over the network, which can disrupt communication and spam the network, slowing down operations and creating vulnerabilities.
- ✓ **Double Spending**:
- A scenario where the same cryptocurrency token is spent more than once, usually by reversing a transaction after it has been confirmed, undermining transaction integrity.
- ✓ **Eclipse Attack**:
- A specific attack on peer-to-peer networks where a malicious entity isolates a node by overwhelming it with fake connections, thus controlling the node's view of the blockchain and feeding it false information.
- ✓ **Smart Contract Vulnerabilities**:
- **Exploits in smart contract code, such as:**
- ➢ **Re-entrancy Attacks**: Exploiting recursive calls to drain funds.
- ➢ **Integer Overflow/Underflow**: Manipulating contract logic by exceeding or lowering numerical limits.
- ✓ **DDoS Attack (Distributed Denial of Service)**:
  Flooding the blockchain network or specific smart contracts with a massive number of requests, causing a slowdown or complete crash of the network's functionality.
- ✓ **Blockchain Spamming**:
  Spamming the blockchain with a large number of low-value transactions to congest the network, increase transaction fees, or delay legitimate transactions.
- ✓ **Long-Range Attack**:
  Common in Proof of Stake (PoS) systems, where an attacker creates a long alternate history of the blockchain, potentially invalidating the current chain.
- ✓ **Selfish Mining**:
  A strategy where miners withhold newly found blocks to form a private chain, later broadcasting it to gain a competitive advantage, disrupting the blockchain's fairness.

✓ **Routing Attacks**:
Intercepting and tampering with the flow of data between blockchain nodes, which can delay or manipulate transactions or cause discrepancies in the ledger.

✓ **Transaction Malleability**:
Modifying the transaction's unique identifier (hash) before it's confirmed, allowing an attacker to alter transaction details and potentially deceive systems tracking the transactions.

✓ **Consensus Manipulation**:
Interfering with the consensus process to validate incorrect or malicious transactions, undermining the integrity of the entire blockchain.
These attacks highlight the potential vulnerabilities within blockchain systems that need to be addressed to maintain security, reliability, and decentralization.

Points to Remember

- **Blockchain Technology Stack Principles** refer to the layered structure that organizes the various components and functionalities of blockchain technology.
- **Blockchain Stack Layers** includes Consensus Layer, Network Layer, Protocol Layer, Smart Contracts Layer, Application Layer, Storage Layer, Identity and Access Management, Security and Encryption, Interoperability Layer, callability Solutions, Governance Mechanisms and User Interfaces
- **consensus mechanism** Ensures that the distributed network of nodes reaches an agreement on the correct version of the blockchain.
- **Functionality**: Protects against malicious actors and ensures that only valid transactions are recorded.
- **Types**: Includes Proof of Work (PoW), Proof of Stake (PoS), Delegated Proof of Stake (DPoS), Proof of Authority (PoA), among others.
- Blockchain technology faces several attacks and vulnerabilities that threaten its security and integrity. **Consensus mechanism attacks**, such as the 51% attack, can manipulate the validation of transactions, while **Sybil attacks** flood the network with fake nodes to disrupt communication.
- **Double spending** undermines trust by allowing the same asset to be spent more than once, and **Eclipse attacks** isolate a node with false data, distorting its view of the blockchain.
- Additionally, **smart contract vulnerabilities**, like re-entrancy and integer overflow, can lead to unauthorized contract execution, while **DDoS attacks** and **blockchain spamming** overwhelm the network, slowing down operations.

- **Selfish mining**, **routing attacks**, and **transaction malleability** exploit weaknesses in mining and data transmission, while **consensus manipulation** alters the consensus process, validating malicious transactions.

Application of learning 1.2

A large industrial company, KB-Chain, handles the import of drinks from multiple suppliers across different countries. They want to integrate blockchain into their supply chain management system to improve traceability, security, and efficiency. You are requested to select the appropriate consensus mechanism they can use.The system will need to handle a high volume of transactions while ensuring transparency, data integrity, and resistance to fraud.

Indicative content 1.3: Design the Architecture of Blockchain Application

Duration: 4hrs

Theoretical Activity 1.3.1: Description of blockchain architecture

Tasks:

1: Answer the following questions:

i. What is the architecture of a blockchain?
ii. What are the key components of blockchain architecture?
iii. How are components of a blockchain connected?
iv. What is the relationship between instances in a blockchain system?

2: Write your findings on paper or flipchart

3: Present your findings to the trainer or colleagues

4: For more clarification read the key readings 1.3.1.

5: In addition, ask questions where necessary.

Key readings 1.3.1.:description of blockchain architecture

✓ **Description of Blockchain Architecture**

Blockchain architecture refers to the underlying structure of blockchain technology, which is a decentralized, distributed ledger system. This architecture is designed to ensure transparency, security, and immutability by linking blocks of data in a chain. Each block contains a list of transactions and is cryptographically secured. The entire system operates without a central authority, relying on consensus mechanisms to validate and record transactions across multiple nodes.

✓ **Components of Blockchain Architecture**

The blockchain system consists of several core components that work together to ensure its functionality:

- **Blocks**: The fundamental unit of the blockchain, containing a list of transactions, a timestamp, and a cryptographic hash linking it to the previous block.
- **Nodes**: Independent computers that store a copy of the blockchain and participate in the network's operation.
- **Consensus Mechanism**: The process used to validate transactions and ensure agreement across the decentralized network (e.g., Proof of Work, Proof of Stake).
- **Smart Contracts**: Self-executing contracts with terms directly written into code, facilitating automated transactions.

- **Cryptography**: Ensures the security of transactions using encryption methods like public and private keys.
- ✓ **Connection in Blockchain**

  In blockchain, **nodes** (computers or devices) connect through a **peer-to-peer (P2P) network**. Each node communicates directly with others, sharing and validating transaction data without relying on a central server. This decentralized structure enhances the system's resilience and prevents a single point of failure. Each node holds an identical copy of the blockchain, ensuring transparency and trust within the network.
- ✓ **Instance Relation in Blockchain**

  The **instance relation** in blockchain refers to how different entities (nodes, miners, validators) interact within the system.
- **Miners** (in Proof of Work) or **Validators** (in Proof of Stake) validate transactions and propose new blocks to be added to the blockchain.
- Once a transaction is validated by consensus, it is propagated across the network, and all participating nodes update their local copies of the blockchain.
- The process is decentralized, meaning no single entity controls or manages the system, ensuring transparency, security, and consensus-driven decision-making.

Practical Activity 1.3.2: Designing system architecture

Task:

1: You are requested to go in computer lab to lab to design blockchain system architecture
2: Read the key readings 1.3.2
3: Apply safety precaution
4: design the blockchain system architecture
5: Present your work to trainer.
6: perform the task provided in application of learning 1.3

Key readings 1.3.2: Design blockchain system architecture

Designing a blockchain system architecture using EdrawMax involves a series of steps to help you create a comprehensive visual representation of the blockchain's components and interactions.

- ✓ **Here's a step-by-step guide:**

  Step 1: Open EdrawMax and Set Up Workspace

- **Launch EdrawMax**: Open the software and choose "New" from the main interface.
- **Select a Template**: Choose a blank drawing canvas or a suitable template for system architecture under the "Network" or "Software" diagram category.
- **Set Grid and Layout**: Set the grid to guide your alignment and layout. Go to the "View" tab and enable "Gridlines" for precise positioning.

Step 2: Identify the Layers and Components of Blockchain Architecture

Before you start drawing, identify the main layers and components of your blockchain system:

- **Consensus Layer**: Defines the consensus mechanisms (e.g., Proof of Work, Proof of Stake).
- **Network Layer**: Nodes, peer-to-peer communication protocols, and networking.
- **Protocol Layer**: Governs the rules for transaction processing and validation.
- **Smart Contracts Layer**: Where decentralized applications and logic reside.
- **Application Layer**: User-facing interfaces and tools.
- **Storage Layer**: Data storage solutions, including on-chain and off-chain storage.

Step 3: Draw the Base Structure of the Blockchain

- **Insert the Main Blocks**:
  - Go to the "Insert" tab and start by inserting **basic shapes** (rectangles or blocks) to represent each of the layers (consensus, network, protocol, etc.).
  - Label each block appropriately.
- **Add Sub-components**:
  - Inside each block, add smaller blocks to represent the various sub-components, such as nodes, smart contracts, and APIs.
  - Use rectangles or other shapes from the "Flowchart" or "Network Diagram" sections in EdrawMax to represent each sub-component.

Step 4: Connect the Components

- **Use Connectors**:
  - Connect each component using arrows or lines to indicate the flow of data, messages, or transactions between them.
  - Go to "Connector" in the "Home" tab to add lines.
- **Specify Relationships**:
  - Label the connectors to specify the types of interactions (e.g., data transfer, smart contract execution, consensus mechanism communication).

Step 5: Add Nodes, Smart Contracts, and Consensus Details

- **Add Nodes**:
  - Draw nodes within the **Network Layer** to represent various blockchain participants.

- Use shapes like circles or squares from the **Network Diagram** section to represent different node types (full nodes, light nodes, etc.).

- **Smart Contracts**:
- In the **Smart Contracts Layer**, add components for smart contracts and link them with the **Application Layer** where the front-end interaction occurs.

- **Consensus Mechanism**:

- In the **Consensus Layer**, detail the consensus mechanism by showing how nodes validate transactions (e.g., use icons to represent proof-of-work or proof-of-stake validators).

Step 6: Add Interactions and Data Flow

- **Transactions**:
- Show how transactions move from the application layer through the network to the consensus layer for validation.
- Use directional arrows to indicate transaction flow and validation.

- **Data Storage**:
- Depict how data is stored and referenced across the storage layer (both on-chain and off-chain).
- Use shapes like cylinders to represent databases or ledgers.

Step 7: Style and Customize the Diagram

- **Customize Colors**:
  - Apply different colors for each layer or component to visually distinguish them. Select a block, and under the "Format" tab, choose "Fill" to change colors.
- **Text and Labels**:

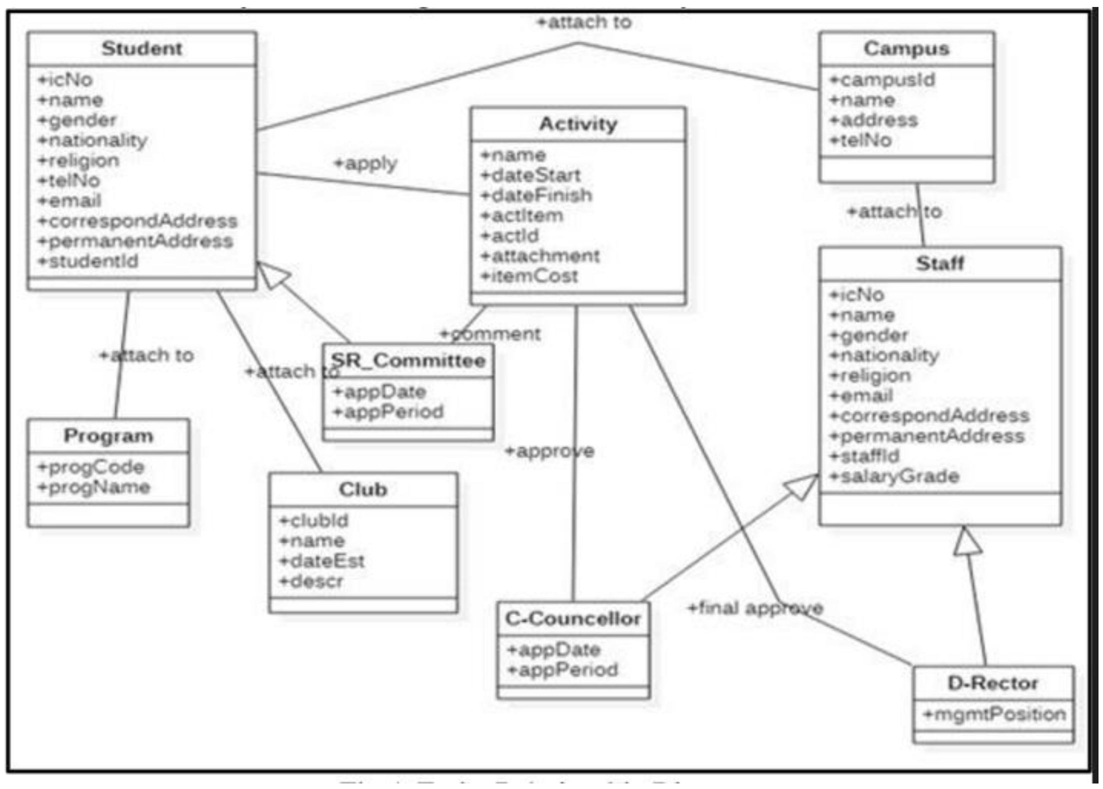
  - Add labels to each component, connector, and layer using the "Text" tool.
  - Make sure each element is clearly labeled for easy understanding.

Step 8: Review and Finalize the Design

- **Verify Completeness**:
  - Review the entire diagram for completeness, ensuring that all components, interactions, and relationships are clearly represented.
- **Save and Export**:

  - Save your project as an EdrawMax file to keep it editable.
  - Export the diagram as a PDF, PNG, or another preferred format for presentation or sharing.

Example: Campus Management System Architecture



Practical Activity 1.3.3: Draw blockchain system architecture

Task:

1: You are requested to go in computer lab to lab to design blockchain system architecture

2: Read the key readings 1.3.3

3: Apply safety precaution

4: design the blockchain system architecture

5: Present your work to trainer.

6: perform the task provided in application of learning 1.3

Key readings 1.3.3: Draw Blockchain system Architecture

Step-by-Step Guide to Drawing Blockchain System Architecture in EdrawMax Based on a Case Study:

Step 1: **Understand the Case Study**

- **Analyze the Case Study**: Review the details of the case study to understand the specific use case, industry, and key requirements (e.g., supply chain management, voting system, or financial platform).

- **Identify Key Components**: Extract essential blockchain components, such as the type of blockchain platform (e.g., Ethereum, Hyperledger), the consensus mechanism, smart contract requirements, nodes, and interactions with external systems (e.g., APIs, databases).

Step 2: **Set Up EdrawMax Workspace**

- **Launch EdrawMax** and choose a **blank drawing canvas** or start from a template under the **Network** or **Software** categories.
- **Create Layers**: Use a layered approach, beginning with key blockchain architecture layers (Consensus Layer, Network Layer, Protocol Layer, Smart Contract Layer, etc.).

Step 3: **Draw the High-Level Architecture**

- **Add Core Blockchain Layers**:
- Use **rectangles** or **blocks** to represent each key layer of the blockchain system:
  - **Consensus Layer**: Select and add a block to represent the consensus mechanism (Proof of Stake, Proof of Work, etc.).
  - **Network Layer**: Add nodes for peer-to-peer interactions.
  - **Protocol Layer**: Define transaction rules.
  - **Smart Contract Layer**: Depict where smart contracts reside and execute logic.
- **Customize Block Names**: Label each block appropriately based on your case study, like “Validator Nodes,” “Smart Contracts,” “API Interface,” etc.

Step 4: **Add Case-Specific Components**

- **External Systems**:
- If the blockchain interacts with external systems (e.g., a supply chain management system or customer interface), use **icons** like databases or APIs.
- **Nodes and Peers**:
- Add **nodes** for full, light, or validator nodes, as applicable.
- Use **circles** or **rectangles** from the **Network Diagram** section to represent different node types.
- **Smart Contracts**:
- Draw blocks within the **Smart Contract Layer** representing the key business logic in the form of smart contracts.
- Link these smart contracts to external inputs (e.g., transactions from users or external data sources like oracles).

Step 5: **Establish Data Flow and Interaction**

- **Connect Nodes**: Use **arrows or connectors** to show the flow of information, transactions, or messages between different layers and components.
- **Label Connections**: Clearly label each arrow to show data flow (e.g., "Transaction Submission," "Validation," "Smart Contract Execution").

Step 6: **Incorporate Advanced Features**

- **Add Consensus Details**: In the **Consensus Layer**, add blocks to describe how consensus is achieved in your system (e.g., how validators work or how mining operates).

- **External APIs**: If your case study involves integration with external systems, illustrate how these systems interact with the blockchain using connectors and labels for API calls.

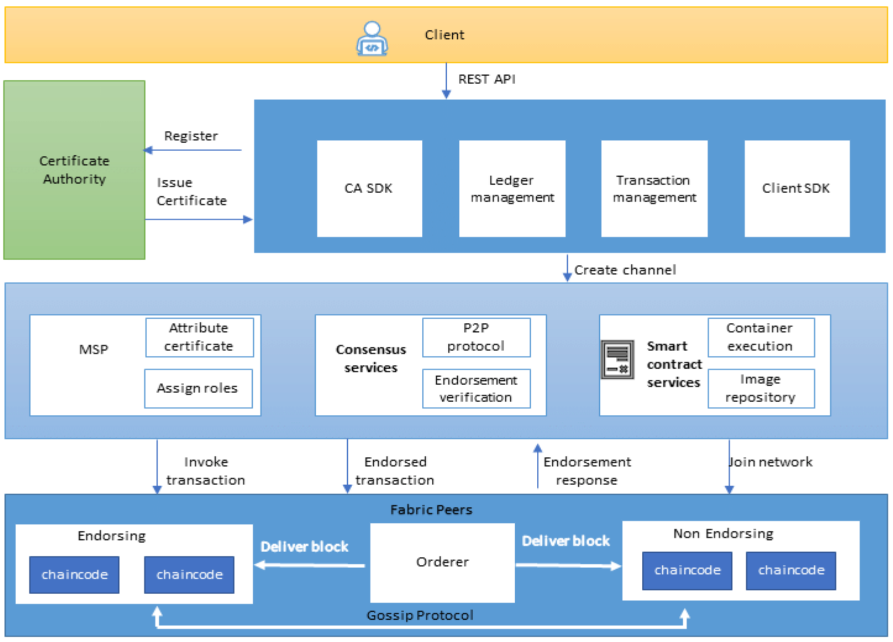
Step 7: **Customize the Appearance**

- **Apply Colors**: Use different colors for various layers and components to improve the readability of the architecture. Go to the **Format** tab to change the fill and stroke color of the blocks.
- **Text and Labels**: Add detailed text and descriptions for each layer, component, and flow to make the diagram clear for others to understand.

Step 8: **Final Review**

- **Check Completeness**: Ensure all components from the case study are represented, and the flow of transactions or data is correctly mapped.

Example: Industrial Management Application – Blockchain Architecture



Points to Remember

- **System Architecture**: Plan interactions between blockchain components and ensure scalability.
- **Blockchain-Based Systems**: Tailor the design to specific use cases, selecting the right platform and consensus.

- **Blockchain Network**: Decide on the network type (public/private), node structure, and security measures.
- **Smart Contracts**: Develop self-executing contracts with secure and efficient code to automate processes.

Application of learning 1.3.

Scenario: A global logistics company, TransChain, specializes in delivering goods from suppliers to retailers across multiple countries. Recently, the company has been facing challenges related to traceability, security, and efficiency in managing its complex supply chain. TransChain wants to integrate blockchain technology to solve these issues, ensuring transparency, data integrity, and improved operational efficiency.As a blockchain expert, you are tasked with designing and drawing a blockchain solution for TransChain.
Your goal is to develop a decentralized system that leverages blockchain functionalities to optimize supply chain operations, including secure tracking, efficient communication between stakeholders, and fraud resistance. Design your blockchain architecture using EdrawMax or similar tools.

Learning outcome 1 end assessment

Theoretical assessment

Q1: Read carefully the following statements related to designing blockchain architecture and Answer by **True** if the statement is correct or by **False** if the statement is incorrect:

i. The Ethereum Virtual Machine (EVM) is responsible for executing smart contracts in the Ethereum network**.**

ii. Blockchain transactions are stored in a linear order in blocks. **.**

iii. Public keys are used to encrypt transactions, while private keys are used to decrypt them. **.**

iv. Blockchain technology only operates in a centralized environment**.**

v. A Sybil attack involves a malicious actor creating multiple fake identities to disrupt the network.

Q2: Circle the letter corresponding to the correct answer.

i. **Which of the following is a core principle of blockchain?**
   a) Centralization
   b) Immutability
   c) Inflation
   d) Secrecy

ii. **Which of the following describes the purpose of the Merkle Tree in blockchain?**
   a) It ensures the immutability of blockchain data.
   b) It links blocks together in the blockchain.
   c) It compresses transaction data for faster verification.
   d) It encrypts private key information.

iii. **Which of these is a vulnerability in smart contracts?**
   a) Routing Attack
   b) Reentrancy Attack
   c) Selfish Mining
   d) Eclipse Attack

Q2. Fill the blank space with the right word based blockchain concepts:

i. A _______________ is a cryptographic data structure used to verify transactions efficiently in blockchain systems**.**
   (Block / Merkle Tree / Private Key)

ii. In a blockchain, the _______________ is used to verify the ownership of an address, while the _______________ is used to sign transactions**.**
   (Public Key / Private Key / Smart Contract)

iii. The _______________ consensus mechanism is energy-efficient and is used in Ethereum 2.0**.**
   (Proof of Work / **Proof of Stake** / Proof of Authority)

iv. A _______________ attack occurs when a malicious node monopolizes communication between a targeted node and the rest of the network.

(Eclipse / Sybil / Selfish Mining)

v. In blockchain, the _______________ ensures the chronological and linear order of blocks.

(Private Key / Consensus Mechanism / Encryption)

vi. A smart contract is a _______________ stored on the blockchain that executes automatically when predefined conditions are met.

(Public Key / Self-executing code / Blockchain)

vii. In a Hierarchical Deterministic Wallet, _______________ are used to generate a sequence of keys.

(Consensus Mechanism / Mnemonic Seeds / Smart Contracts)

Q3: Match the Blockchain Consensus Mechanism (**Column B**) with their corresponding key characteristics (**Column C**). Write the letter of the correct answer in the provided blank space in **Column A.**

Column A	**Column B**	**Column C**
Answer	**Consensus Mechanism**	**Key Characteristic**
1……	a) Proof of Work (PoW)	1. .Relies on a voting system where selected nodes validate blocks.
2……..	b) Proof of Stake (PoS)	2. Miners solve computational puzzles to validate blocks.
3……..	c) Delegated Proof of Stake	3. . Validators are chosen based on the amount of stake they hold.
4……….	d) Proof of Authority (PoA)	4. Validators are chosen based on reputation and identity.
		5. .Uses trusted execution environments to randomly assign time slots for nodes to validate transactions.
		6. Selects validators based on their activity and contribution to the network, such as transaction volume and network support.

Q4. What are the advantages and disadvantages of using public blockchains over private blockchains?

Q5: Given the following use case (e.g., a decentralized voting system), identify the most

Q6. Explain the role of Ethereum's Peer-to-Peer (P2P) network in ensuring decentralization and security in the blockchain.

Q7. How does the InterPlanetary File System (IPFS) improve the efficiency and scalability of data storage on blockchain systems?

Practical assessment

Draw and explain the architecture of a blockchain-based supply chain system that integrates with third-party services like payment gateways. Show the flow of goods, payment, and data in the system.

END

References

A. Sohel Rana, e. a. (2019). Secure Smart Contract Development: A Survey.

Alexander, C. (2019). *Ethereum Cookbook.* O'Reilly Media.

Antonopoulos, A. M. (2018). *Mastering Blockchain Programming: Build, Deploy, and Manage Your Own Blockchain Applications.* O'Reilly Media.

Casey, M. J. (2017). Solidity Programming: A Beginner's Guide.

ConsenSys. (n.d.). Best Practices for Writing Secure Solidity Contracts.

ConsenSys. (n.d.). Optimizing Solidity Contracts for Gas Efficiency.

Egorov, B. Š. (2021). *Solidity Programming Cookbook.* Packt Publishing.

Gavin Wood, e. a. (2014). Solidity: A Programming Language for the Ethereum Blockchain.

Lewis, A. (2017). *Blockchain Basics: A Non-Technical Introduction.* Wiley.

Raggio, G. (2021). *Blockchain Development with Solidity and Ethereum.* Packt Publishing.

Škvorc, B. (2019). Solidity Programming: A Developer's Guide. *Packt Publishing*.

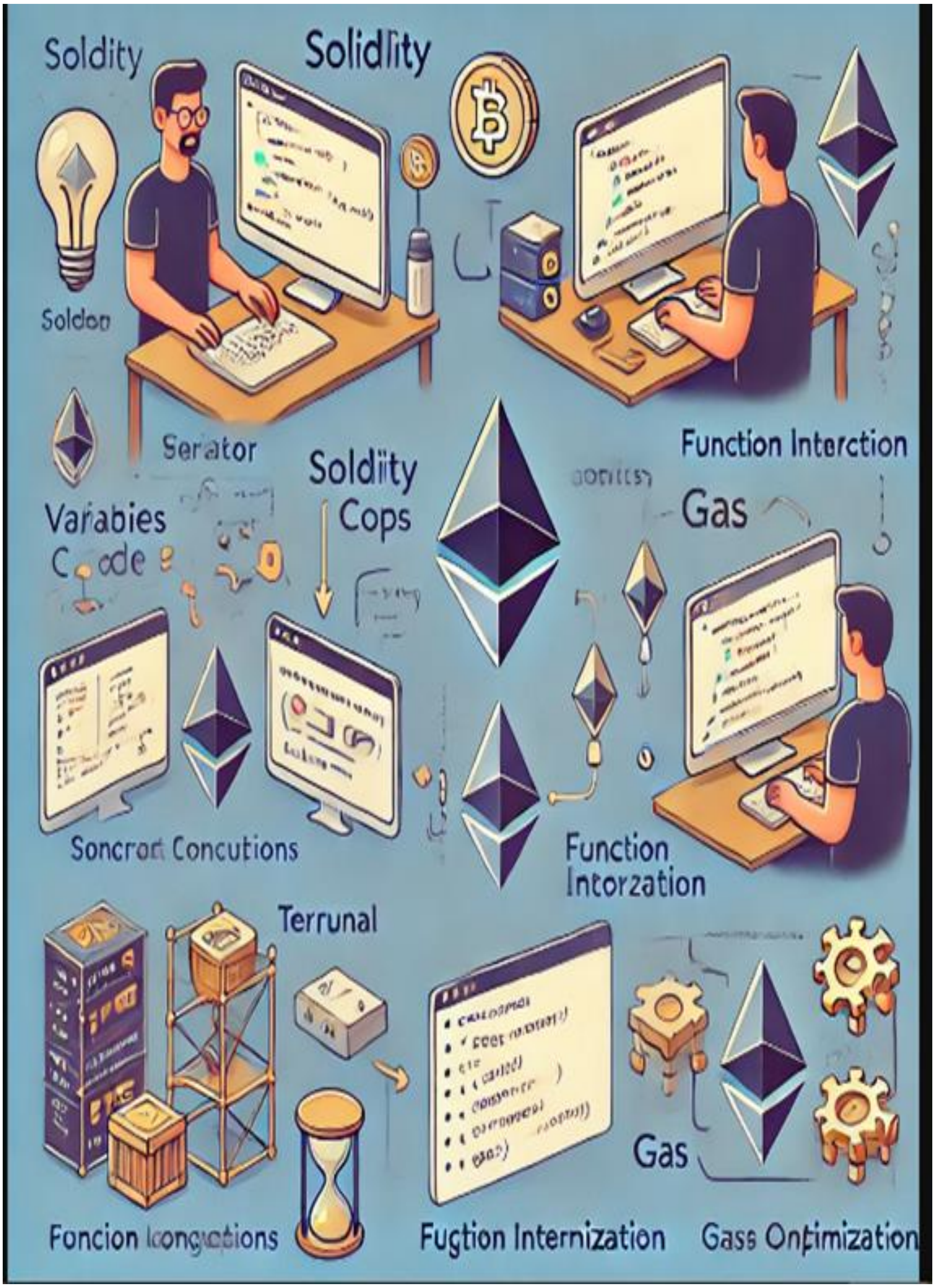
Team, C. (n.d.). *CryptoZombies.* CryptoZombies.

Team, R. I. (n.d.). *Remix IDE.*

Vitalik Buterin, e. a. (2014). Solidity: A Language for Secure Smart Contracts.

Vitalik Buterin, e. a. (2014). Solidity: A Language for Secure Smart Contracts.

Wood, A. M. (2019). *Ethereum Programming: The Definitive Guide.* O'Reilly Media.


Soldity
Solidlity
Soldon
Serator
Variabies
C ode
Soldiity
Cops
Function Interction
Gas
Soncrod Concutions
Function
Intorzation
Terrunal
Foncion longications
Fugtion Internization
Gas
Gass Onfimization

Indicative Contents
2.1 Preparation of Environment
2.2 Applying Solidity Concepts
2.3 Implementing Function Interaction
2.4 Optimizing Gas Costs

Key Competencies for Learning Outcome 2: Apply Solidity Basics

Knowledge	Skills	Attitudes
● Description of key terms ● Elaboration of Storage ● Optimization of Memory cost ● Explanation of Ethereum virtual Machine(EVM) ● Description of Blockchain Explorer ● Explanation of function operations	● Installing Code editor ● Installing node.js and npm. ● Installing Solidity compiler (solc) and Ethereum development tools ● Applying solidity concepts ● Implementing function Interaction ● Calculating the cost of Ethereum transfer ● Setting up solidity environment ● Optimizing Gas Costs	● Being Curiosity and adaptability when selecting Code editor and Installing it ● Being Patience in debugging and paying attention to versioning while you are Installing node.js ● Being problem solver ● Being Analytical thinker and persistence

Duration:20 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Install Code editor accurately based on system requirement
2. Apply properly solidity concepts based on solidity principles
3. Implement correctly function Interactions based on blockchain technology
4. Set up properly solidity environment based on development tools standards
5. Optimize efficiently Gas Costs based on function definition
6. Describe clearly key terms based on blockchain protocol
7. Explain clearly Ethereum virtual Machine(EVM) based on Ethereum Standards
8. Describe properly Blockchain Explorer based on blockchain protocol

Resources

Equipment	**Tools**	**Materials**
● Computer	● VSCode ● Remix, Truffle, and Hardhat ● Node.js,Web3.js and ether.js ● Browser	● Internet ● Electricity

Indicative content 2.1: Preparation of Environment

Duration: 5 hrs

Theoretical Activity 2.1.1: Description of key terms

Tasks:

1: Answer the following questions:

I. What is the basic structure of a Solidity contract?
II. How are arrays declared and used in Solidity?
III. What is the difference between a state variable and a local variable?
IV. What are the rules for naming identifiers in Solidity?
V. How can you access elements in an array in Solidity?
VI. What is a struct in Solidity?
VII. What is the difference between a public function and a private function?
VIII. What are the different control structures available in Solidity?
IX. What is the difference between a public state variable and a private state variable?
X. What are modifiers in Solidity?
XI. What is a smart contract?

2: Write your findings on paper or flipchart

3: Present your findings to the trainer or colleagues

4: For more clarification read the key readings 2.1.1.

5: In addition, ask questions where necessary.

key readings 2.1.1.: Description of key terms

- ✓ **Solidity:** is a high-level programming language specifically designed for developing smart contracts on the Ethereum blockchain.
- ✓ **Basic Structure:** A Solidity contract typically consists of a contract declaration, variable declarations, function definitions, and modifiers.
- ✓ **Comments:** Use // for single-line comments and /* ... */ for multi-line comments.
- ✓ **Variable Declaration:** Declare variables using the **var** keyword or specifying a specific data type (e.g., **uint, address**).
- ✓ **Data Types:**
- **Primitive Data Types: uint, int, address, bool, bytes, string, bytes32, fixed, ufixed**.

- ✓ **Arrays:** Declare arrays using **[]** syntax (e.g., **uint[] myArray**).
- ✓ **Structs:** Define custom data structures using the **struct** keyword.
- ✓ **Variables:**
- **Scope:** Variables can have global (contract-level) or local (function-level) scope.
- **Initialization:** Variables can be initialized during declaration or later using assignment.
- **State Variables:** State variables are stored on the blockchain and persist between function calls.

- ✓ **Identifiers:**
- **Naming Rules:** Identifiers must start with a lowercase letter or underscore and can contain letters, numbers, and underscores.
- Keywords cannot be used as identifiers.
- ✓ **Arrays:**
- **Access Elements:** Use index notation (e.g., **myArray[0])** to access elements.
- **Dynamic vs. Fixed-Size:** Dynamic arrays can change size, while fixed-size arrays have a predetermined length.
- **Iteration:** Use loops (e.g., **for, while**) to iterate over arrays.
- ✓ **Structs:**
- Create custom data types with multiple fields.
- Instantiate structs and access their fields using dot notation.
- Structs can be nested within other structs.
- ✓ **Functions:**
- **Declaration:** Use the **function** keyword followed by the function name, parameters, and return type.
- **Visibility:** Public functions can be called from anywhere, while private functions can only be called within the contract.
- **Arguments:** Pass arguments to functions using parentheses.
- ✓ **Control Structures:**
- **Conditional Statements:** Use **if**, **else**, and **else if** for decision-making.
- **Loops:** Use **for** and **while** loops for repetitive tasks.
- ✓ **State Variables:**
- State variables are stored on the blockchain and persist between function calls.
- Public state variables can be accessed from outside the contract, while private state variables are only accessible within the contract.
- ✓ **Modifiers:**
- Use modifiers to apply conditions or restrictions to functions.
- ➢ **Common Modifiers:**
  - view,
  - pure,

- payable.

✓ **Smart Contracts**

- Self-executing contracts with terms directly written into code.
- Deploy smart contracts to the Ethereum blockchain using a transaction.
- **Advantages:**
  - Automation, transparency, security, and immutability.

✓ **Ethereum:**

- **Ethereum isA** decentralized platform for building and running decentralized applications (dApps).
- **EVM:** The Ethereum Virtual Machine executes smart contracts on the network.
- **Transactions:** Transactions are used to interact with smart contracts and transfer value.

✓ **Ethereum Virtual Machine (EVM)**

- **Execution Environment:** Provides a sandboxed environment for executing Solidity code.
- **Gas:** A unit of computational power used to measure the cost of executing transactions and smart contracts.
- **Stack-Based Architecture:** The EVM operates on a stack-based architecture, where data is pushed onto and popped from a stack during execution.

Practical Activity 2.1.2: Setting up solidity environment

Task:

1: You are requested to go in computer lab to demonstrate setting up a Solidity environment.

2: Read the key readings 2.1.2

3: Apply safety precaution

4: Demonstrate setting up a Solidity environment

5: Present your work to trainer.

6: Perform the task provided in application of learning 2.1

Key readings 2.1.2. Setting Up Solidity Environment

This guide provides step-by-step instructions to help you set up a Solidity development environment. The goal is to install all the necessary tools for writing and testing Solidity smart contracts. Follow each section carefully to complete the setup successfully.

✓ **Installing a Code Editor**

- **Remix IDE (Online)**

  Remix is an online Integrated Development Environment (IDE) specifically for Solidity smart contract development. It allows you to write, compile, and deploy smart contracts directly from your browser without needing any installation.

  Steps to Access Remix:

- ➢ Open your web browser (e.g., Chrome, Firefox).
- ➢ Go to the following URL: **https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa7a.js)**
- ➢ Remix will load in your browser, and you can begin writing Solidity contracts immediately.
- ➢ You don't need to install any extensions or software—Remix handles everything within the browser.

  Benefits of Remix:

- ➢ No installation required.
- ➢ Pre-integrated with the Solidity compiler.
- ➢ Suitable for beginners and for small-to-medium projects.

- **Visual Studio Code (Local)**

  Visual Studio Code (VS Code) is a powerful code editor that requires installation on your computer. It supports extensions, which are useful for Solidity development.

- o **Steps to Install Visual Studio Code:**
- o Download VS Code from the official site:_Visual Studio Code Download.(https://code.visualstudio.com/download)
- o Install it following the instructions for your operating system (Windows, Mac, or Linux).
- o **Install Visual Studio Code on Your System**
- o **For Windows**:
  - ▪ Once the download is complete, open the downloaded file (**VSCodeUserSetup.exe**).
  - ▪ The installation wizard will open. Click **Next**.
  - ▪ Accept the terms and conditions, then click **Next**.
  - ▪ Choose the installation location or leave it as the default, then click **Next**.
  - ▪ Select any additional tasks you want (e.g., creating a desktop icon, adding VS Code to the system PATH), then click **Next**.
  - ▪ Click **Install** to begin the installation.
  - ▪ After the installation completes, click **Finish** to launch VS Code.
  - ▪ After installation, open VS Code.
- o **Installing the Solidity Extension:**

- Open **VS Code** and click on the **Extensions** icon on the sidebar (or press Ctrl+Shift+X),or click on **view menu then Extaensions**
- Search for the **Solidity** extension.
- Click **Install**.
- This extension will enable Solidity syntax highlighting, linting, and compiling within VS Code.
- **Benefits of VS Code:**
- Suitable for larger projects.
- Integrates well with other development tools like Truffle and Hardhat.
- Allows debugging and advanced testing setups.
- Installing Node.js and npm (Node Package Manager)

  Node.js is a JavaScript runtime environment, and **npm** (Node Package Manager) is used to manage the dependencies and libraries required for developing Solidity smart contracts.

  Steps to Install Node.js and npm:
- Go to the official Node.js website: **Node.js Download**.or(**https://nodejs.org/en**)
- Download the **LTS (Long-Term Support)** version for your operating system.
- Follow the installation instructions for your OS.

  Steps to Install Node.js

  Once you've downloaded Node.js, follow these steps to complete the installation process. The steps vary slightly depending on your operating system.
- **For Windows:**
- **Run the Installer:**
  - Locate the downloaded Node.js installer (.msi file) in your Downloads folder or the location where you saved it.
  - Double-click the installer to start the installation process.
- **Follow the Installation Wizard:**
  - **Welcome Screen**: Click **Next**.
  - **License Agreement**: Read and accept the license agreement, then click **Next**.
  - **Choose Installation Location**: Choose the destination folder where Node.js will be installed or leave the default location. Click **Next**.
  - **Select Components**: Ensure that the **Node.js runtime**, **npm package manager**, and **Add to PATH** options are selected. Click **Next**.
  - **Install**: Click **Install** to begin the installation process.
  - **Finish**: Once the installation completes, click **Finish** to exit the installer.
  - After installation, verify Node.js and npm are installed by running the following commands in your terminal:

```
node –v
npm -v
```

This should return the versions of Node.js and npm, respectively.

Why Node.js and npm are Important:

- **Node.js** enables running JavaScript code outside of a browser, which is crucial for blockchain development tools.
- **npm** is essential for installing development libraries like Truffle, Hardhat, and solc (Solidity compiler).

○ **Installing Solidity Compiler (solc)**

The **Solidity compiler (solc)** compiles Solidity source code into bytecode that can be executed on the Ethereum Virtual Machine (EVM).

○ **Installing solc Using npm:**

▪ Open your terminal (Command Prompt, PowerShell, or Terminal on Mac/Linux).

▪ Run the following command to install the Solidity compiler globally:

○ **Verify the installation by checking the version:**

npm solc –version

This will show the installed version of the Solidity compiler.

○ **Alternative: Installing solc Locally**

If you prefer to install **solc** only for a specific project:

▪ Navigate to the project directory in your terminal:

cd /path/to/project

▪ Run the following command to install **solc** locally:

npm install solc

Why solc is Important:

- **solc** translates Solidity code into bytecode, allowing it to be deployed to an Ethereum blockchain.
- It is required for testing and deployment of smart contracts.

○ **Installing Ethereum Development Tools: Truffle and Hardhat**

Both **Truffle** and **Hardhat** are popular frameworks for developing, testing, and deploying smart contracts. They provide a suite of tools that make blockchain development easier and more efficient.

✓ **Installing Truffle**

Truffle is a development environment, testing framework, and asset pipeline for Ethereum smart contracts.

Steps to Install Truffle:

- Open your terminal.
- Run the following command to install Truffle globally:

  npm install -g truffle

- Verify the installation by checking the version:

npm truffle version

- To create a new Truffle project, navigate to a project folder and run:

  truffle init

This will initialize a new Truffle project with the necessary folder structure.

Why Truffle is Important:

- Truffle helps manage the deployment of smart contracts, automates testing, and supports multiple Ethereum networks.
- It integrates well with **Ganache**, a local blockchain simulator.

✓ **Installing Hardhat**

Hardhat is another powerful Ethereum development environment that allows you to compile, deploy, and test Solidity contracts. It is designed for more complex projects.

Steps to Install Hardhat:

- Open your terminal.
- In the project directory, run the following command to install Hardhat:

  npm install --save-dev hardhat
- Initialize a new Hardhat project by running:

  npx hardhat

Follow the prompts to create a new project.

Hardhat will create a configuration file and other project files necessary for Solidity development.

Why Hardhat is Important:

- Hardhat supports advanced debugging, error messages, and network management.
- It is highly customizable and works well for testing and deploying contracts to local, test, or live Ethereum networks.

Use Case Recap:

In this task, you are required to set up a complete Solidity development environment as a blockchain developer. Here's how the task breaks down:

- **Code Editor**: Choose between **Remix** (browser-based) or **Visual Studio Code** (local installation) to write and test smart contracts.
- **Node.js & npm**: These are essential for managing the tools and dependencies required to run and test Solidity code.
- **Solidity Compiler**: The **solc** compiler will turn your Solidity smart contracts into bytecode, allowing them to be deployed on Ethereum.
- **Truffle and Hardhat**: These are development environments that help you manage contract compilation, testing, and deployment across Ethereum networks.

By following these steps, you will have a fully functional Solidity development environment, ready for writing and deploying smart contracts.

Points to Remember

- **Code Editor:**
  - **Remix IDE**: Web-based, no installation required, good for beginners.
  - **Visual Studio Code (VS Code)**: Requires installation, use with Solidity extension for advanced features.
- **Node.js and npm**:
  - **Node.js**: JavaScript runtime necessary for development tools.
  - **npm**: Manages dependencies and packages, install globally using npm install -g.
- **Solidity Compiler (solc):**
  - **Global Installation**: Install with npm install -g solc.
  - **Verify Installation**: Check with solc --version.
- **Ethereum Development Tools:**
  - **Truffle**: Framework for contract management, use truffle init for setup.
  - **Hardhat**: Advanced development environment, initialize with npx hardhat.
- **Verify Installations:**
  - Check versions using **node -v, npm -v, and solc version**.

Test setup by creating and compiling a simple Solidity contract.

Application of learning 2.1.

You are a blockchain developer working for a tech company that has been tasked with developing a decentralized voting system for an upcoming election. The system will allow voters to cast their votes using blockchain technology, ensuring transparency, security, and immutability of the voting process.
Before you can begin coding the smart contracts that will handle vote registration and counting, you need to set up the development environment for writing and testing your Solidity code.
Your task is to:

1. Set up a complete Solidity development environment to start coding the smart contracts for the voting system.
2. Follow the instructions below to install and configure the required tools.

Task Instructions:

- ✓ **Install Code Editor:**

    - Use **Remix IDE** (browser-based) or install **Visual Studio Code** on your machine. If using Visual Studio Code, ensure you install the **Solidity extension** for smart contract development.

- ✓ **Install Node.js and npm:**
    - Download and install **Node.js** from the official website.
    - After installation, use the terminal to verify the installation by typing node **-v** and **npm -v** to check their versions.
- ✓ **Install the Solidity Compiler (solc):**
    - Use **npm** to install the **Solidity compiler (solc)** by running the following command in the terminal:

      ```
      npm install -g solc
      ```
    - Verify the installation by checking the version of the Solidity compiler:

      ```
      solc --version
      ```
- ✓ **Install Ethereum Development Tools (Truffle/Hardhat):**
    - Install **Truffle** globally using npm:

      ```
      npm install -g truffle
      ```
    - Initialize a new Truffle project by navigating to the folder where you want to store the project and running:

      ```
      truffle init
      ```
    - Alternatively, install **Hardhat** by running:

      ```
      npm install --save-dev hardhat
      ```
    - Set up a new Hardhat project by running:

      ```
      npx hardhat
      ```
- ✓ **Testing the Setup:**
    - After installing and configuring the environment, create a sample smart contract in either **Remix**, **Visual Studio Code**, **Truffle**, or **Hardhat**.
    - Compile the contract and check for any errors.
    - If using **Truffle or Hardhat**, run the development environment and deploy the contract to a local Ethereum network.
- ✓ **Deliverable:**
    - Submit a report detailing the steps you followed to install and configure the Solidity environment.
    - Include screenshots showing:
        - Installed software versions (Node.js, npm, solc).
        - A basic smart contract created and compiled.
        - Successful setup of a new project using Truffle or Hardhat.

Indicative content 2.2: Applying solidity concepts

Duration: 5 hrs

Practical Activity 2.2.1: Data types and variables in solidity

Task:

1: You are requested to go in computer lab to write smart contract that contain data types and variables.
2: Read the key readings 2.2.1
3: Apply safety precaution
4: Write smart contract that contain data types and variables.
5: Present your work to trainer.
6: Perform the task provided in application of learning 2.2

Key readings 2.2.1: Data types and variables in solidity

When writing a smart contract in Solidity, it's crucial to properly use data types and variables to store and manipulate data. Here's a step-by-step guide on how to do this:

✓ **Declaring State Variables**

State variables are used to store data on the blockchain and are accessible throughout the contract. You define them using a data type and can optionally assign a value.

This example below,it show how to declare state variable:

```
pragma solidity ^0.8.0;
contract ExampleContract {
  // State variables
  uint public count;        // An unsigned integer variable
  address public owner;            // An Ethereum address
  string public greeting;          // A string variable
  // Constructor to initialize state variables
  constructor() {
count = 0;         // Initialize the count variable
  owner = msg.sender;  // Initialize owner to the address that deployed the
contractgreeting = "Hello, World!"; // Initialize the greeting
  }}
```

- ✓ **Using Local Variables Inside Functions**

  Local variables are temporary and only exist during the function execution. They are not stored on the blockchain.

```
pragma solidity ^0.8.0;
contract LocalVariableExample {
  function add(uint a, uint b) public pure returns (uint) {
  // Local variables
  uint sum = a + b;// Local variable 'sum'
  return sum;     }}
```

  In this example, **a**, **b**, and **sum** are local variables that only exist while the add function is executing.

- ✓ **Defining Structs (Custom Data Types)**

  Structs allow you to create complex data types by combining multiple variables. They are useful for defining more complex entities like a person or product.

```
pragma solidity ^0.8.0;
contract StructExample {
  // Define a custom struct data type
  struct Person {
  string name;
  uint age;
address wallet;
  }
  // State variable of type struct
  Person public person;
  // Constructor to initialize the struct
  constructor(string memory _name, uint _age) {
  person = Person(_name, _age, msg.sender);  // Set the struct values
  }}
```

  Here, we define a struct **Person** with three variables: **name**, **age**, and **wallet**. The constructor initializes the struct when the contract is deployed.

- ✓ **Global Variables**

  Solidity has several built-in global variables that provide information about the blockchain. Examples include **msg.sender**, **block.timestamp**, and **msg.value**.

```
pragma solidity ^0.8.0;
contract GlobalVariableExample {
  // Function to return the address of the sender
  function getSenderAddress() public view returns (address) {
  return msg.sender;  // msg.sender is a global variable
  }
  // Function to return the current block timestamp
  function getBlockTimestamp() public view returns (uint) {
  return block.timestamp;  // block.timestamp is a global variable
  }}
```

Practical Activity 2.2.2: Use of Functions

Task:

1: You are requested to go in computer lab to use functions.
2: Read the key readings 2.2.2
3: Apply safety precaution
4: Use functions in smart contract
5: Present your work to trainer.
6: Perform the task provided in application of learning 2.2

Key readings 2.2.2: Use of functions

In Solidity, functions are the primary way to interact with a smart contract. They define specific behavior that allows for reading from or writing to the contract's state, executing logic, and interacting with other contracts. Here's a breakdown of how functions work in Solidity, including the key components and types of functions.

- ✓ **Key Components of Functions:**
- **Function Signature**: The function signature includes the function name, parameter list, and return types.

  Syntax:

  function functionName(parameterType parameterName) public returns (returnType) { }

- **Visibility Specifiers**:

  - **Public**: Can be called both internally and externally (i.e., by users or other contracts).
  - **Private**: Can only be called within the contract itself and not by derived contracts.
  - **Internal**: Can be called within the contract and derived contracts.
  - **External**: Can only be called from outside the contract. These functions are generally more gas-efficient when called externally.

- ✓ **State Mutability**:

- **view**: A function marked with view ensures that the function does not alter the contract's state but only reads it.

  Syntax:

  ```
  function getBalance() public view returns (uint) {return balance;}
  ```

- **Pure**: A pure function ensures that the function neither reads nor modifies the contract's state. It purely performs calculations.

  Syntax:

  ```
  function add(uint a, uint b) public pure returns (uint) {
     return a + b;}
  ```

- **payable**: A payable function allows the contract to receive Ether. Without this modifier, functions cannot accept Ether.

  Syntax:

  ```
  function deposit() public payable {// Accepts Ether transfers}
  ```

✓ **Return Values**: Solidity functions can return single or multiple values. These values are defined in the function signature after the returns keyword.

Syntax:

```
function getInfo() public view returns (uint, bool) {return (age, isActive);}
```

Practical Activity 2.2.3: Control structure

Task:

1: You are requested to go in computer lab to use control structures.

2: Read the key readings 2.2.3

3: Apply safety precaution

4: Use control structures in smart contract

5: Present your work to trainer.

6: Perform the task provided in application of learning 2.3

Key readings 2.2.3:Control structures

✓ **Conditional Statements**

- if statements:

  Execute a block of code if a specified condition is true.

  Syntax:

  Solidity

  ```
  if (condition) {// Code to execute if condition is true}
  ```

- **else** and **else if**:

  Provide alternative code blocks to execute if the initial condition is false.

Syntax:

Solidity

if (condition1) {// Code to execute if condition1 is true} else if (condition2) {// Code to execute if condition1 is false and condition2 is true} else {// Code to execute if neither condition1 nor condition2 is true}

- **Nested conditions:**

  You can nest **if** statements within other **if** statements for more complex logic.

✓ **Loops**

- **for** loops:

  Execute a block of code a specified number of times.

  Syntax:

  Solidity

  for (initialization; condition; increment) {// Code to execute}

- **while** loops:

  Execute a block of code as long as a condition is true.

  Syntax:

  Solidity

  while (condition) {// Code to execute}

- **do-while** loops:

  Execute a block of code at least once, then repeat as long as a condition is true.

  Syntax:

  Solidity

  do {// Code to execute} while (condition);

Practical Activity 2.2.4: Arrays and structs

Task:

1: You are requested to go in computer lab to use arrays and structs.

2: Read the key readings 2.2.4

3: Apply safety precaution

4: Use arrays and structs in smart contract

5: Present your work to trainer.

6: Perform the task provided in application of learning 2.4

Key readings 2.2.4: Arrays and structs

✓ **Arrays**

- **Declaration:** Arrays are declared using square brackets []. You can specify the size of the array or leave it dynamic.
  - **Fixed-size array: uint[5] myArray;**
  - **Dynamic-size array: uint[] myDynamicArray;**
- **Accessing elements:** Use index notation to access elements within an array. The first element has an index of 0.
  - **myArray[0]** accesses the first element.
- **Modifying elements:** You can modify elements by assigning new values to them.
  - **myArray[2] = 10;**
- **Iterating over arrays:** Use loops like f**or** or **while** to iterate over each element in an array.

✓ **Structs**

Defining structs: Create custom data types with multiple fields using the **struct** keyword.

Solidity

struct Student {string name;uint8 grade;}

- Use code with caution.

Creating instances: Create instances of structs and access their fields using dot notation.

Solidity

Student student1 = Student("Alice", 95);

String studentName = student1.name;

Storing structs in arrays: You can store structs in arrays for efficient data management.

Solidity

Student[] students;

students.push(Student("Bob", 88));

Example:

Solidity

pragma solidity ^0.8.0;

contract StudentGradeCalculator {struct Student {string name;uint8 grade;}
Student[] public students;function addStudent(string memory _name, uint8 _grade) public {students.push(Student(_name, _grade));}
function calculateAverageGrade() public view returns (uint8) {uint256 totalStudents = students.length;uint256 totalGrades = 0;
for (uint256 i = 0; i < totalStudents; i++) {totalGrades += students[i].grade;}return totalGrades / totalStudents;}}

Practical Activity 2.2.5: Events and Logging

Task:

1: You are requested to go in computer lab to use Events and Logging

2: Read the key readings 2.2.5

3: Apply safety precaution

4: Use events and Logging

5: Present your work to trainer.

6: Perform the task provided in application of learning 2.5

Key readings 2.2.5: Events and Logging

✓ **Events**

- **Purpose:** Used to emit information to external listeners or clients.
- **Declaration:** Use the **event** keyword followed by the event name and parameters.
- **Indexing:** Index event parameters for efficient filtering and querying.
- **Emitting:** Use the **emit** keyword to trigger an event.

Example:

Solidity

pragma solidity ^0.8.0;

contract MyContract {event Transfer(address indexed from, address indexed to, uint256 value);function transferFunds(address to, uint256 amount) public {// ... transfer logic ...emit Transfer(msg.sender, to, amount);}}

✓ **Logging**

- Logging is used record information for debugging and auditing.
- Use functions like log0, log1, log2, etc., to log data.
- Format log messages using placeholders and arguments.
- Use logging judiciously to avoid overwhelming the blockchain.

Example:

Solidity

pragma solidity ^0.8.0;

contract MyContract {function doSomething() public {uint256 value = 42;log0("Value:", value);}}

Practical Activity 2.2.6: Error handling

Task:

1: You are requested to go in computer lab to adopts error handling
2: Read the key readings 2.2.6
3: Apply safety precaution
4: Apply error handling in smart contract
5: Present your work to trainer.
6: Perform the task provided in application of learning 2.6

Key readings 2.2.6: Error handling

✓ **Key Points:**

- **Require:** Used to assert conditions and revert the transaction if the condition is not met.
- **Assert:** Like **require**, but intended for internal errors that should never occur under normal circumstances.
- **Revert:** Reverts the transaction and sends a custom error message.

Example:

Solidity

```
pragma solidity ^0.8.0;
contract MyContract {
 function divide(uint256 a, uint256 b) public pure returns (uint256) require(b != 0, "Division by zero");return a / b;}}
```

Points to Remember

- **Error handling** refers to mechanisms that allow developers to manage and respond to unexpected conditions or failures during the execution of a smart contract
- They are three primary ways to handle errors: **require, assert** and **revert**
- To declare variables choose the correct data type for your variables based on the data you want to store (e.g., uint for numbers, address for Ethereum addresses).
- Variables can be used within functions for temporary data storage and manipulation (local variables).
- **Access Global Variables**: Use Solidity's global variables to get information about the transaction and blockchain.

Application of learning 2.2

You are developing a decentralized marketplace on the Ethereum blockchain that allows users to list items for sale, purchase items, and track transactions securely. As part of this project, you are required to create a smart contract in Solidity that handles item listings, sales transactions, and ensures proper gas and memory optimization. Additionally, the contract needs to connect to user wallets (e.g., MetaMask), perform both read-only and write operations, and log events for transaction tracking.

Indicative Content 2.3: Implementing function Interaction

Duration: 5hrs

Practical Activity 2.3.1 Connect to wallet

Task:

1: You are requested to go in computer lab to connect wallets
2: Read the key readings 2.3.1
3: Apply safety precaution
4: Connect to wallets
5: Present your work to trainer.
6: Perform the task provided in application of learning 2.3

Key readings 2.3.1.: Connect to wallet

Connecting a smart contract to MetaMask wallets involves several steps, from setting up MetaMask to writing code that interacts with MetaMask. Below is a guide on how to achieve this.

Steps to Connect to MetaMask Wallets:

✓ **Install MetaMask:**

- **Browser Extension**: Go to the **MetaMask website** and install the browser extension (available for Chrome, Firefox, and Brave).
- **Create or Import a Wallet**: Follow the instructions to either create a new wallet or import an existing one using the seed phrase.
- **Connect to a Blockchain Network**: By default, MetaMask connects to the Ethereum mainnet, but you can switch to a test network (e.g., Ropsten, Goerli) for development.

✓ **Connect MetaMask to a Smart Contract (Frontend Setup)**

You can use JavaScript along with Web3.js or Ethers.js to connect to MetaMask and interact with your smart contract.
Here's how:

Option 1: Using Web3.js

- **Install Web3.js**

First, install the Web3 library to interact with Ethereum:

npm install web3

- **HTML & JavaScript Code to Connect to MetaMask**

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>MetaMask Connection</title>
</head>
<body>
<h1>Connect to MetaMask</h1>
<button id="connectButton">Connect Wallet</button>
<p id="account">No account connected</p>
<script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script>
<script>
let web3;
// Check if MetaMask is installed
if (typeof window.ethereum !== 'undefined') {
web3 = new Web3(window.ethereum);
// Connect to MetaMask
async function connectMetaMask() {try {// Request account access const
accounts = await ethereum.request({ method: 'eth_requestAccounts' });
document.getElementById('account').innerText = `Connected account:
${accounts[0]}`;} catch (error) {console.error("Error connecting to MetaMask",
error);}}
document.getElementById('connectButton').addEventListener('click',
connectMetaMask);} else {alert('MetaMask is not installed. Please install
MetaMask and try again.');}
</script>
</body>
</html>
```

- The above code creates a simple page where you can click the "Connect Wallet" button to connect to MetaMask.
- If MetaMask is installed, it will prompt the user to connect their wallet and return the connected account.

Interacting with the Smart Contract

- Once connected to MetaMask, you can interact with a deployed smart contract by providing its ABI (Application Binary Interface) and contract address.

Here's how you can do that using Web3.js:

```
const contractABI = [/* Your contract's ABI here */];
const contractAddress = "0xYourContractAddress"; // Create contract instance
const contract = new web3.eth.Contract(contractABI, contractAddress);//
Example: Calling a function from the smart contract
contract.methods.someFunction().call()then(result => {console.log("Smart
```

contract function result: ", result);})catch(error => {console.error("Error calling smart contract function: ", error);});

Option 2: Using Ethers.js

Ethers.js is another popular library for interacting with Ethereum:

- **Install Ethers.js**:

  Open cmd,type: ***npm install ethers***

- **Connecting MetaMask with Ethers.js**:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>MetaMask and Ethers.js</title>
</head>
<body>
<h1>Connect to MetaMask</h1>
<button id="connectButton">Connect Wallet</button>
<p id="account">No account connected</p>
<script src="https://cdn.jsdelivr.net/npm/ethers/dist/ethers.min.js"></script>
<script>
// Check if MetaMask is installed
if (typeof window.ethereum !== 'undefined') {
async function connectMetaMask() {
try {// Request account access
const provider = new ethers.providers.Web3Provider(window.ethereum);await
provider.send("eth_requestAccounts", []);
const signer = provider.getSigner();
const account = await signer.getAddress();
document.getElementById('account').innerText = `Connected account:
${account}`;} catch (error) {console.error("Error connecting to MetaMask",
error);}}
document.get
ElementById('connectButton').addEventListener('click', connectMetaMask);}
else {alert('MetaMask is not installed. Please install MetaMask and try again.');}
</script>
</body>
</html>
```

- **Interacting with the Smart Contract using Ethers.js**

  Similar to Web3.js, you can interact with your smart contract once connected to MetaMask:

```
const contractABI = [/* Your contract's ABI here */];
const contractAddress = "0xYourContractAddress";
// Create provider and signer
const provider = new ethers.providers.Web3Provider(window.ethereum);
const signer = provider.getSigner();
// Create contract instance
const contract = new ethers.Contract(contractAddress, contractABI, signer);
// Example: Calling a function from the smart contract
contract.someFunction()
then(result => {
console.log("Smart contract function result: ", result);}).catch(error => {
console.error("Error calling smart contract function: ", error);});
```

✓ **Testing and Deploying**

- Once the smart contract is written and deployed on a blockchain (either on a local blockchain like Ganache, a testnet like Ropsten, or Ethereum mainnet), users can interact with it via the front end that connects to MetaMask.
- The interaction will allow users to deploy contracts, send transactions, or execute smart contract functions via MetaMask.
- To connect a decentralized application (dApp) to Trust Wallet, you can use the **WalletConnect** protocol. Trust Wallet supports WalletConnect, which allows users to interact with dApps using their mobile wallet without needing a browser extension like MetaMask.

  Here's how to connect to Trust Wallet using WalletConnect:

✓ **Steps to Connect to Trust Wallet:**

- **Set Up WalletConnect in Your dApp**

  You can use WalletConnect to connect Trust Wallet (or any supported wallet) to your dApp. You can achieve this using JavaScript libraries like Web3.js or Ethers.js.

Option 1: Using Web3.js **with WalletConnect**

✓ **Install Required Libraries**

Install Web3.js and WalletConnect provider:

```
npm install web3 @walletconnect/web3-provider
```

✓ **Code to Connect Trust Wallet via WalletConnect**

Here's an example using Web3.js and WalletConnect:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Connect to Trust Wallet</title>
</head>
<body>
  <h1>Connect to Trust Wallet via WalletConnect</h1>
  <button id="connectButton">Connect Wallet</button>
  <p id="account">No account connected</p>
  <script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/@walletconnect/web3-
provider/dist/umd/index.min.js"></script>
  <script>
    let web3;
    // Set up WalletConnect provider
    const provider = new WalletConnectProvider.default({
      rpc: {
        1: "https://mainnet.infura.io/v3/YOUR_INFURA_PROJECT_ID" // Replace with your
Infura or other RPC URL
      },
    });
    // Connect to WalletConnect
    async function connectWalletConnect() {
      try {
        // Enable the provider (QR code modal will show up)
        await provider.enable();
        // Create Web3 instance
        web3 = new Web3(provider);
        // Get accounts
        const accounts = await web3.eth.getAccounts();
        document.getElementById('account').innerText = `Connected account:
${accounts[0]}`;
      } catch (error) {
        console.error("Error connecting to WalletConnect", error);
      } }
    document.getElementById('connectButton').addEventListener('click',
connectWalletConnect);
  </script>
</body></html>
```

✓ **How it works**:

- When the user clicks the **Connect Wallet** button, it uses WalletConnect to open a QR code scanner in Trust Wallet.
- The user scans the QR code with Trust Wallet to establish the connection.
- Once connected, the dApp can interact with the user's wallet for transactions and contract interactions.

Option 2: Using **Ethers.js** with WalletConnect

You can also use WalletConnect with Ethers.js:

✓ **Install Ethers.js and WalletConnect**

npm install ethers @walletconnect/web3-provider

✓ **Code Example with Ethers.js and WalletConnect**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Connect to Trust Wallet with WalletConnect</title>
</head>
<body>
  <h1>Connect to Trust Wallet</h1>
  <button id="connectButton">Connect Wallet</button>
  <p id="account">No account connected</p>
  <script src="https://cdn.jsdelivr.net/npm/ethers/dist/ethers.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/@walletconnect/web3-
provider/dist/umd/index.min.js"></script>
  <script>
    // Set up WalletConnect provider
    const provider = new WalletConnectProvider.default({
      rpc: {
        1: "https://mainnet.infura.io/v3/YOUR_INFURA_PROJECT_ID" // Replace with your
Infura or other RPC URL
      }
    });

    // Connect to WalletConnect
    async function connectWalletConnect() {
      try {
        // Enable WalletConnect
        await provider.enable();
        // Create Ethers.js provider
        const ethersProvider = new ethers.providers.Web3Provider(provider);
        // Get signer
        const signer = ethersProvider.getSigner();
        // Get account address
        const account = await signer.getAddress();
        document.getElementById('account').innerText = `Connected account: ${account}`;
      } catch (error) {
        console.error("Error connecting to WalletConnect", error);
      } }
    document.getElementById('connectButton').addEventListener('click',
connectWalletConnect);
  </script></body></html>
```

- This code above is similar to the Web3.js approach, but here it uses **Ethers.js** for interacting with the blockchain.

✓ **Interacting with Smart Contracts**

Once you have connected Trust Wallet via WalletConnect, you can interact with your smart contract like any other Ethereum wallet:

- **Deploy contracts** or **send transactions** using the Web3 or Ethers instance connected to Trust Wallet.

Here's an example of calling a contract function using Web3.js:

```
const contractABI = [/* Your contract ABI here */];
const contractAddress = "0xYourContractAddress";
// Create contract instance
const contract = new web3.eth.Contract(contractABI, contractAddress);
```

// Example: Calling a contract function (e.g., transfer tokens)
contract.methods.someFunction().send({ from: accounts[0] })
then(receipt => {console.log("Transaction successful: ", receipt);}) catch(error => {
console.error("Error calling contract function: ", error);});

✓ **Disconnecting Trust Wallet**

To disconnect Trust Wallet (WalletConnect), you can call:
provider.disconnect();
This will terminate the connection with the user's wallet.

Practical Activity 2.3.2: Access the contract address

Task:

1: You are requested to go in computer lab to access the contract address
2: Read the key readings 2.3.3
3: Apply safety precaution
4: Access the contract address
5: Present your work to trainer.
6: Perform the task provided in application of learning 2.8

Key readings 2.3.3. Access the Contract Address

Accessing the contract address can be done in several contexts, such as during deployment, after deployment, and within the smart contract itself. Below are the methods for each context.

✓ **Accessing the Contract Address During Deployment**

When you deploy a smart contract, the blockchain assigns it an address, which you can access programmatically using the deployment tools or libraries like Web3.js or Ethers.js.
Using Web3.js
Here’s an example of how to access the contract address during deployment using Web3.js:
const Web3 = require('web3');
const web3 = new Web3('https://YOUR_ETHEREUM_NODE_URL'); // Replace with your Ethereum node URL
const contract ABI = [/* Your contract's ABI */];

const contract By tecode = '0xYourContractBytecode';// Create a new contract instance

const My Contract = new web3.eth.Contract(contract ABI);

// Deploy the contract

My Contract.deploy({

data: contract Bytecode,

arguments: [/* Constructor arguments if any */]})

send({from: '0xYourDeployerAddress', // Address that deploys the contract gas: 1500000,gasPrice: '30000000000' // Gas price in wei}) then((new Contract Instance) => {

console.log('Contract deployed at address:', new Contract Instance.options.address);});

- **New Contract Instance.options.address**: This will give you the address of the newly deployed contract.

➢ **Using Ethers.js**

Here's how to do the same using Ethers.js:

const { ethers } = require("ethers");

// Set up provider and signer

const provider = new ethers.providers.Web3Provider(window.ethereum);

const signer = provider.getSigner();

const contractABI = [/* Your contract's ABI */];

const contractBytecode = '0xYourContractBytecode';

// Create a contract factory

const MyContractFactory = new ethers.Contract Factory(contract ABI, contract Bytecode, signer);// Deploy the contract

const my Contract = await My ContractFactory.deploy(/* Constructor arguments if any */);// Wait for the transaction to be mined

await my Contract.deployed();

// Access the contract address

console.log('Contract deployed at address:', my Contract.address);

- **My Contract.address**: This gives you the address of the contract after it is deployed.

✓ **Accessing the Contract Address After Deployment**

If you have the contract address already (from deployment or documentation), you can use it to interact with the contract in your application.

- **Using Web3.js**

const contractABI = [/* Your contract's ABI */];

const contractAddress = "0xYourContractAddress"; // The known deployed contract address

// Create contract instance

const MyContract = new web3.eth.Contract(contractABI, contractAddress);

// Now you can call contract methods

- **Using Ethers.js**

const contractABI = [/* Your contract's ABI */];

const contractAddress = "0xYourContractAddress"; // The known deployed contract address.
// Create a contract instance
const My Contract = new ethers.Contract (contractAddress, contract ABI, signer);
// Now you can call contract methods

✓ **Accessing the Contract Address Within the Smart Contract**

Within the smart contract itself, you can access its own address using the address(this) keyword in Solidity.
Example:
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract MyContract {// Function to return the contract's address
function getContractAddress() public view returns (address) {return address(this); // Returns the contract's own address}}

- **address(this)**: This returns the address of the contract when called.

✓ **Finding the Contract Address on a Blockchain Explorer**

If the contract has already been deployed and you want to find the address, you can use a blockchain explorer like Etherscan:

- **Go to Etherscan**: Navigate to **Etherscan** or a similar explorer.
- **Search by Address or Transaction Hash**: You can search using the wallet address that deployed the contract or the transaction hash of the deployment.
- **Contract Information**: The contract address will be listed in the transaction details or the wallet address's transactions.

Theoretical Activity 2.3.3: Use of Blockchain Explorer

Tasks:

1: Answer the following questions:

I. What do you understand by the following term Blockchain Explorer
II. What are different types of Blockcchain Explorer?
III. What role does a blockchain explorer play in enhancing transparency in blockchain networks?

2: Write your findings on paper or flipchart
3: Present your findings to the trainer or colleagues
4: For more clarification read the key readings 2.3.4.
5: In addition, ask questions where necessary.

Key readings 2.3.4. Use of Blockchain Explorer

A **Blockchain Explorer** is an online platform or tool designed to provide a user-friendly interface for viewing, searching, and analyzing the data contained within a blockchain. It functions as a search engine for blockchain networks, allowing users to access and explore various activities on the blockchain, such as transactions, blocks, wallet addresses, and network statistics.

✓ **Keyfeatures of Blockchain Explorer**

- Transaction Tracking:
  Users can search for individual transactions by entering the transaction ID or hash. The explorer provides details such as the transaction amount, sender and receiver addresses, transaction fees, and the timestamp of when the transaction was confirmed.
- **Block Information**:
  The explorer allows users to view specific blocks within the blockchain, showing details such as the block number, block size, the number of transactions included, the time of block creation, and the miner responsible for validating the block.
- **Address Monitoring**:
  Users can input a wallet address to view its transaction history, balance, and all activities associated with that address. This feature allows for monitoring cryptocurrency holdings and tracking the movement of funds.
- **Smart Contract and Token Information**:
  On smart contract-enabled blockchains (e.g., Ethereum), explorers allow users to track interactions with smart contracts and view token transfers (such as ERC-20 and ERC-721 tokens). Users can explore token balances, transfers, and contract execution history.
- **Mining Data**:
  Blockchain explorers provide insights into mining activities, including the rewards miners receive for validating blocks, the difficulty level of mining, and the overall hash rate of the network.
- **Network Statistics**:
  Explorers display real-time data on network activity, such as the number of active nodes, block times, transaction volumes, and total circulating supply of a cryptocurrency.

- **Auditing and Verification**:

  Blockchain explorers offer transparency by allowing anyone to verify transactions and track the movement of assets. This ensures accountability and allows for auditing on both public and permissioned blockchains.

✓ **Types of Blockchain explorer**

There are several types of **blockchain explorers**, each tailored to specific blockchain networks, applications, or use cases. Below is a breakdown of the types of blockchain explorers based on the technology they explore and their key functionalities.

- **Bitcoin Blockchain Explorer**:

➢ These explorers are dedicated to tracking activities on the Bitcoin blockchain, including transactions, blocks, and wallet addresses.

➢ Transaction status, mining data, block details, wallet balances, and transaction fees.

Examples of Bitcoin Blockchain Explorer:

- **Blockchain.com Explorer**: Widely used to track Bitcoin transactions.
- **Blockstream Explorer**: Offers advanced features for Bitcoin, including support for testnet and Liquid network.

- **Ethereum Blockchain Explorer**:

➢ Focused on tracking data on the Ethereum blockchain, including ETH transactions, token transfers, smart contracts, and decentralized finance (DeFi) activities.

➢ Ethereum transaction tracking, ERC-20/721 token transfers, smart contract interactions, and gas fees.

Examples Ethereum Blockchain Explorer:

- **Etherscan**: The most popular Ethereum explorer for tracking transactions, tokens, and smart contracts.
- **Ethplorer**: Specializes in token and smart contract transactions, providing detailed token-related information.

- **Multi-Blockchain Explorers**:

➢ **Purpose**: These explorers support multiple blockchain networks, allowing users to track and compare transactions across various blockchains within a single platform.

➢ **Features**: Cross-chain tracking, multi-currency wallet balance views, and transaction history across different blockchains.

Examples of Multi-Blockchain Explorers :

- **Blockchair**: Supports Bitcoin, Ethereum, Bitcoin Cash, Litecoin, Dash, and more, offering cross-chain search.
- **Blockchain.com Explorer**: Provides tracking for Bitcoin, Ethereum, and Bitcoin Cash.

- **Altcoin Blockchain Explorers**:
  - **Purpose**: Designed for specific alternative cryptocurrencies (altcoins) other than Bitcoin and Ethereum.
  - **Features**: Transaction tracking, block details, and mining data for specific altcoins.

  Examples of Altcoin Blockchain Explorers:
  - **Litecoin Explorer**: Provides insight into Litecoin-specific transactions and blocks.
  - **Monero Explorer (XMR)**: Designed for Monero, emphasizing privacy while showing essential transaction and mining details.
- **Private Blockchain Explorers:**
  - **Purpose**: Built for **private or permissioned blockchains** where access to the blockchain is restricted to authorized participants.
  - **Features**: Allows internal users to track transactions, blocks, and network status, ensuring transparency within private or consortium blockchains.

  Examples of private Blockchain Explorers:
  - **Hyperledger Explorer**: A tool for tracking Hyperledger Fabric-based blockchain activities.
  - **Quorum Explorer**: For the Quorum blockchain, a permissioned blockchain built for enterprise use.
- **DeFi Blockchain Explorers**:
  - **Purpose**: Focused on **Decentralized Finance (DeFi)** activities, allowing users to track and analyze DeFi transactions, liquidity pools, staking, and more.
  - **Features**: Tracks interactions across DeFi protocols, token swaps, liquidity pools, lending, and borrowing.

  Examples of DeFi Blockchain Explorers:
  - **Zapper.fi**: Tracks user activities across multiple DeFi platforms.
  - **Dune Analytics**: Allows users to create and explore customized dashboards that analyze DeFi activities across various platforms.
- **NFT Blockchain Explorers**:
  - **Purpose**: Designed to track the creation, transfer, and ownership of **non-fungible tokens (NFTs)** across various blockchain networks.
  - **Features**: Shows NFT ownership history, transfers, minting activities, and sales data.

  Examples of NFT Blockchain Explorers:
  - **Opensea NFT Explorer**: Tracks NFT ownership and transactions on the Opensea marketplace.
  - **Etherscan NFT Tracker**: A feature of Etherscan that tracks NFT movements on the Ethereum blockchain.
- **Layer 2 Blockchain Explorers**:
  - **Purpose**: Focused on **Layer 2 scaling solutions** that run on top of Layer 1 blockchains like Ethereum to improve transaction speeds and reduce costs.

- **Features**: Tracks Layer 2 transactions, smart contracts, and token transfers on scaling solutions such as **Optimistic Rollups** and **zk-Rollups**.

  Examples of Layer 2 Blockchain Explorers:

  - **Polygonscan**: An explorer for the Polygon (formerly Matic) Layer 2 solution.
  - **Optimistic.etherscan.io**: A Layer 2 explorer for Optimism on Ethereum, showing transactions and token transfers.

- **Interoperability Blockchain Explorers**:
  - **Purpose**: These explorers support **interoperability protocols** that enable data and asset transfers across different blockchains.
  - **Features**: Tracks cross-chain transactions, bridges, and token transfers between interoperable blockchain networks.

  Examples of Interoperability Blockchain Explorers:

  - **Polkascan**: For Polkadot's parachains, allowing cross-chain transaction tracking.
  - **Cosmos Explorer**: For the Cosmos blockchain, which focuses on enabling interoperability between different blockchain networks.

✓ **Uses of a Blockchain Explorer**

- **Transparency**: It enables transparency by making the blockchain's contents visible to the public, ensuring that every transaction and block can be scrutinized.
- **Verification**: Users can verify the success, status, and details of their transactions.
- **Research**: Developers, miners, and blockchain analysts can use it to study the performance of the blockchain and smart contracts.
- **Compliance and Auditing**: Organizations can use blockchain explorers to audit transactions and ensure compliance with regulatory requirements.

Practical Activity 2.3.4: Perform function operations

Task:

1: You are requested to go in computer lab to Perform function operations

2: Read the key readings 2.3.4

3: Apply safety precaution

4: Perform function operations

5: Present your work to trainer.

6: Perform the task provided in application of learning 2.3

Key readings 2.3.4 Perform Function Operation

In blockchain, **read-only** and **write operations** refer to the ways users interact with the blockchain's state. Understanding how each operation works is critical for building and interacting with decentralized applications (dApps) and smart contracts. Here's how they function practically:

✓ **Read-Only Operations (View/Pure Functions)**

Read-only operations **do not modify the blockchain's state**. They are typically used to query or retrieve information from the blockchain. These operations do not require gas fees because they do not involve mining or consensus among network nodes. They can be performed locally by nodes without broadcasting the request across the network.

Key Characteristics:

- **No State Change**: Read-only operations only fetch existing data without altering it.
- **No Gas Fees**: Since no state change is happening, no gas is consumed.
- **Immediate Execution**: They return the result almost immediately since there's no need to wait for block confirmation.
- **Local Execution**: These operations are executed on a local node without involving other network participants.

How it Works Practically:

- **View Function**: This type of function can read from the blockchain's state (e.g., reading a balance, checking a stored value) but cannot change it.

  Example in Solidity:

  // A read-only function that returns a stored number

  function getStoredNumber() public view returns (uint) {return storedNumber;}

- **Pure Function**: These functions are also read-only but are entirely self-contained, meaning they do not even access the blockchain's state. They can only perform calculations based on inputs.

  Example in Solidity:

  // A pure function that returns the sum of two numbers

  function addNumbers(uint a, uint b) public pure returns (uint) {return a + b;}

- **Execution**: When you call a read-only function (e.g., using Web3.js or any blockchain wallet), the function is executed on your local node or a blockchain node provider (like Infura). It doesn't generate a transaction on the network, so it doesn't need to be mined.

  Example with Web3.js (for an Ethereum smart contract):

  // Calling a view function to read a stored number

  const storedNumber = await myContract.methods.getStoredNumber().call();

  console.log(storedNumber); // No gas is spent

✓ **Write Operations (State-Changing Transactions)**

Write operations **modify the blockchain's state**. These operations change the data stored on the blockchain, such as updating variables, transferring tokens, or interacting with smart contracts in a way that affects their state.

Key Characteristics:

- **State Change**: Write operations update the state (e.g., change a variable's value, transfer tokens, create or delete records).
- **Gas Fees**: Gas is required to incentivize miners or validators to include the transaction in a block.
- **Delayed Execution**: Write operations are processed as transactions and must be included in a block before becoming part of the blockchain, which may take time depending on the network's traffic.
- **Consensus Required**: Because the operation changes the blockchain's state, it must be broadcasted to the network and validated by nodes or miners.

How it Works Practically:

- **State-Changing Functions**: These are functions that alter the blockchain's state. For example, a function that sets a value or transfers tokens will require a transaction.

  Example in Solidity:

  ```
  // A state-changing function that sets a stored number
  function setStoredNumber(uint _num) public {storedNumber = _num;}
  ```

- **Transaction Submission**: When you call a state-changing function, it creates a transaction on the blockchain. This transaction must be broadcasted to the network and included in a block, which requires gas fees.

  Example with Web3.js (for an Ethereum smart contract):

  ```
  // Calling a state-changing function that sets a number
  await myContract.methods.setStoredNumber(42).send({ from: myAddress, gas: 30000});
  ```

  In this example above:

  - **setStoredNumber(42)** is a state-changing function call.
  - **send()** submits the transaction to the network.
  - The **from** parameter is the Ethereum address sending the transaction.
  - The **gas** parameter specifies the gas limit for the transaction.

- **Gas and Cost**: The amount of **gas** required depends on the complexity of the function being executed. Simple operations (like transferring tokens) consume less gas than complex operations (like executing multiple smart contracts in a single transaction). Users must pay gas fees, typically in the native cryptocurrency of the blockchain (e.g., Ether for Ethereum).

- **Transaction Confirmation**: After the transaction is submitted, it gets included in the next available block. The blockchain network's miners or validators will process the transaction, verify it, and record the result on the blockchain. Once the transaction is confirmed, the state is updated immutably.

- **Immutability**: After the transaction is mined and included in a block, the state change becomes permanent. This could be updating a variable in a smart contract, transferring tokens, or any other action that alters blockchain data.
- **Events and Logs**: Many smart contracts use **events** to log important data after state changes. These logs are useful for tracking state changes and triggering off-chain actions.

  Example of event emission in Solidity:

  ```
  // State-changing function that emits an event
  event NumberChanged(uint newNumber);
  function setStoredNumber(uint _num) public {storedNumber = _num;emit
  NumberChanged(_num);  // Emitting an event}
  ```

✓ **Example in Context: Token Transfer**

In a token contract (e.g., ERC-20), reading a user's balance is a **read-only operation**, while transferring tokens between accounts is a **write operation**.

- **Read-Only (Balance Check)**:

  ```
  // A view function that checks the balance of an address
  function balanceOf(address account) public view returns (uint) {return
  balances[account];}
  ```

  Example with Web3.js:

  ```
  // Reading a token balance (read-only, no gas)
  const balance = await tokenContract.methods.balanceOf(myAddress).call();
  console.log(balance);
  ```

- **Write Operation (Token Transfer)**:

  ```
  // A state-changing function to transfer tokens
  function transfer(address recipient, uint amount) public returns (bool) {
  require(balanceOf(msg.sender) >= amount, "Insufficient balance");
  balances[msg.sender] -= amount;
  balances[recipient] += amount;
  emit Transfer(msg.sender, recipient, amount);return true;}
  ```

  Example with Web3.js:

  ```
  // Transferring tokens (write operation, requires gas)
  await tokenContract.methods.transfer(recipientAddress, amount).send({ from:
  myAddress });
  ```

Points to Remember

- ✓ **Read-only operations** fetch data from the blockchain without altering the state. They are free and fast to execute.
- ✓ **Write operations** change the state of the blockchain and require a transaction to be mined, which incurs gas fees and requires confirmation from the network.
- ✓ A **Blockchain Explorer** is an online tool or web application that allows users to view and search detailed information about a blockchain's data, including transactions, blocks, addresses, and other blockchain activities.
- ✓ **Types of blockchain explorers** categorized by the blockchain network they operate on, the features they offer, and their use cases:
  - Bitcoin Blockchain Explorer
  - Ethereum Blockchain Explorer
  - Multi-Blockchain Explorers
  - Altcoin Blockchain Explorers
  - Private Blockchain Explorers
  - DeFi (Decentralized Finance) Blockchain Explorers
  - NFT(Non-Fungible token) Blockchain Explorers
  - Layer 2 Blockchain Explorers
  - Interoperability Blockchain Explorers
- ✓ **Function of a Blockchain explorer**
  - Transparency
  - Verification of transactions
  - Used in Research
  - Compliance and Auditing,etc

This balance between read-only and write operations ensures that blockchain remains secure and transparent while minimizing unnecessary computational costs.

Application of learning 2.3.

As a blockchain developer, you are requested to:

I. Write a simple smart contract where you declare state variables of different data types like uint, bool, string, address, and bytes. Additionally, implement functionality to perform function operations and connect the smart contract to wallets for interaction.

II. Assign values to these variables, make them public so that their values can be read from outside the contract, perform function operations, and ensure wallet integration for reading and writing values.

Indicative content 2.4: Optmizing Gas cost

Duration: 5hrs

Practical Activity 2.4.1. Proper analysis of Gas cost

Task:

1: You are requested to go in computer lab to apply the analysis of gas cost.
2: Read the key readings 2.4.1
3: Apply safety precaution
4: perform proper analysis of gas cost
5: Present your work to trainer.
6: perform the task provided in application of learning 2.4

Key readings 2.4.1: proper analysis of gas cost

✓ **Calculating the Cost of Ethereum Transfer**

In Ethereum, gas is the unit used to measure the amount of computational effort required to execute operations. When transferring Ether, the gas cost depends on several factors:

- **Formula to Calculate Gas Cost**:
  Gas Cost = Gas Used × Gas Price
- **Gas Used**: The amount of gas consumed by the transaction.
- **Gas Price**: The price per unit of gas, usually set in Gwei (1 Gwei = 0.000000001 ETH).
- **Steps to Calculate Ethereum Transfer Cost:**
- **Transfer Ether using a function** in your smart contract like transfer() or send().
- Check the **Gas Used** using tools like Remix IDE or MetaMask, which provides gas information after the transaction.
- Multiply **Gas Used by Gas Price** to get the final cost of the transfer in ETH.

Example:

- **Gas Used**: 21,000 units (standard for Ether transfers)
- **Gas Price**: 30 Gwei
- **Cost in ETH**:
  21,000 × 30 Gwei = 0.00063 ETH

✓ **Heavy and Light Functions**

Smart contract functions are classified as either "heavy" (high gas consumption) or "light" (low gas consumption) based on the computational load they require.

Heavy Functions:

- **Characteristics**: Perform complex operations such as multiple state changes, loops, or large data manipulations.
- **Gas Consumption**: High.
- **Best Practices**:
  - Minimize the use of heavy functions whenever possible.
  - Break complex logic into smaller functions to improve gas efficiency.

Light Functions:

- **Characteristics**: Perform simple, read-only operations, such as returning data without altering the blockchain state.
- **Gas Consumption**: Low or zero (for view functions).
- **Best Practices**:
  - Use light functions for operations that do not require changes to the blockchain state.
  - Always prefer view functions for read-only data retrieval.

Example:

```
// Heavy function (high gas)
function updateBalance(uint _amount) public {
balances[msg.sender] = balances[msg.sender] + _amount;}
// Light function (low gas, read-only)
function getBalance() public view returns (uint) {return balances[msg.sender];}
```

✓ **Block Limit**

Ethereum blocks have a gas limit, which is the maximum amount of gas that can be consumed by transactions within a single block. This ensures that blocks are processed efficiently.

- **Current Block Gas Limit:**
  - Approximately **30 million gas** per block (as of 2024), but this value may fluctuate slightly.
- **Key Considerations:**
  - If your smart contract consumes too much gas, your transaction may exceed the block gas limit and **fail**.
  - Always aim to reduce gas usage in complex operations to avoid exceeding the block limit.
- **Best Practice:**
  - **Optimize loops**: Avoid unbounded loops and excessive computational steps.
  - **Batch operations**: Split large, gas-intensive operations into multiple smaller transactions.

- ✓ **Opcode Gas Cost**

  Each operation (or **opcode**) in Ethereum has a specific gas cost. Understanding these costs can help optimize your smart contract's gas usage.

- **Common Opcode Gas Costs:**
  - **ADD/SUB (Addition/Subtraction)**: 3 gas.
  - **MUL (Multiplication)**: 5 gas.
  - **SSTORE (Storage operation)**: 20,000 gas (writing a new value) or 5,000 gas (updating a stored value).
  - **CALL (External contract calls)**: 700 gas.
- **Optimization Tips:**
  - **Avoid unnecessary storage operations**: Writing to storage (SSTORE) is costly.
  - **Use arithmetic wisely**: Operations like multiplication and division are more expensive than addition and subtraction.
  - **Minimize external calls**: Calling another contract can be costly due to the gas consumed during interaction.

  Example:

  // High gas usage due to SSTORE operation function updateValue(uint _newValue) public {

  storedValue = _newValue; // SSTORE (costly)}

  // Optimized version using fewer storage writes

  function updateValueOptimized(uint _newValue) public {f i (storedValue != _newValue) {storedValue = _newValue;}}

- ✓ **Non-Payable Functions**

  Functions in Solidity can be marked as **non-payable** using the nonpayable modifier, which prevents the function from accepting Ether.

  Why Use Non-Payable Functions?

- **Prevents accidental Ether transfers**: If your function doesn't require Ether, marking it as nonpayable ensures Ether cannot be mistakenly sent.
- **Security**: Helps to avoid potential vulnerabilities from unintended Ether transfers.

  Functions(role) of non-payable functions

- Non-payable functions are used by default unless marked as payable.
- When a non-payable function receives Ether, the transaction will fail.

  Example:

  // Non-payable function (default)

  function updateData(uint _data) public { storedData = _data;}

  // Payable function (allows receiving Ether)

  function deposit() public payable {balance[msg.sender] += msg.value;}

Practical Activity 2.4.2: Elaboration of storage

Task:

1: You are requested to go in computer lab to apply the elaboration of storage in blockchain.
2: Read the key readings 2.4.2
3: Apply safety precaution
4: Apply the elaboration of storage
5: Present your work to trainer.
6: perform the task provided in application of learning 2.4

Key readings 2.4.2: Elaboration of storage

✓ **Elaboration of Storage**

Storage is one of the most expensive resources in Ethereum smart contracts, so optimizing it can drastically reduce gas costs.

- **Solidity Storage**: Each storage slot is 32 bytes (256 bits). When variables are written to storage, they consume a full slot regardless of the data type.
- **Mapping vs Arrays**: Mappings are more gas-efficient for dynamic key-value storage, while arrays offer better usability for indexed data.

Example:

```
// Storage use in arrays
uint[] public dataArray; // Dynamic array
// Storage use in mapping
mapping(address => uint) public balances; // Mapping of addresses to balances
```

✓ **Smaller Integers, Unchanged Storage Values, Arrays**

- **Smaller Integers:**

Solidity allows you to use smaller data types such as uint8, uint16, or uint32 to save storage space and gas when the values are guaranteed to be within a specific range.

Example:

```
// Larger integer (costlier)
uint256 public largeNumber = 100;
// Smaller integer (cheaper)
uint8 public smallNumber = 100;
```

- **Unchanged Storage Values:**

Avoid rewriting a storage variable if the value hasn't changed. Writing to storage is expensive, while reading from storage is much cheaper.

Example:

// Inefficient: always writes to storage

function setValue(uint _newValue) public {storedValue = _newValue;}

// Efficient: only writes if value changes

function setValueEfficient(uint _newValue) public {if (storedValue != _newValue) {storedValue = _newValue;}}

- **Arrays:**

When working with arrays, use memory instead of storage when temporary data is sufficient. Storage operations are costly compared to memory.

Example:

// More expensive (modifies storage)

uint[] public storageArray; // Cheaper (temporary array in memory)

function tempArrayOperation() public {uint;}

- ✓ **Refunds and Setting to Zero**

Ethereum allows a gas refund when storage values are set to zero. This can help reduce overall transaction costs if used correctly.

Example:

// Setting a storage variable to zero to get a gas refund

function resetBalance() public {balances[msg.sender] = 0; // Gas refund for clearing storage}

- ✓ **ERC20 Transfers**

When handling token transfers in an ERC20-compliant contract, it's essential to minimize storage writes and gas usage. Use the transfer() function efficiently, and avoid redundant updates.

Example:

// ERC20 token transfer example

function transfer(address _to, uint256 _value) public returns (bool) {require(balances[msg.sender] >= _value, "Insufficient balance"); // Avoid redundant storage writes balances[msg.sender] -= _value; balances[_to] += _value;

emit Transfer(msg.sender, _to, _value); return true;}

- ✓ **Storage Cost for Files**

Storing large amounts of data (such as files) on-chain is highly expensive. Instead of storing the actual file data on-chain, store a hash or pointer (e.g., IPFS hash) that references the file off-chain.

Example:

// Storing a hash instead of a large file

bytes32 public fileHash;

// Efficient method: storing file metadata only

function storeFileHash(bytes32 _hash) public {fileHash = _hash;}

- ✓ **Structs and Strings, Variable Packing, Array Length**
- **Structs:**

  When using structs, place smaller data types together to take advantage of **variable packing**. Solidity packs variables into storage slots when possible, reducing gas costs.

  Example:

  // Inefficient struct (uint256 takes up an entire slot)

  struct User {uint256 id;bool isActive;}

  // Optimized struct (variable packing for smaller data types)

  struct PackedUser {uint8 id; bool isActive; uint16 age;}

Practical Activity 2.4.3: Optimization of Memory cost

Task:

1: You are requested to go in computer lab to apply the optimization of memory cost
2: Read the key readings 2.4.3
3: Apply safety precaution
4: Apply the optimization of memory cost
5: Present your work to trainer
6: Perform the task provided in application of learning 2.4

Key readings 2.4.3: Optimization of memory cost

- ✓ **Memory vs Call Data**

  In Solidity, function arguments can be stored in either **memory** or **calldata**.
- **Memory**: Used for temporary storage. The data exists only during function execution, making it more expensive than calldata for read-only operations.
- **Call Data**: Used for function inputs that cannot be modified, and it is cheaper because it prevents copying of data.

  Example:

  // Using memory (can be modified within the function)

  function processDataMemory(uint[] memory data) public {data[0] = 100; // Data in memory can be changed}

  // Using calldata (cannot be modified, cheaper than memory)

  function processDataCalldata(uint[] calldata data) public {// data[0] = 100; // This will throw an error because calldata is immutable}

  Best Practices:

- Use calldata for function parameters when the data is read-only.
- Use memory for temporary variables within the function that need to be modified.

✓ **Mappings vs Arrays**

- **Mappings**: Efficient for storing key-value pairs. Access is constant time O(1), but mappings do not have a length or iterable properties.
- **Arrays**: Useful for ordered, index-based data storage. Arrays are less gas-efficient than mappings for large-scale data operations.

Example:

// Mapping (key-value pair storage)
mapping(address => uint) public balances;
// Dynamic array (ordered storage)
address[] public addressList;

✓ **Freeing Up Unused Storage**

Storage in Ethereum is expensive. To optimize gas costs, clear unused storage by setting values to zero. This may also lead to gas refunds.

Example:

// Function to clear storage and receive a gas refund
function clearBalance() public {balances[msg.sender] = 0; // Gas refund for freeing up storage}

- **Best Practices:**

➢ Always free up unused storage by setting values to zero when no longer needed.
➢ Avoid unnecessary writes to storage, as they are costly.

✓ **Immutable and Constant**

- **Immutable**: Variables that can only be set once at contract deployment. They are more flexible than constant but still save gas because they are not stored in storage.
- **Constant**: Variables that are set at compile time and cannot be changed. They are the cheapest form of storage since they are not written to storage at all.

Example:

// Immutable variable (set once at deployment)
address public immutable owner;
constructor() {
owner = msg.sender; // Set the immutable variable once}
// Constant variable (set at compile time)
uint public constant MAX_SUPPLY = 10000;

- **Best Practices:**

➢ Use immutable for variables that need to be set during contract initialization but never change afterward.

- Use constant for values that are known at compile time and will never change, to save gas.

✓ **Access Modifiers**

Access modifiers restrict access to functions or variables. Common modifiers include public, private, internal, and external.

Example:

```
contract Example {
// Public variable accessible from anywhere
uint public data;
// Private function, only accessible within this contract
function _privateFunction() private view returns(uint) {
return data;}
// External function, can be called from outside the contract only
function externalFunction() external view returns(uint) {return data;}}
```

✓ **Indexed Events**

Events in Solidity are used to log data on the blockchain. Using **indexed** parameters in events allows for efficient searching and filtering of event logs.

Example:

```
// Event with indexed parameters
event Transfer(address indexed from, address indexed to, uint256 value);
function transfer(address _to, uint256 _amount) public {emit
Transfer(msg.sender, _to, _amount);}
```

✓ **Minimizing On-Chain Data**

Storing large amounts of data on-chain is very costly. Use off-chain storage solutions like IPFS or external APIs for large data sets, and store only necessary metadata on-chain.

Example:

```
// Efficient method: storing file metadata or hash on-chain, file off-chain
string public fileHash;
function storeFileHash(string memory _hash) public {fileHash = _hash; // Only
store hash, not the entire file}
```

Points to Remember

- Use **calldata** for read-only external function arguments to save gas.
- Use **memory** for temporary variables that need to be modified inside functions.
- **Mappings** offer constant-time access (O(1)) but can't be iterated.
- **Arrays** allow ordered data and iteration but are more expensive for large data sets
- Minimize unnecessary writes to storage, as they are gas-expensive.
- **immutable** variables can be set once during deployment and save storage costs.
- **constant** variables are fixed at compile-time and are the most gas-efficient.
- **private** restricts function visibility to the contract only.
- **external** functions are callable from outside the contract and save gas when passed complex data types like arrays.

Application of learning 2.4

XYZ Company Ltd, is a company Located in Western province. It works different project based on blockchain technology. It has a problem where there are different smart contract not connected to the wallets and also wallet integration is a problem. As a blockchain developer, you are requested to:

I. Write a simple smart contract where you declare state variables of different data types like uint, bool, string, address, and bytes. Additionally, implement functionality to perform function operations and connect the smart contract to wallets for interaction.

II. Assign values to these variables, make them public so that their values can be read from outside the contract, perform function operations, and ensure wallet integration for reading and writing values.

Learning outcome 2 end assessment

Theoretical assessment

Q1. Read carefully the following statements related to Apply Solidity Basics and Answer by **True** if the statement is correct or by **False** if the statement is incorrect:

i. The Solidity compiler solc is used for deploying smart contracts on Ethereum.
ii. Arrays in Solidity can be of both fixed and dynamic size.
iii. A view function in Solidity can modify the state of a contract.
iv. The modifier keyword in Solidity is used to define state variables.
v. The cost of storage in Ethereum increases when setting variables to zero.

Q2. Circle the letter corresponding with the correct answer

i. **Which of the following best describes the purpose of the Ethereum Virtual Machine (EVM)?**
   a) It processes off-chain transactions.
   b) It executes smart contracts and manages the state of the blockchain.
   c) It handles encryption and key management.
   d) It governs the behavior of decentralized applications (dApps).

ii. **Which of the following is an immutable keyword in Solidity that saves gas costs?**
   a) Memory
   b) constant
   c) public
   d) indexed

iii. **In Solidity, which of the following operations consumes the most gas?**
   a) Reading a state variable
   b) Writing a new state variable
   c) Executing a view function
   d) Calling an external function

iv. **Which access modifier ensures that a function can only be called from within the contract it's defined in?**
   a) public
   b) private
   c) internal
   d) external

Q3. Match the Solidity Concepts **(Column B)** with their corresponding Description **(Column C**). Write the letter of the correct answer in the provided blank space in **Column A.**

Column A Answer	Column B Solidity Concept	Column C Description
1……..	1. Struct	a) Used to connect to the Ethereum network
2……..	2. Metamask Wallet	b) A collection of key-value pairs
3……..	3. Mapping	c) Defines a custom data type in Solidity
4……	4. Visibility and Access Control	d) Controls access to contract functions
		e) Fetching data from a smart contract

Q4. Scenario:

i.) You are working on a Solidity smart contract that includes multiple functions. Some functions require high computational effort (heavy functions), and others only read data (light functions). You want to optimize gas consumption for each function.

Task: Analyze the functions and propose three techniques to reduce gas costs for the contract.

ii.) You have written a smart contract that interacts with a decentralized application (dApp). The contract needs to store a large dataset. What strategies would you employ to reduce storage costs and optimize the use of memory?

Q5. Evaluate the differences between memory and call data in Solidity and explain how you would optimize the use of each for functions involving large data sets.

Practical assessment

Q6. Design a smart contract that logs an event whenever a user transfers tokens. Optimize this contract for gas cost by utilizing Solidity's indexed keyword and other relevant techniques.

Checklist for Assessment

No	**Performance Criteria**	**Indicator to be Checked**	**Yes**	**No**
1	Solidity Environment Setup	Correct installation of Node.js, npm, solc		
2	Use of Solidity Data Types & Variables	Correct use of state variables and data types		
3	Smart Contract Functions	Accurate implementation of function operations		
4	Function Interaction	Successful connection to Metamask wallet		
5	Gas Optimization Techniques	Use of cost-reduction techniques (opcode, memory vs calldata, etc.)		
6	Smart Contract Deployment	Successful deployment of contract on testnet		
7	Use of Events	Efficient use of event logging and indexing		
8	Solidity Code Readability	Clear structure and well-documented code		
Decision				

END

References

A. Sohel Rana, e. a. (2019). Secure Smart Contract Development: A Survey.

Alexander, C. (2019). *Ethereum Cookbook.* O'Reilly Media.

Antonopoulos, A. M. (2018). *Mastering Blockchain Programming: Build, Deploy, and Manage Your Own Blockchain Applications.* O'Reilly Media.

Casey, M. J. (2017). Solidity Programming: A Beginner's Guide.

ConsenSys. (n.d.). Best Practices for Writing Secure Solidity Contracts.

ConsenSys. (n.d.). Optimizing Solidity Contracts for Gas Efficiency.

Egorov, B. Š. (2021). *Solidity Programming Cookbook.* Packt Publishing.

Gavin Wood, e. a. (2014). Solidity: A Programming Language for the Ethereum Blockchain.

Lewis, A. (2017). *Blockchain Basics: A Non-Technical Introduction.* Wiley.

Raggio, G. (2021). *Blockchain Development with Solidity and Ethereum.* Packt Publishing.

Škvorc, B. (2019). Solidity Programming: A Developer's Guide. *Packt Publishing*.

Team, C. (n.d.). *CryptoZombies.* CryptoZombies.

Team, R. I. (n.d.). *Remix IDE.*

Vitalik Buterin, e. a. (2014). Solidity: A Language for Secure Smart Contracts.

Vitalik Buterin, e. a. (2014). Solidity: A Language for Secure Smart Contracts.

Wood, A. M. (2019). *Ethereum Programming: The Definitive Guide.* O'Reilly Media.

A Fult Contract System
Sloldlity
ERC-20
Creating Smart Contracts
Slodity
Tukens Smart Tokens
ERC
Creating Smart Contracts
ERC
Fun:cons Smart Contarts
Applying Smart Contract
AUDIT
721
Security Smort Contracts
Applying Smort Contracts
Deploying

Indicative Contents

3.1 Creating Smart Contracts

3.2 Creating Tokens

3.3 Applying Security of Smart Contracts

3.4 Deploying Smart Contracts

Key Competencies for Learning Outcome 3: Develop smart contract system.

Knowledge	Skills	Attitudes
● Description of Smart Contract ● Explanation of mapping, arrays, structs, and Error handling ● Description of Non-Fungible Token ● Identifying smart-contracts re-entrancy attack ● Description of infrastructure services for blockchain applications ● Description of development blockchain network	● Applying of Solidity programming language ● Creating smart contracts ● Applying security of smart contracts ● Deploying smart contracts ● Selecting of development blockchain network ● Creating infrastructure services for blockchain applications ● Using third-party libraries	● Being Creative in designing blockchain project ● Being Self-Motivated in learning on latest technologies of blockchain ● Being Innovative of blockchain technology ● Teamwork in collaborating with developers, designers, and stakeholders in block chain project

Duration: 45 hrs

Learning outcome 3 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe clearly Smart Contract based on blockchain protocol
2. Explain clearly mapping, arrays, structs, and Error handling based on specific requirements
3. Describe properly Non-Fungible Token based on token standards
4. Apply accurately Solidity programming language based on blockchain technology
5. Apply correctly security of smart contracts based specific requirements
6. Select development blockchain network based on protocol
7. Deploy properly smart contracts based on infrastructure

Resources

Equipment	Tools	Materials
● Computer	● Alchemy or Infura ● Truffle or HardHat ● Browser ● Vscode	● Internet ● Electricity

Indicative content 3.1: Creating smart contract

Duration:10hrs

Practical Activity 3.1.1: Applying Solidity programming language

Task:

1: You are requested to go in computer lab to apply Solidity Programming Language.
2: Read the key readings 3.1.1
3: Apply safety precaution
4: Apply Solidity Programming Language
5: Present your work to trainer.
6: perform the task provided in application of learning 3.1

Key readings 3.1.1.: Apply solidity Programming Language

- ✓ **Advanced Features:**
  - **Inheritance:** Create new contracts based on existing ones to reuse code and functionality.
  - **Custom data types:** Define your own data structures using structs and enums for specific needs.
  - **Events and logging:** Emit events to notify external listeners and use logging for debugging and monitoring.
  - **Storage and memory:** Understand the differences between storage and memory variables for efficient data storage.
  - **Assembly:** Use assembly for low-level optimizations, but with caution due to its complexity.
- ✓ **Best Practices:**
  - **Modularization:** Break down complex contracts into smaller, reusable functions to improve readability and maintainability.
  - **Naming conventions:** Use clear and meaningful names for variables, functions, and contracts to enhance code understanding.
  - **Comments:** Explain the purpose of code sections using comments to aid comprehension and future modifications.
  - **Testing:** Write comprehensive unit tests to ensure the correctness and reliability of your smart contracts.
  - **Security:** Be aware of common security vulnerabilities like reentrancy attacks, integer overflows, and denial-of-service attacks.

Implement appropriate measures to prevent them.

- **Gas optimization:** Minimize gas usage to reduce transaction costs and improve efficiency.

✓ **Advanced Topics:**

- **Delegatecall:** Call functions on another contract without transferring control.
- **Calldata:** Pass data to functions without storing it on the blockchain.
- **Libraries:** Create reusable code modules that can be linked to multiple contracts.
- **Custom errors:** Define custom error messages for specific exceptions.

Example:

pragma solidity ^0.8.0;

contract Voting System {struct Voter {address voter Address; bool has Voted;}

struct Election {string name; uint256 start Time; uint256 end Time; Candidate[] candidates;}

struct Candidate {string name; uint256 voteCount;} // other variables and functions

function vote(uint256 electionId, uint256 candidateIndex) public {

// verify voter eligibility, update ballot, and tally votes }

function getElectionResults(uint256 electionId) public view returns (Candidate[] memory) {// retrieve election results from the blockchain}}

✓ **Advanced Features:**

- **Inheritance:** Create new contracts based on existing ones.
- **Custom data types:** Define your own structs and enums.
- **Events and logging:** Use events to notify external listeners and log information.
- **Storage and memory:** Understand the differences between storage and memory variables.
- **Assembly:** Use assembly for low-level optimizations, but with caution.

✓ **Best Practices:**

- **Modularization:** Break down complex contracts into smaller, reusable functions.
- **Naming conventions:** Use clear and meaningful variable and function names.
- **Comments:** Explain the purpose of code sections using comments.
- **Testing:** Write thorough unit tests to verify code correctness.
- **Security:** Be aware of common vulnerabilities and implement appropriate measures.
- **Gas optimization:** Minimize gas usage to reduce transaction costs.
- **Delegatecall:** Call functions on another contract without transferring control.
- **Calldata:** Pass data to functions without storing it on the blockchain.
- **Libraries:** Create reusable code modules.
- **Custom errors:** Define custom error messages.

Practical Activity 3.1.2: Writing a smart contract

Task:

1: You are requested to go in computer lab to write smart contract.

2: Read the key readings 3.1.2

3: Apply safety precaution

4: Write Smart Contract.

5: Present your work to trainer.

6: perform the task provided in application of learning 3.1

Key readings 3.1.2: Writing a smart contract

- ✓ **Modularization:** Break down complex contracts into smaller, reusable functions and modules to improve readability and maintainability.
- ✓ **Naming Conventions:** Use clear and consistent naming conventions for variables, functions, and contracts to enhance code understanding.
- ✓ **Comments:** Explain the purpose of code sections using comments to aid comprehension and future modifications.
- ✓ **Testing:** Write comprehensive unit tests to verify the correctness and reliability of your smart contracts.
- ✓ **Security:** Be aware of common security vulnerabilities like reentrancy attacks, integer overflows, and denial-of-service attacks. Implement appropriate measures to prevent them.
- ✓ **Gas Optimization:** Minimize gas usage to reduce transaction costs and improve efficiency.
- ✓ **Advanced Features:** Consider using advanced features like inheritance, custom data types, events, and assembly when necessary, but use them judiciously and only when they provide significant benefits.

Example:

```
pragma solidity ^0.8.0;
contract Auction {
struct AuctionItem {
uint256 itemId;
string itemName;
uint256 endTime;
address highestBidder;
uint256 highestBid;}
```

```
        AuctionItem[] public auctionItems;
        event AuctionEnded(uint256 itemId, address winner, uint256 winningBid);
        //  ther functions }
```

Points to Remember

- ✓ **Modularization:**
  - Break down complex contracts into smaller, reusable functions and modules.
  - This improves code readability, maintainability, and testability.
- ✓ **Naming Conventions:**
  - Use clear and descriptive names for variables, functions, and contracts.
  - Consistent naming conventions enhance code understanding.
- ✓ **Gas Optimization:**
  - Minimize gas usage to reduce transaction costs.
  - Use efficient data structures and algorithms.
  - Avoid unnecessary computations.
- ✓ Other information:
  - Use advanced features judiciously and avoid unnecessary complexity.
  - Focus on writing clean, efficient, and secure code.
  - Stay updated on the latest Solidity developments and best practices.

Application of learning 3.1.

Create a Solidity smart contract that implements a simple auction system. The contract should allow users to bid on items, track the highest bidder, and facilitate the transfer of ownership when the auction ends. Use advanced Solidity features like inheritance, custom data types, and events

Indicative content 3.2: creating tokens

Duration: 10 hrs

Practical Activity 3.2.1: implementation of fungible token(ft) standard

Task:

1: You are requested to go in computer lab to implementation of fungible token(ft) standard.
2: Read the key readings 3.2.1
3: Apply safety precaution
4: Implementation of fungible token(ft) standard.
5: Present your work to trainer.
6: Perform the task provided in application of learning 3.2.

Key readings 3.2.1: Implementation of Fungible Token(FT) Standard

- ✓ **ERC-20 Standard:** Adhere to the ERC-20 token standard for compatibility and interoperability with other Ethereum-based applications.
- ✓ **Token Structure:** Define **a Token** struct with fields for the token name, symbol, total supply, and balances. Use appropriate data types (e.g., string for name and symbol, **uint256** for supply and balances).
- ✓ **Token Functions:** Implement core functions:
  - **totalSupply():** Returns the total supply of tokens.
  - **balanceOf(address):** Returns the balance of a specific address.
  - **transfer(address, uint256):** Transfers tokens from one address to another.
  - **transferFrom(address, address, uint256**): Transfers tokens on behalf of another address.
  - **approve(address, uint256):** Approves a spender to spend a certain amount of tokens on the owner's behalf.
  - **allowance(address, address):** Returns the approved amount for a spender.
- ✓ **Event Emitting:** Emit events (e.g., **Transfer, Approval**) to notify listeners of token transfers and approvals.
- ✓ **Math Operations:** Use a SafeMath library or implement custom overflow/underflow checks to ensure safe mathematical operations.
- ✓ **Security:**
  - Prevent transfers to the zero address.

- Guard against reentrancy attacks using a reentrancy guard pattern.
- Implement appropriate access controls and permissions.

✓ **Customizations:** Consider adding custom features like:

- **Minting and Burning:** Allow for the creation and destruction of tokens.
- **Pausing:** Temporarily halt token transfers.
- **Blacklisting:** Prevent certain addresses from participating in transactions.

Example:

```
pragma solidity ^0.8.0;
import "./SafeMath.sol";
contract MyERC20Token is ERC20 {
using SafeMath for uint256;
string public name = "My Token";
string public symbol = "MYT";
uint256 public totalSupply = 1000000;
mapping(address => uint256) private _balances;
mapping(address => mapping(address => uint256)) private _allowances;
constructor() {_balances[_msgSender()] = totalSupply;
emit Transfer(address(0), _msgSender(), totalSupply);}function
balanceOf(address account) public view override returns (uint256) {return
_balances[account];
}function
transfer(address recipient, uint256 amount) public override returns (bool)
{_transfer(_msgSender(), recipient, amount);
 return true;}function approve(address spender, uint256 amount) public override
returns (bool)
 {_approve(_msgSender(), spender, amount);return true;}function transfer
From(address sender, address recipient, uint256 amount) public override returns
(bool)_transfer(sender, recipient, amount);
_approve(sender,_msgSender(),
_allowances[sender][_msgSender()].sub(amount,"ERC20:insufficient
allowance"));
return true;
}
 function _transfer(address sender, address recipient, uint256 amount) internal
{require(sender != address(0),
"ERC20: transfer from the zero address");
require(recipient!=address(0),
"ERC20:transfer to the zero address");
_balances[sender] = _balances[sender].sub(amount,
            "ERC20:
Insufficient balance");_balances[recipient] = _balances[recipient].add(amount);
emit Transfer(sender, recipient, amount);
}
function _approve(address owner, address spender, uint256 amount) internal {
require(owner != address(0),"ERC20: approve from the zero address");
 require(spender != address(0), "ERC20: approve to the zero address");
          _allowances[owner][spender] = amount;
          emit Approval(owner, spender,

           amount);

          }}
```

Practical Activity 3.2.2: Implementation of Non Fungible Token (NFT) Standard

Task:

1: You are requested to go in computer lab to Implementation of Non-Fungible Token (NFT).
2: Read the key readings 3.2.2
3: Apply safety precaution
4: Implementation of Non-Fungible Token (NFT).
5: Present your work to trainer.
6: perform the task provided in application of learning 3.2

Key readings 3.2.2: Implementation of Non Fungible Token (NFT) Standard

- ✓ **ERC721 Standard:** Adhere to the ERC721 token standard for NFTs, which defines core functions like **mint**, **transfer**, **balanceOf**, and **ownerOf.**
- ✓ **Auction Mechanism:** Implement a mechanism for creating auctions, placing bids, and ending auctions. Consider factors like auction duration, reserve prices, and bidding increments.
- ✓ **Token Ownership:** Use the ERC721 library to manage NFT ownership and transfers. Implement functions for creating new NFTs, transferring ownership, and checking ownership.
- ✓ **Payment Handling:** Integrate a payment system (e.g., using a stablecoin) to

facilitate transactions. Consider factors like fees, payment methods, and dispute resolution.

- ✓ **Security:** Implement security measures to prevent fraud, unauthorized access, and other vulnerabilities. This may include using access controls, preventing reentrancy attacks, and safeguarding user funds.
- ✓ **User Interface:** Design a user-friendly interface that allows users to browse NFTs, place bids, and manage their collections. Consider factors like navigation, search functionality, and mobile compatibility.
- ✓ **Scalability:** If your marketplace is expected to handle a large number of users and transactions, consider scalability solutions like off-chain scaling or sharding.
- ✓ **Interoperability:** Explore ways to integrate your marketplace with other platforms or protocols to expand its reach and functionality.
- ✓ **Legal and Regulatory Compliance:** Ensure that your marketplace complies with relevant laws and regulations, such as those related to money laundering, consumer protection, and intellectual property.

- ✓ **Additional Considerations:**
- **NFT Metadata:** Provide detailed metadata for each NFT, including the title, description, image URL, and any additional relevant information.
- **Royalties:** Implement a royalty system to allow creators to earn a percentage of future sales of their NFTs.
- **Secondary Market:** Consider facilitating a secondary market where users can buy and sell NFTs from each other.
- **Marketing and Promotion:** Develop a marketing strategy to attract users to your marketplace and promote your NFTs.

Example:

```
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
contract NFTMarketplace is ERC721 {

//other variables and functions ...

Function createAuction(string memory _tokenURI, uint256 _startingPrice,
uint256 _endTime) public {
// ... auction logic ...}
function placeBid(uint256 _itemId, uint256 _bidAmount) public {
// ... bidding logic ...}
function endAuction(uint256 _itemId) public {
// ... auction ending logic ...}}
```

Points to Remember

- ✓ **Define your NFT data structure:**

An NFT is typically represented by a unique identifier (often called a token ID) and a set of attributes that define its properties. You can use a struct to define the data structure for your NFTs, specifying the attributes that are relevant to your use case. For example, an NFT representing a piece of digital art might have attributes such as the title, artist, and image data.

- ✓ **Consider using a library:**

There are several libraries available that can simplify the process of implementing NFTs. These libraries often provide helper functions and ensure compliance with common NFT standards. Some popular libraries include OpenZeppelin's **openzeppelin-contracts-token** and ERC721.

- ✓ **Deploy your NFT contract:**

Once you are satisfied with your testing, deploy your NFT contract to the Ethereum blockchain. You can use a variety of tools and platforms to deploy your contract, such as Remix, Truffle, or Ganache.

- ✓ **Token Structure:**
- Define a **Token** struct with fields for the token name, symbol, total supply, and balances.
- Use appropriate data types (e.g., **string** for name and symbol, **uint256** for supply and balances).
- ✓ **Token Functions:**
- **totalSupply():** Returns the total supply of tokens.
- **balanceOf(address):** Returns the balance of a specific address.
- **transfer(address, uint256):** Transfers tokens from one address to another.
- **transferFrom(address, address, uint256):** Transfers tokens on behalf of another address.
- **approve(address, uint256):** Approves a spender to spend a certain amount of tokens on the owner's behalf.
- **allowance(address, address):** Returns the approved amount for a spender.

Application of learning 3.2.

Create a Solidity smart contract that implements a simple NFT marketplace. The contract should allow users to list NFTs for sale, place bids, and purchase NFTs.

Indicative content 3.3: Applying security of Smart contract

Duration: 10 hrs

Practical Activity 3.3.1: Protection of Smart contracts re-entrancy

Task:

1: You are requested to go in computer lab to protect smart contract re-entrancy.

2: Read the key readings 3.3.1

3: Apply safety precaution

4: Protect smart contract re-rentrancy

5: Present your work to trainer.

6: perform the task provided in application of learning 3.3

Key readings 3.3.1: Protection of Smart Contracts against re-entrancy

✓ **Reentrancy Attacks:**

- **Definition:** A reentrancy attack occurs when a smart contract calls another contract that can modify the state of the original contract before it finishes executing.
- **Vulnerability:** Contracts that allow external calls within a function are susceptible to reentrancy attacks.

✓ **Prevention Techniques:**

- **Reentrancy Guards:** Use a reentrancy guard pattern to prevent recursive calls within a function. This involves setting a lock flag before executing the function and resetting it after the function completes.
- **Check Effects, Interact Later (CEIL):** Separate the effect of a function (modifying the contract state) from the interaction with external contracts. This helps prevent reentrancy attacks by ensuring that external calls are made after the state has been updated.
- **Use Libraries:** Consider using libraries that implement reentrancy guards and other security measures.

Example:

```
pragma solidity ^0.8.0;
contract ReentrancyGuard {
bool private _locked;modifier nonReentrant() {require(_locked == false,
"ReentrancyGuard: reentrant call");locked = true;_locked = false;}}
contract MyContract is ReentrancyGuard {// ... contract logic ...
function withdrawFunds() public nonReentrant {// ... withdraw funds}}
```

Practical Activity 3.3.2: Securing Smart Contract Using Escrow Service

Task:

1: You are requested to go in computer lab to write secure smart contract using escrow service.

2: Read the key readings 3.3.2

3: Apply safety precaution

4: Write secure smart contract using escrow service

5: Present your work to trainer.

6: perform the task provided in application of learning 3.3

Key readings 3.3.2: Securing Smart Contract Using Escrow Service

✓ **Escrow Service Overview:**

- An escrow service is a neutral third-party that holds funds or assets on behalf of two parties involved in a transaction.
- It ensures that the transaction is completed as agreed upon before releasing the funds or assets.
- Escrow services are commonly used in online marketplaces, real estate transactions, and other scenarios where trust and security are important.

✓ **Key Components of an Escrow Service Smart Contract:**

- **Escrow Creation:** A function that allows parties to create an escrow transaction by depositing funds and specifying the terms of the agreement.
- **Fund Release:** A function that releases the funds to the appropriate party once the agreed-upon conditions are met.
- **Dispute Resolution:** A mechanism for handling disputes if the parties cannot agree on the terms of the transaction or if one party fails to fulfill their obligations.
- **Time Constraints:** Deadlines for parties to fulfill their obligations and for the escrow to be completed.
- **Transparency:** A way for parties to view the status of the escrow transaction and the terms of the agreement.

✓ **Security Considerations:**

- **Secure Fund Holding:** Implement robust mechanisms to prevent unauthorized access or manipulation of the escrow funds. Consider using multi-sig wallets or time-lock contracts.
- **Dispute Resolution:** Define clear dispute resolution procedures and consider incorporating an off-chain dispute resolution mechanism or using an oracle to resolve disputes.
- **Time Constraints:** Set clear deadlines for the fulfillment of obligations by both parties. Implement automatic release of funds or cancellation of the contract if deadlines are not met.
- **Transparency:** Ensure that the contract terms, transaction history, and dispute resolution process are transparent to all parties involved.
- **Security Audits:** Have your escrow contract audited by security experts to identify and address potential vulnerabilities.

Practical Activity 3.3.3: Usage of third-party libraries

Task:

1: You are requested to go in computer lab to use third-party ibraries.

2: Read the key readings 3.3.3

3: Apply safety precaution

4: Use third-party ibraries

5: Present your work to trainer.

6: perform the task provided in application of learning 3.3

Key readings 3.3.3: Usage of third-party libraries

✓ **Third-Party Libraries:**

- **Definition:** Pre-written code modules that provide specific functionalities or solve common problems.
- **Benefits:**
  - Save development time and effort.
  - Ensure code quality and security.
  - Access specialized features and expertise.
  - Promote code reusability and standardization.

✓ **Considerations:**

- **Security:** Carefully evaluate the security of third-party libraries. Choose reputable sources and review code audits.

- **Compatibility:** Ensure compatibility with your project's requirements, including Solidity version and other dependencies.
- **Licensing:** Understand the licensing terms of the library to avoid legal issues.
- **Maintenance:** Stay updated with library updates and security patches.
- **Dependencies:** Be aware of any additional dependencies introduced by the library.

✓ **Popular Libraries:**

- **OpenZeppelin:** Provides a comprehensive suite of contracts for common use cases, including token standards, governance, and security.
- **DappTools:** Offers tools for development, testing, and debugging smart contracts.
- **Chainlink:** Provides decentralized oracle networks for accessing real-world data and triggering smart contract actions.
- **Aave:** A lending protocol that offers interest rates and collateralization.
- **Uniswap:** A decentralized exchange protocol for trading ERC-20 tokens.

✓ **Best Practices:**

- **Research and Selection:** Carefully research and select libraries that meet your project's needs and are well-maintained.
- **Integration:** Follow the library's documentation and examples for proper integration into your contract.
- **Testing:** Thoroughly test your contract after integrating the library to ensure it functions as expected.
- **Security Review:** Conduct security audits to identify potential vulnerabilities introduced by the library.
- **Updates:** Stay updated with library updates and security patches to maintain compatibility and security.

Example:

```
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
contract MyToken is ERC20 {constructor()
{super("My Token", "MYT");}}
```

Points to Remember

✓ **Third-party libraries can be a valuable resource for developers.** They can provide a wide range of functionality, including security, payment processing, and data management.

✓ **When choosing a third-party library, it is important to carefully evaluate its security, reliability, and compatibility with your project requirements.**

- ✓ **Third-party libraries should be used judiciously and only when they are necessary.** Overreliance on third-party libraries can introduce additional dependencies and security risks.
- ✓ **It is important to thoroughly test any third-party library you use before integrating it into your smart contract.** This will help ensure that the library is secure and functions as expected.
- ✓ **When using third-party libraries, it is important to be aware of their licensing terms and conditions.** Some libraries may be open source, while others may be proprietary and require a license fee.

Application of learning 3.3.

Create a Solidity smart contract for an escrow service that facilitates transactions between buyers and sellers. The contract should hold funds in escrow until the seller fulfills their obligations, and then release the funds to the seller. Implement necessary security measures and consider best practices for escrow service smart contracts.

Indicative content 3.4: Deploying Smart Contract

Duration:15 hrs

Theoretical Activity 3.4.1: Selection of development blockchain network

Tasks:

1: Introduce the activity by asking trainees to answer the following questions:

i. What are the key factors to consider when selecting a blockchain network for development?

ii. What are the differences between public, private, and consortium blockchains?

iii. How does the choice of consensus mechanism impact a blockchain network's characteristics?

iv. What are the key advantages and disadvantages of Ethereum?

v. What are the key advantages and disadvantages of Bitcoin?

vi. What are the key advantages and disadvantages of Hyperledger Fabric?

vii. What are the key advantages and disadvantages of Solana?

viii. How do the characteristics of a blockchain network influence the choice of use cases?

ix. What factors should be considered when selecting a blockchain network for a specific use case?

2: Write your findings on paper or flipchart

3: Present your findings to the trainer or colleagues

4: For more clarification read the key readings 3.4.1.

Key readings 3.4.1.: Selection of development blockchain network

- ✓ **Consensus Mechanism:** Evaluate the security, scalability, and energy efficiency of different consensus mechanisms (e.g., Proof of Work, Proof of Stake, Delegated Proof of Stake).
- ✓ **Transaction Speed and Cost:** Consider the network's transaction throughput and fees, especially if your application requires fast and affordable transactions.
- ✓ **Smart Contract Capabilities:** Assess the network's support for smart contracts, including programming languages, execution environment, and gas costs.
- ✓ **Community and Ecosystem:** Evaluate the size and activity of the network's community, as well as the availability of tools, resources, and third-party integrations.

- ✓ **Scalability:** Consider the network's ability to handle increasing transaction volumes and maintain performance.
- ✓ **Privacy and Confidentiality:** Assess the network's privacy features and whether it can meet your data confidentiality requirements.
- ✓ **public, private, and consortium blockchains**
- **Public blockchains:** Accessible to anyone, highly decentralized, and transparent.
- **Private blockchains:** Controlled by a single organization or consortium, offering more control but less decentralization.
- **Consortium blockchains:** Controlled by a group of organizations, providing a balance between decentralization and control.
- ✓ **Impact consensus mechanism on blockchain network's characteristics**
- **Proof of Work (PoW):** Secure but energy-intensive, suitable for public blockchains.
- **Proof of Stake (PoS):** More energy-efficient but potentially less secure than PoW, suitable for both public and private blockchains.
- **Delegated Proof of Stake (DPoS):** Combines elements of PoW and PoS, offering a balance between security and efficiency.
- ✓ **advantages and disadvantages of Ethereum**
- **Advantages:** Large community, rich ecosystem, and extensive smart contract capabilities.
- **Disadvantages:** Scalability limitations and high transaction fees.
- ✓ **Advantages and disadvantages of Bitcoin**
- **Advantages:** Strong security and decentralization.
- **Disadvantages:** Limited smart contract capabilities and slow transaction speeds.
- ✓ **Advantages and disadvantages of Hyperledger Fabric**
- **Advantages:** Customizable, scalable, and suitable for enterprise use cases.
- **Disadvantages:** Less decentralized than public blockchains.
- ✓ **Advantages and disadvantages of Solana**
- **Advantages:** High transaction speed and scalability.
- **Disadvantages:** Relatively new and less mature compared to established networks.
- ✓ **Characteristics of a blockchain network influence the choice of use cases**
- **Public blockchains:** Suitable for decentralized applications, digital assets, and transparent governance.
- **Private blockchains:** Suitable for enterprise use cases, supply chain management, and financial services.
- **Consortium blockchains:** Suitable for industry-specific applications, regulatory compliance, and collaborative projects.
- ✓ **factors to be considered when selecting a blockchain network for a specific use case**

- **Security requirements:** Choose a network that can meet your security needs, especially if you are handling sensitive data.
- **Scalability:** Consider the expected transaction volume and growth potential of your application.
- **Cost:** Evaluate the transaction fees and other costs associated with using the network.
- **Community and ecosystem:** Consider the availability of tools, resources, and third-party integrations.
- **Regulatory compliance:** Ensure that the network complies with relevant regulations and standards.

Practical Activity 3.4.2: Create infrastructure services for block chain application

Task:

1: You are requested to go in computer lab to create infrastructure service for blockchain application.

2: Read the key readings 3.4.2

3: Apply safety precaution

4: Create infrastructure service for blockchain application

5: Present your work to trainer.

6: Perform the task provided in application of learning 3.4

Key readings 3.4.2: Create infrastructure services for blockchain application

✓ **Infrastructure Services Overview:**

- **Definition:** Essential components that provide the foundation for building and deploying blockchain applications.
- **Key Services:**

➢ **Node Hosting:** Running full nodes on your own hardware or using cloud-based solutions.

➢ **Blockchain-as-a-Service (BaaS):** Pre-configured platforms that handle blockchain infrastructure and management.

➢ **Data Centers and Cloud Computing:** Providing computing resources, storage, and networking for blockchain applications.

- **Oracles:** Connecting blockchain networks to external data sources and real-world events.
- **Developer Tools and Frameworks:** Simplifying development and deployment processes.
- **Factors to Consider:**
- **Scalability:** The ability to handle increasing transaction volumes and user growth.
- **Security:** Robust security measures to protect against attacks and ensure data integrity.
- **Cost-Efficiency:** Balancing performance with cost-effectiveness.
- **Ease of Use:** User-friendly interfaces and tools for developers.
- **Integration:** Compatibility with your chosen blockchain network and development tools.
- ✓ **Popular Infrastructure Providers:**
- **Amazon Web Services (AWS):** Offers a wide range of blockchain-related services, including EC2, S3, and Lambda.
- **Microsoft Azure:** Provides a cloud platform with support for various blockchain networks and tools.
- **Google Cloud Platform (GCP):** Offers blockchain-specific services like BigQuery and Cloud Functions.
- **Infura:** A specialized blockchain infrastructure provider focusing on Ethereum and other compatible networks.
- **Alchemy:** Another popular blockchain infrastructure provider with a focus on developer tools and scalability.
- ✓ **Deployment Process:**
- **Choose an Infrastructure Provider:** Select a provider that aligns with your project's requirements and budget.
- **Set Up Your Environment:** Configure your infrastructure, including network settings, storage, and security measures.
- **Deploy Smart Contracts:** Use development tools and command-line interfaces to deploy your smart contracts to the chosen blockchain network.
- **Connect to Your Application:** Integrate your application with the deployed smart contracts using APIs or libraries.
- ✓ **Additional Considerations:**
- **Monitoring and Maintenance:** Continuously monitor your infrastructure for performance, security, and availability issues.
- **Cost Optimization:** Implement strategies to optimize your infrastructure costs, such as scaling resources dynamically.
- **Compliance:** Ensure compliance with relevant regulations and standards, especially for financial or healthcare applications.

Points to Remember

- Infrastructure services are essential for building and deploying blockchain applications. They provide the foundation for running and interacting with smart contracts, as well as connecting to other blockchain networks.
- There are many different types of infrastructure services available, including **cloud-based services, dedicated servers,** and **blockchain-specific platforms.**
- When choosing an infrastructure service, it is important to consider factors such as security, scalability, cost, and ease of use.
- Some popular infrastructure services for blockchain applications include Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and Infura.
- Once you have selected an infrastructure service, you will need to deploy your smart contract to the network. This typically involves deploying the contract bytecode and initializing it with the appropriate parameters.
- After deploying your smart contract, you can interact with it using various tools and interfaces, such as web browsers, mobile apps, or command-line interfaces.

Practical Activity 3.4.3: Deploy Smart contract

Task:

1: You are requested to go in computer lab to deploy smart contract.
2: Read the key readings 3.4.3
3: Apply safety precaution
4: Deploy Smart Contract
5: Present your work to trainer.
6: perform the task provided in application of learning 3.4

Key readings 3.4.3: Deploy Smart Contract

✓ **Development Environment:**

- **Choose a suitable IDE or development environment:** Remix, Visual Studio Code, Truffle, Hardhat, etc.
- **Install necessary tools:** Solidity compiler, Node.js, and other dependencies.

✓ **Smart Contract Development:**

- **Write Solidity code:** Define the contract's functions, variables, and logic.
- **Compile the contract:** Use the Solidity compiler to convert your code into bytecode.

- **Test the contract:** Use testing frameworks like Truffle or Hardhat to simulate interactions and identify issues.

✓ **Blockchain Network Selection:**

- **Consider factors:** Security, scalability, transaction fees, and community support.
- **Popular options:** Ethereum, Binance Smart Chain, Polygon, etc.

✓ **Wallet and Account Setup**

- **Create a wallet:** Use a hardware or software wallet to store your private key.
- **Obtain a network address:** Get an address for the chosen blockchain network.

✓ **Deployment:**

- **Use a deployment tool or directly interact with the blockchain network:**
  - **Truffle:** truffle deploy
  - **Hardhat:** npx hardhat run scripts/deploy.js
  - **Web3.js or Ethers.js:** Directly interact with the network using JavaScript libraries.
- **Provide necessary parameters:** Contract address, transaction fees, and gas limit.

✓ **Testing and Verification:**

- **Test the deployed contract:** Ensure it functions as expected and doesn't have vulnerabilities.
- **Verify the contract's address and transaction hash:** Use blockchain explorers to view the deployment details.

✓ **Monitoring and Maintenance:**

- **Monitor contract activity:** Track transactions, gas usage, and potential issues.
- **Update the contract:** If necessary, deploy a new version with bug fixes or added features.

✓ **Best Practices:**

- **Version control:** Use Git or other version control systems to track changes and collaborate.
- **Security audits:** Conduct security audits to identify vulnerabilities.
- **Documentation:** Document your contract's functionality, usage, and security considerations.
- **Testing:** Write comprehensive unit and integration tests.
- **Backup:** Regularly back up your private key and contract data.

Example (using Truffle):

```
# Compile the contract
truffle compile
# Deploy to the Ethereum mainnet
truffle deploy --network mainnet
```

Points to Remember

- Development Environment:
- Choose a suitable development environment with tools like Solidity compiler, testing frameworks (Truffle, Hardhat), and IDEs (Remix, Visual Studio Code).
- Compilation:
- Compile your Solidity code into bytecode, which is the machine-readable format for smart contracts.
- **Network Selection:**
- Decide on the blockchain network where you want to deploy your contract (e.g., Ethereum, Binance Smart Chain, Polygon).
- Consider factors like fees, security, and community support.
- **Deployment:**
- Use a deployment tool or directly interact with the blockchain network to deploy the compiled bytecode.
- Provide necessary parameters and transaction fees.

Application of learning 3.4.

You are tasked with developing a decentralized application (dApp) on the Ethereum blockchain. What infrastructure services would you choose, and how would you deploy your smart contracts?

Learning outcome 3 end assessment

Theoretical assessment

Q1. Read carefully the following statements related to the development of smart contract and answer by **True** if the statement is correct or by **False** if the statement is incorrect.

I. Solidity supports error handling through the use of require, assert, and revert statements.
II. The modifier keyword is not used for inheritance but for altering function behaviour.
III. ERC20 tokens are fungible, meaning they are interchangeable.
IV. The ERC721 standard is used for Non-Fungible Tokens (NFTs), which are unique.
V. Re-entrancy attacks occur when a function calls another contract before resolving its state, which can lead to malicious exploits.
VI. Ganache is a local blockchain network, not a public network like mainnet or testnet.

Q2. Circle the letter corresponding with the correct answer:

i. **Which of the following is a key feature of Solidity?**
a) Automatic Memory Allocation
b) Strong Type Checking
c) Automatic Garbage Collection
d) Dynamic Scope Resolution

ii. **Which of the following is part of the ERC20 token standard?**
a) transfer()
b) mint()
c) burn()
d) increment()

iii. **Which network should you choose for deploying a smart contract in a production environment?**
a) Ganache
b) Rinkeby
c) Mainnet
d) Hardhat

iv. **Which third-party library can be used to secure smart contracts against overflow and underflow attacks?**
a) OpenZeppelin
b) Chainlink
c) Truffle
d) Mocha

Q3. Match the Solididty Concepts**(Column B)** with their corresponding Description **(Column C).** Write the letter of the correct answer in the provided blank space in **Column A.**

Column A	**Column B**	**Column C**
Answer	**Solidity Concept**	**Description**
1……..	1. Re-entrancy Attack	a. A fungible token standard where each token is identical to another
2……..	2. Ganache	b. A vulnerability where external calls can repeatedly access incomplete state
3……..	3. OpenZeppelin	c. A local blockchain network used for development and testing
4……	4. d) ERC20 Token	d. A library for secure smart contract development including SafeMath
		e. A Hybrid blockchain network used for development and testing

Q4. Scenario:

You are writing a smart contract that manages a list of user balances using mapping. You are required to add a function that transfers balance from one user to another. How would you apply error handling to ensure the sender has enough balance to perform the transaction?

Q5. Scenario:

You are tasked with creating a fungible token (ERC20) for a decentralized exchange (DEX) on the Ethereum network. What steps would you follow to write and deploy this ERC20 contract, and how would you test it using chai and mocha?

Practical assessment

Q1: Evaluate the pros and cons of using the ERC721 vs ERC1155 standards for NFTs**.**

Q2: Create an optimized smart contract for an NFT marketplace using the ERC721 standard. Contract Features:

- ✓ Create a marketplace where users can mint NFTs.
- ✓ The contract should support buying and selling NFTs.
- ✓ Use indexed events for tracking NFT transfers.

END

References

A. Sohel Rana, e. a. (2019). Secure Smart Contract Development: A Survey.

Alexander, C. (2019). *Ethereum Cookbook.* O'Reilly Media.

Antonopoulos, A. M. (2018). *Mastering Blockchain Programming: Build, Deploy, and Manage Your Own Blockchain Applications.* O'Reilly Media.

ConsenSys. (n.d.). Best Practices for Writing Secure Solidity Contracts.

ConsenSys. (n.d.). Building Scalable DApps on Ethereum.

ConsenSys. (n.d.). Creating Tokenized Assets on the Ethereum Blockchain.

ConsenSys. (n.d.). Designing Secure Smart Contracts: A Practical Guide.

ConsenSys. (n.d.). Optimizing Solidity Contracts for Gas Efficiency.

Egorov, B. Š. (2021). *Solidity Programming Cookbook.* Packt Publishing.

Lewis, A. (2017). *Blockchain Basics: A Non-Technical Introduction.* Wiley.

Raggio, G. (2021). *Blockchain Development with Solidity and Ethereum.* Packt Publishing.

Škvorc, B. (2019). *Solidity Programming: A Developer's Guide .* Packt Publishing.

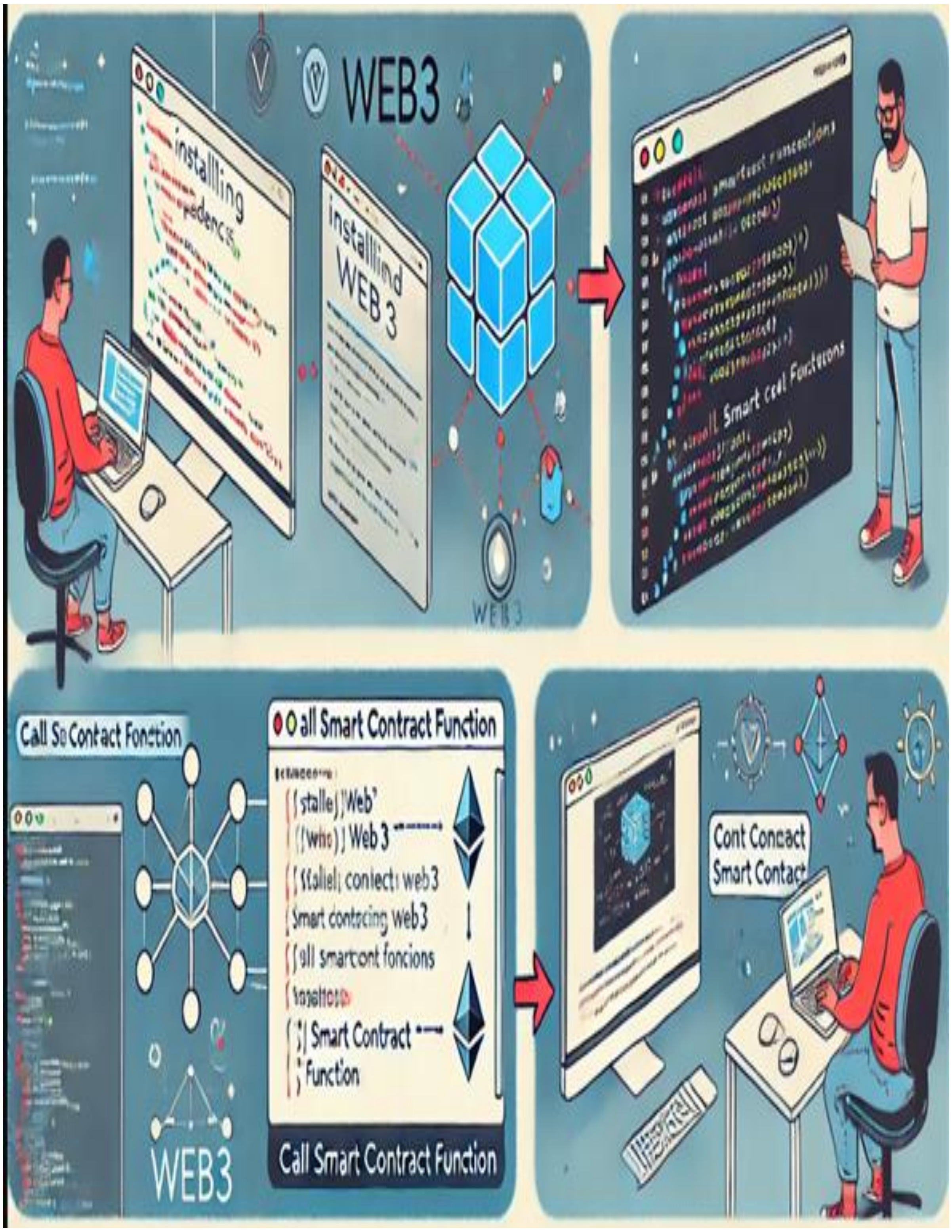
Team, C. (n.d.). *Chainlink Documentation*. Retrieved from https://docs.chain.link/

Team, O. (n.d.). *OpenZeppelin Contracts:*. Retrieved from https://openzeppelin.com/contracts/

Team, T. (n.d.). *Truffle Framework*. Retrieved from https://archive.trufflesuite.com/

Vitalik Buterin, e. a. (2014). Solidity: A Language for Secure Smart Contracts.

Woo, A. M. (2019). *Ethereum Programming: The Definitive Guide.* O'Reilly Media.


WEB3
installing
installind WEB 3
Smart ceal Functions
Call S Contact Fonction
all Smart Contract Function
Smart Contract
Function
Call Smart Contract Function
WEB3
Cont Concact Smart Contact

Indicative Contents **4.1 Installing Web3 Dependencies** **4.2 Connecting Smart Contract** **4.3 Use of Function**

Key Competencies for Learning Outcome 4: Apply Frontend Intergration

Skills	Knowledge	Attitudes
● Description of contract address ● Explanation of Application Binary Interface(ABI) ● Description of faucet balance ● Description of production bundles ● Explanation of function usage	● Installing web3 dependencies ● Installing contract extension in browser for development ● Creating wallet ● Connecting to smart contract using keys ● Creating instance of smart contract ● Consuming smart contract functions ● Deploying web3 frontend	● Teamwork in collaborating with developers, designers, and stakeholders in block chain project ● staying up to date with the latest advancements in blockchain protocols, frameworks, and tools. ● Being resilient and patient while working methodically through challenges

Duration: 25hrs

Learning outcome 4 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Install properly Web3 dependencies based on versions.
2. Create correctly Wallet based on protocol.
3. Create correctly Smart contracts based on keys.
4. Instance appropriately smart contract based on deployed version.
5. Consume properly Smart contract functions based on deployed version.
6. Deploy effectively Web3 frontend based on specific requirements.
7. Describe clearly Smart contract address based on deployed version.
8. Explain clearly Faucet balance based in protocol.

Resources

Equipment	**Tools**	**Materials**
● Computer	● V**S**Code ● Web3.js and ether.js ● Remix, Truffle, and Hardhat	● Electricity ● Internet

Indicative content 4.1: installing web3 dependencies

Duration: 5 hrs

Theoretical Activity 4.1.1: Configure Contract network

Tasks:

1: Introduce the activity by asking trainees to answer the following questions:

I. What is the primary purpose of configuring a contract network?
II. What are the key parameters to configure when interacting with a blockchain network?
III. How can you optimize transaction costs on a blockchain network?
IV. What are some common challenges faced when configuring contract networks?
V. What are some best practices for configuring contract networks?
VI. What is the role of a node provider in blockchain networks?
VII. How can you choose the right node provider for your needs?
VIII. What are some popular node providers for Ethereum?
IX. What is the importance of using a reliable node provider?
X. How can you monitor the performance of your contract network configuration?

2: Write your findings on paper or flipchart

3: Present your findings to the trainer or colleagues

4: For more clarification read the key readings 4.1.1.

Key readings 4.1.1.: Configure Contract network

A **contract network** refers to a block chain-based system where smart contracts are deployed and executed within a decentralized environment. In such a network, the **smart contracts** act as self-executing code with predefined rules and logic, allowing parties to interact and enforce agreements without intermediaries.

✓ Key features of a **contract network** include:

- **Decentralization**: The network operates on a distributed blockchain, with no central authority controlling the execution of contracts.
- **Smart Contracts**: Self-executing contracts with code that automatically enforces and executes terms when specific conditions are met.

- **Trustless Transactions**: Participants in the network can engage in transactions or agreements without needing to trust each other or any third party. The contract ensures fairness and adherence to the terms.
- **Transparency and Security**: All contract interactions are recorded on the block chain, ensuring transparency and security through cryptographic verification.
- **Automation**: Contracts are executed automatically when conditions are met, reducing the need for manual intervention, middlemen, or legal enforcement.

✓ **Primary purpose of configuring a Contract network**

- To establish a connection between your smart contract and the block chain network.

✓ **The key parameters to configure when interacting with a block chain network**

- Gas limit, gas price, network ID, and provider URL.

✓ **To optimize transaction costs on a block chain network**

- Adjust gas limits and prices, batch transactions, and optimize smart contract code.
- **Common challenges faced when configuring contract networks**

➢ Network congestion
➢ security vulnerabilities
➢ compatibility issues.

✓ **Best practices for configuring contract networks**

- Thoroughly test your network configuration, use a secure development environment, and stay updated with network changes.

✓ **Role of a node provider in blockchain networks?**

- To provide access to the network and its services.

✓ **To choose the right node provider for your needs?**

- Consider factors such as reliability, performance, cost, and features.

✓ **Popular node providers for Ethereum**

- Infura, Alchemy, and QuikNode.

✓ **Importance of using a reliable node provider?**

- A reliable node provider ensures stable connections and minimizes downtime.

✓ **To monitor the performance of your contract network configuration**

- Use tools to track transaction times, gas usage, and network congestion.

✓ **Network Selection**

- **To consider network consider the following factors:**

➢ Security,
➢ Scalability
➢ transaction fees
➢ community support.

- **Popular options:** Ethereum, Binance Smart Chain, Polygon, etc.

- **Research and compare:** Evaluate different networks based on your project's needs.

✓ **Node Provider:**

A **node provider** is a service or organization that runs and maintains blockchain nodes, allowing developers and applications to access a blockchain network without having to operate their own infrastructure. Blockchain nodes are essential for validating transactions, maintaining the network's ledger, and ensuring decentralization.

Node providers offer access to these nodes via APIs and other interfaces, making it easier for developers to interact with the blockchain.

✓ **Key Roles and Features of a Node Provider:**

- **Blockchain Connectivity**: Node providers give developers access to blockchain networks (like Ethereum, Bitcoin, etc.) through API endpoints, enabling them to send transactions, query data, and interact with smart contracts.
- **Infrastructure Management**: Instead of running a full node (which can be resource-intensive), developers rely on node providers to manage the technical aspects of running and maintaining blockchain nodes. This includes updating software, ensuring uptime, and securing the infrastructure.
- **Decentralized Applications (DApps)**: DApp developers use node providers to interact with blockchain networks without handling the complexities of node setup, allowing them to focus on building their applications.
- **Scalability**: Node providers often offer scalable solutions, handling a large number of API requests and ensuring fast and reliable interactions with the blockchain.
- **Popular Node Providers**: Examples include **Infura**, **Alchemy**, **QuickNode**, and **Chainstack**, which provide services to interact with Ethereum, Binance Smart Chain, Polygon, and other blockchain networks.

✓ To choose a reliable node provider Consider factors like

- Reputation
- Performance
- pricing.

✓ Popular node provider options are **Infura, Alchemy, QuikNode.**

✓ To set up a connection of node provider Follow the provider's instructions to establish a connection to the network.

✓ **Configuration Parameters of web3 includes:**

- **Gas limit:** The maximum amount of gas a transaction can consume.
- **Gas price:** The amount of Ether paid per unit of gas.
- **Network ID:** The unique identifier of the blockchain network.
- **Provider URL:** The endpoint for connecting to the network.

- ✓ **Optimization of web3 include**
- **Adjust gas settings:** Experiment with different gas limits and prices to find the optimal balance.
- **Batch transactions:** Combine multiple transactions into a single batch to reduce costs.
- **Optimize smart contract code:** Write efficient code to minimize gas usage.
- ✓ **Testing web3 application:**

- **Test network configuration:** Ensure the connection is working properly and transactions are being processed correctly.
- **Monitor performance:** Track transaction times, gas usage, and network congestion.
- ✓ **Security**
- **Protect your private key:** Keep your private key secure to prevent unauthorized access.
- **Use secure communication channels:** Avoid exposing your private key or other sensitive information.
- **Stay updated:** Keep your node provider and software up-to-date with security patches.
- ✓ **Monitoring:**

- **Use monitoring tools:** Track network performance, transaction status, and potential issues.
- **Respond to alerts:** Address any problems promptly to avoid disruptions.
- ✓ **Documentation:**
- **Document your configuration settings:** Record network details, gas settings, and other relevant information.
- **Share with team members:** Ensure that others can understand and maintain your configuration.
- ✓ **Best Practices:**
- **Regularly review and update your configuration:** Stay informed about network changes and adjust your settings accordingly.
- **Consider using a cloud-based solution:** This can simplify management and provide scalability.
- **Seek professional help if needed:** If you encounter difficulties, consult experts or community forums for assistance.

Practical Activity 4.1.2: Connect to smart contract wallet using frontend

Task:

1: You are requested to go in computer lab to Connect smart contract wallet using frontend

2: Read the key readings 4.1.2

3: Apply safety precaution

4: Connect smart contract wallet using frontend.

5: Present your work to trainer.

6: perform the task provided in application of learning 4.1

Key readings 4.1.2: Connect to smart contract wallet using frontend

Web3 and Smart Contract Connection Guide

This guide covers the essential steps for connecting to smart contracts using web3 libraries, focusing on installation, connection, and frontend integration.

✓ **Installing Web3 Libraries**

- **ethers.js**

Ethers.js is a popular, complete and compact library for interacting with the Ethereum Blockchain.

npm install ethers

web3.js

- Web3.js is the original Ethereum JavaScript API.

npm install web3

✓ **Connecting to Smart Contracts**

To connect to a smart contract, you'll need two key pieces of information:

- Contract Address
- Application Binary Interface (ABI)

- **Using ethers.js**

```
const ethers = require('ethers');
// Connect to an Ethereum node
const provider = new
ethers.providers.JsonRpcProvider('https://mainnet.infura.io/v3/YOUR-PROJECT-
ID');
// Contract address and ABI
const contractAddress = '0x123...';
const contractABI = [...]; // Your contract ABI here
// Create a contract instance
const contract = new ethers.Contract(contractAddress, contractABI, provider);
// Now you can call contract methods
// For example:
// const result = await contract.someMethod();
```

- **Using web3.js**

```
const Web3 = require('web3');
// Connect to an Ethereum node
const web3 = new Web3('https://mainnet.infura.io/v3/YOUR-PROJECT-ID');
// Contract address and ABI
const contractAddress = '0x123...';
const contractABI = [...]; // Your contract ABI here
// Create a contract instance
const contract = new web3.eth.Contract(contractABI, contractAddress);
// Now you can call contract methods
// For example:
// contract.methods.someMethod().call().then(console.log);
```

✓ **Connecting to Smart Contract Wallet Using Frontend**

To connect a wallet in the frontend, you'll typically use a wallet provider like MetaMask. Here's a basic example using ethers.js and React:

```
import { ethers } from 'ethers';
import React, { useState, useEffect } from 'react';
function App() {
const [wallet, setWallet] = useState(null);
const [contract, setContract] = useState(null);useEffect(() => {
const connectWallet = async () => {
if (typeof window.ethereum !== 'undefined') {
try {// Request account accessawait window.ethereum.request({ method: 'eth_requestAccounts' });
const provider = new ethers.providers.Web3Provider(window.ethereum);
const signer = provider.getSigner();
setWallet(signer);// Connect to your contract
const contractAddress = '0x123...';
const contractABI = [...]; // Your contract ABI here
const contractInstance = new ethers.Contract(contractAddress, contractABI,
signer);
setContract(contractInstance);} catch (error) {console.error('Failed to connect wallet:', error);}} else {console.log('Please install MetaMask!');}};
connectWallet();}, []);
// Now you can use the 'contract' state to interact with your smart contract
// For example:
// const handleContractInteraction = async () => {
//   const result = await contract.someMethod();
//   console.log(result);
// };
return (<div>
{wallet ? (<p>Wallet connected!</p>) : (<p>Please connect your wallet</p>)}
{/* Add your contract interaction UI here */}
</div>);}
export default App;
```

Points to Remember

- Interacting with a deployed smart contract by calling functions and confirming transactions on the blockchain via MetaMask.
- Ensure you are connected to the appropriate blockchain network (Mainnet, Testnet, or custom/private network)
- Every wallet has a unique public address used to receive tokens
- Always double-check your wallet address before pasting it into the faucet to ensure the test ether is sent to the right wallet.

Application of learning 4.1.

A local organization wants to run a small-scale voting system to select a new committee member. They wish to use a decentralized application (DApp) on the Ethereum block chain for transparency and security. As block chain developer, you are requested to set up the voting system by configuring a contract network, installing browser extensions like MetaMask, creating a wallet, and loading test ether. The frontend of the DApp will allow users to connect their wallets and cast their votes.

Indicative content 4.2: Connecting smart contract

Duration: 5hrs

Practical Activity 4.2.1: Consume smart contract functions based on defined functionalities

Task:

1: You are requested to go in computer lab to Consume smart contract functions based on defined functionalities.
2: Read the key readings 4.2.1
3: Apply safety precaution
4: Consume smart contract functions based on defined functionalities
5: Present your work to trainer.
6: Perform the task provided in application of learning 4.2

Key readings 4.2.1: Consume smart contract functions based on defined functionalities

✓ **Smart contract Function Types :**

- **view:** Reads data from the blockchain without modifying the state.
- **pure:** Performs calculations without accessing the blockchain's state.
- **payable:** Accepts Ether as a parameter.
- **nonpayable:** Does not accept Ether as a parameter.

✓ **ABI (Application Binary Interface):**

- Defines the function signatures, parameters, and return types of a smart contract.
- Used by clients to interact with the contract.

✓ **Parameter Encoding:**

- The process of converting function parameters into a format that can be understood by the blockchain.
- Libraries like Web3.js or Ethers.js handle parameter encoding.

✓ **Error Handling:**

- Implementing mechanisms to catch and handle exceptions that may occur during function calls.
- Using try-catch blocks or custom error codes.

✓ **Transaction Fees:**

- The amount of gas consumed by a transaction, which determines the transaction fee.

- Optimize gas usage to reduce costs.

✓ **State Changes:**

- Functions that modify the blockchain state are called state-changing functions.
- view and pure functions do not modify the state.

✓ **Asynchronous Calls:**

- Executing functions without blocking the main thread, improving user experience.
- Using promises or callbacks in JavaScript.

✓ **Security:**

- Be aware of potential vulnerabilities like reentrancy attacks and input validation.
- Implement safeguards to prevent malicious attacks.

✓ **Testing:**

- Write unit tests to verify function behavior and identify potential issues.
- Use testing frameworks like Truffle or Hardhat.

✓ **Documentation:**

- Provide clear and concise documentation for developers using your functions.
- Explain parameters, return values, and usage examples.

Practical Activity 4.2.2: Create instance of smart contract

Task:

1: You are requested to go in computer lab to Create instance of smart contract.
2: Read the key readings 4.2.2
3: Apply safety precaution
4: Create instance of smart contract.
5: Present your work to trainer.
6: perform the task provided in application of learning 4.2

Key readings 4.2.2: Create instance of smart contract

✓ **Contract Address:**

- Obtain the address of the deployed smart contract from the blockchain explorer or transaction receipt.
- The contract address is a unique identifier that identifies the contract on the network.

✓ **ABI (Application Binary Interface):**

- The ABI defines the functions, parameters, and return types of the smart contract.
- It is essential for interacting with the contract's functionalities.
- You can obtain the ABI from the contract's source code or deployment transaction.

- ✓ **Web3 Library:**
- Use a Web3 library (e.g., Web3.js, Ethers.js) to interact with the Ethereum blockchain and your smart contract.
- These libraries provide a convenient way to connect to the network and call smart contract functions.
- ✓ **Contract Instance Creation:**
- Create an instance of the smart contract using the contract address and ABI.
- The instance represents the contract on your application's side.
- ✓ **Function Calls:**
- Call functions on the contract instance to interact with its functionalities.
- Pass the required parameters to the functions according to their definitions.
- Use asynchronous calls to avoid blocking the user interface.
- ✓ **Error Handling:**
- Implement error handling mechanisms to catch and handle potential exceptions.
- Use try-catch blocks or custom error codes.
- ✓ **Security:**
- Be mindful of security best practices to prevent vulnerabilities like reentrancy attacks and input validation.
- Validate input parameters and avoid sending sensitive data to untrusted contracts.
- ✓ **Testing:**
- Thoroughly test your contract interactions to ensure they function as expected.
- Use testing frameworks like Truffle or Hardhat to write unit tests.
- ✓ **Documentation:**
- Document your contract interactions for future reference and collaboration.
- Explain the purpose of each function and its parameters.

Example (using Web3.js):

```
const contractAddress = '0xYOUR_CONTRACT_ADDRESS';
const contractAbi = [/* ... your contract's ABI ... */];
const web3 = new Web3(window.ethereum);
const contract = new web3.eth.Contract(contractAbi, contractAddress);
async function callContractFunction() {const result = await
contract.methods.yourFunction().call();console.log(result);}
```

Points to Remember

- **Contract Address:** Obtain the address of the deployed smart contract. This address is unique and identifies the contract on the blockchain.
- **ABI (Application Binary Interface):** Obtain the ABI of the smart contract. The ABI defines the functions, parameters, and return types of the contract.
- **Web3 Library:** Use a Web3 library (e.g., Web3.js, Ethers.js) to interact with the Ethereum blockchain and your smart contract.
- **Contract Instance Creation:** Create an instance of the smart contract using the contract address and ABI.
- **Function Calls:** Call functions on the contract instance to interact with its functionalities.

Application of learning 4.2.

A university wants to conduct decentralized elections to select a new student council president. To ensure transparency and security, they choose to use a decentralized voting system (DApp) built on the Ethereum blockchain. This system will allow students to vote for candidates in a trustless manner.As the **blockchain developer**, you are tasked with setting up the voting DApp. Your responsibilities include creating smart contract instances for each election and ensuring that users can interact with the voting functions through the frontend. The system should allow students to vote, and the smart contract will automatically tally the votes.

Indicative content 4.3: Use of function

Duration: 15 hrs

Practical Activity 4.3:1 Implement operations based on smart contract predefined functions

Task:

1: You are requested to go in computer lab to Implement operations based on smart contract.
2: Read the key readings 4.3.1
3: Apply safety precaution
4: Implement operations based on smart contract.
5: Present your work to trainer.
6: perform the task provided in application of learning 4.3

Key readings 4.3.1: Implement operations based on smart contract

✓ **Function Understanding:**

- **Purpose:** Clearly understand the intended functionality of each function.
- **Parameters:** Identify the required input parameters and their data types.
- **Return Values:** Understand the expected output values and their data types.

✓ **ABI (Application Binary Interface):**

- The ABI defines the function signatures, parameters, and return types.
- Use tools like Solidity compiler or online ABI decoders to analyze the ABI.

✓ **Web3 Library:**

- Choose a suitable Web3 library (e.g., Web3.js, Ethers.js) based on your project's requirements.
- Set up the library and connect to the blockchain network.

✓ **Function Calls:**

- Use the appropriate methods provided by the Web3 library to call functions.
- Pass the required parameters in the correct format.
- Handle asynchronous calls using promises or callbacks.

✓ **Transaction Fees:**

- Estimate gas costs using gas estimation tools.
- Adjust gas limits and prices as needed to optimize transaction fees.

✓ **State Changes:**

- Be aware of how functions modify the blockchain state.

- Use **view** or **pure** functions for read-only operations.
- ✓ **Security:**
- Validate input parameters to prevent malicious attacks.
- Be mindful of reentrancy attacks and implement safeguards.
- Use secure coding practices to avoid vulnerabilities.
- ✓ **Testing:**
- Write unit tests to verify function behavior and identify potential issues.
- Test with different input values and scenarios.
- ✓ **Documentation:**
- Document the usage of each function, including parameters, return values, and expected behavior.
- Provide clear explanations and examples for developers.
- ✓ **Best Practices:**
- Use meaningful function names and comments to improve code readability.
- Handle errors and exceptions gracefully.
- Consider using a linter to identify potential issues in your code.

Practical Activity 4.3.2: Deploy web3 frontend based on specific requirements

Task:

1: You are requested to go in computer lab to Deploy web3 frontend based on specific.
2: Read the key readings 4.3.2
3: Apply safety precaution
4: Deploy web3 frontend based on specific 5: Present your work to trainer.
6: perform the task provided in application of learning 4.3

Key readings 4.3.2: Deploy web3 frontend based on specific requirements

- ✓ **Framework Selection:**
- Choose a suitable frontend framework (e.g., React, Vue, Angular) based on your team's expertise and project requirements.
- Consider factors like community support, component libraries, and performance.

- ✓ **Web3 Library Integration:**
- Integrate a Web3 library (e.g., Web3.js, Ethers.js) into your frontend application.

- Configure the library to connect to the desired blockchain network.

✓ **Smart Contract Interaction:**

- Create instances of your smart contracts using the Web3 library.
- Call functions on the smart contracts to interact with their functionalities.
- Handle asynchronous calls and potential errors.

✓ **User Interface (UI) Design:**

- Design an intuitive and user-friendly UI that meets your target audience's needs.
- Consider factors like layout, navigation, and visual appeal.

✓ **Data Management:**

- Implement mechanisms to store and manage user data, transaction history, and other relevant information.
- Use local storage, session storage, or databases as needed.

✓ **Security:**

- Protect your users' data and prevent security vulnerabilities like phishing attacks and cross-site scripting (XSS).
- Implement security best practices and use security libraries.

✓ **Testing:**

- Thoroughly test your frontend application to ensure it functions correctly and provides a good user experience.
- Use testing frameworks like Jest or Cypress.

✓ **Deployment:**

- Deploy your frontend application to a web server or hosting platform.
- Consider factors like performance, scalability, and security when choosing a hosting provider.

✓ **Maintenance:**

- Regularly update your frontend application to address bugs, improve performance, and add new features.
- Monitor user feedback and make necessary adjustments.

✓ **Best Practices:**

- Follow frontend development best practices for clean, maintainable, and efficient code.
- Use version control to track changes and collaborate effectively.
- Consider using a build tool like Webpack or Parcel to streamline the development and deployment process.

Points to Remember

- Make sure the DApp is connected to the correct Ethereum network (mainnet, testnet like Rinkeby or Goerli).
- Check if the smart contract is deployed on the same network as the frontend application.
- Ensure these values are correctly imported and utilized by the Web3 library (e.g., ethers.js or web3.js).
- Ensure the Web3 frontend correctly connects to MetaMask or other wallet extensions.
- Test wallet connection in the browser before deployment.
- Provide clear instructions for users to connect their wallets.
- Always test your DApp on a testnet (e.g., Rinkeby, Goerli) before deploying to the Ethereum mainnet.

Application of learning 4.3.

A small business startup wants to create a decentralized online marketplace using blockchain technology. The marketplace will allow sellers to list their products, and buyers will be able to purchase these products using a cryptocurrency (ETH). For transparency and security, the entire process will be managed by a smart contract on the Ethereum blockchain.

As a **blockchain developer**, your task is to implement the marketplace's functionality, which includes allowing sellers to add products, buyers to purchase products, and handling payments securely through the Ethereum network. You will also need to set up a frontend where users can interact with the marketplace (list products, browse, and make purchases) and the application can be deployed.

Learning outcome 4 End assessment

Q1.Circle the letter corresponding with the correct answer

i. **Which of the following is a common library used to integrate web3 applications with Ethereum smart contracts?**
   a. React.js
   b. Node.js
   c. ether.js
   d. MongoDB

ii. **What is the purpose of MetaMask in blockchain development?**
   a. To write smart contracts
   b. To interact with the Ethereum blockchain from the browser
   c. To compile smart contracts
   d. To create cryptographic hashes

iii. **What information is required to connect a smart contract to a web3 frontend?**
   a. Wallet balance and private key
   b. Contract address and ABI (Application Binary Interface)
   c. Wallet password and MetaMask address
   d. Public key and faucet details

iv. **What function does the smart contract typically provide in a decentralized crowdfunding platform?**
   a. Managing contributor identities
   b. Validating votes
   c. Handling contributions and withdrawals
   d. Deploying frontend files

Q2. Read carefully the following statements related to application of blockchain frontend integration and Answer by **True** if the statement is correct or by **False** is the statement is incorrect

i. The MetaMask browser extension is required to create a wallet and interact with the Ethereum blockchain.

ii. Once the smart contract is deployed, it is possible to update its code without redeploying it.

iii. Connecting to a smart contract using its ABI allows interaction with the functions defined within the contract.

iv. Test Ether from a faucet has real-world value and can be traded for goods or services.

Q3.Choose the right word to complete the sentence:

In order to load balance in the MetaMask wallet, you need to access a __________ and request test Ether.

A) contract
B) node
C) faucet
D) block

ii. **When creating a smart contract instance in a frontend, the _______________ and _____________ are used to establish a connection with the deployed contract.**
A) public key and gas fee
B) contract address and ABI
C) hash and nonce
D) private key and transaction ID

iii. **To consume smart contract functions, developers use libraries like ether.js, which communicates with the blockchain through _____________.**
A) HTTP
B) WebSocket
C) JSON-RPC
D) API

iv. **Once the web application is ready for production, developers must build the production bundle using tools like ____________ and configure the ______________ for deployment.**
A) Git and Docker
B) Webpack and environment variables
C) npm and API keys
D) Truffle and hardcoded secrets

Q4. Describe the process of configuring a contract network in Ethereum. What tools and steps are involved?

Q5.Explain how to connect a web3 frontend to a smart contract using keys (contract address, ABI). Provide details on how these components interact.

Q6. What are the steps involved in consuming smart contract functions through a web3 frontend? Give an example of a function call (e.g., contributing to a crowdfunding contract).

Practical assessment

A start-up company is developing a **decentralized crowdfunding platform** on the Ethereum blockchain to allow project creators to raise funds transparently. As the blockchain developer, you are tasked with setting up and deploying the platform. Your first step is to configure the Ethereum test network, ensuring the contract network matches the deployed smart contract. You'll install the MetaMask browser extension for development, create a new wallet, and load test Ether from a faucet to simulate real transactions.

Next, you'll integrate the Web3 frontend with the smart contract. This involves installing Web3 libraries like ether.js to enable communication between the frontend and the blockchain. Using the smart contract's address and ABI, you'll connect the contract to the frontend. The platform will provide functionalities such as contributing to a campaign, checking the fundraising status, issuing refunds, and withdrawing funds once the goal is met. You'll also create an instance of the smart contract to consume its functions and ensure users can contribute and withdraw through their connected wallets.

Finally, you will deploy the Web3 frontend based on the project's requirements, test the application to ensure everything works smoothly, build production-ready bundles, configure environment variables for security, and deploy the platform so users can interact with it in real-time.

END

Reference

OpenZeppelin. (n.d.). Security Considerations in Web3 Frontend Development.

Raggio, G. (2021). *Blockchain Development with Solidity and Ethereum.* Packt Publishing.

Škvorc, B. (2019). *Solidity Programming: A Developer's Guide.* Packt Publishing.

Team, E. (n.d.). *Ethers.js Documentation*. Retrieved from https://docs.ethers.org/

Team, R. (n.d.). *React Documentation*. Retrieved from https://reactjs.org/

Team, V. (n.d.). *Vue.js Documentation*. Retrieved from https://vuejs.org/

Team, W. (n.d.). Web3.js Documentation.

Team, W. (n.d.). *Web3.js Documentation*. Retrieved from https://web3js.readthedocs.io/en/latest/

Wood., A. M. (2019). *Ethereum Programming: The Definitive Guide.* O'Reilly Medi.

RTB | RWANDA TVET BOARD

October, 2024