IT Management & Audits

Practical Lab Manual

Smart Contract Development Kit

Practical P12

Learning Domain

Blockchain & Smart Contract Development

Course Learning Outcomes

CLO12: Develop, test, and deploy smart contracts on Ethereum

Unit

Unit VIII: Blockchain & FinTech Innovation

Time Allocation: 3 hours

Learning Mode: Hands-on (80%) + Theory (20%)

Difficulty Level: Intermediate-Advanced

Smart Contract Development Kit

Practical P12

Quick Reference

Practical Code	P12
Practical Name	Smart Contract Development Kit
Slot	T/P-12
Duration	3 hours
CLO Mapping	CLO12
Unit	Unit VIII: Blockchain & FinTech Innovation
Delivery Mode	Hands-on Lab
Target Audience	Intermediate-Advanced Level
India Integration	MEDIUM
Screenshot Count	5 Required

Prerequisites

- Basic JavaScript programming knowledge (variables, functions, async/await)
- Understanding of blockchain concepts (blocks, transactions, consensus)
- MetaMask browser extension installed and configured
- Node.js 18+ and npm installed on local machine
- Familiarity with command-line interface (terminal)

Tools Required

Tool	Version	Free Tier	Notes
Node.js	18+	✓	Runtime environment
npm	9+	✓	Bundled with Node.js
Hardhat	Latest	✓	Ethereum dev framework
Solidity	0.8.x	✓	Smart contract language
ethers.js	6.x	✓	Ethereum interaction library
MetaMask	Latest	✓	Browser wallet extension
Web Browser	Latest	✓	Chrome/Firefox recommended

Learning Objectives

- ✓ Set up a complete Ethereum smart contract development environment using Hardhat
- ✓ Write, compile, and understand Solidity smart contracts (storage, tokens, escrow)
- ✓ Execute comprehensive test suites for smart contract validation
- ✓ Deploy smart contracts to a local Hardhat blockchain network
- ✓ Interact with deployed contracts via scripts and a web frontend
- ✓ Understand ERC-20 token standards and smart contract security patterns
- ✓ Connect India-specific blockchain initiatives (Digital Rupee, SEBI tokenization) to practical concepts

What You Will Learn

By the end of this practical, you will:

1. Understand the Ethereum smart contract development lifecycle (write, compile, test, deploy)
2. Write and analyze Solidity smart contracts including state variables, functions, and events
3. Implement and understand ERC-20 fungible token contracts
4. Use Hardhat framework for compilation, testing, and deployment
5. Deploy contracts to a local blockchain and interact with them programmatically
6. Connect a web frontend to smart contracts using ethers.js and MetaMask
7. Recognize smart contract security vulnerabilities and mitigation strategies

Real-World Application

Smart contracts power decentralized finance (DeFi), tokenized assets, and digital identity systems worldwide. In India, the Reserve Bank of India (RBI) is piloting the **Digital Rupee (e-Rupee)** as a Central Bank Digital Currency (CBDC), while **SEBI** explores tokenization of securities. Platforms like **WazirX** and **CoinDCX** leverage smart contracts for token listings and escrow services. The **IFSCA blockchain sandbox** enables Indian startups to prototype blockchain-based financial products.

Hands-On Procedure

Part A: Environment Setup

Step 1: Clone the Smart Contract DevKit Repository

Objective: Set up the project structure with all smart contracts, tests, scripts, and frontend files.

1. Open your terminal and navigate to your preferred working directory
2. Clone the smart-contract-devkit repository
3. Explore the project structure

```
# Clone the repository
git clone https://github.com/smart-contract-devkit/starter.git
cd smart-contract-devkit

# Project structure:
# smart-contract-devkit/
# |-- contracts/
# |   |-- SimpleStorage.sol       # Basic storage contract
# |   |-- BasicToken.sol          # ERC-20 token contract
# |   |-- Escrow.sol              # Escrow payment contract
# |   |-- Voting.sol              # Decentralized voting
# |   |-- NFTMarketplace.sol      # NFT marketplace contract
# |-- test/                       # Test files for each contract
# |-- scripts/
# |   |-- deploy.js               # Deployment script
# |   |-- interact.js             # Contract interaction script
# |-- frontend/
# |   |-- index.html              # Web UI for contracts
# |   |-- app.js                  # Frontend logic
# |   |-- style.css
# |-- hardhat.config.js
# |-- package.json
# |-- .env.example
```

Clone Repository and Explore Structure

Expected Output

Repository cloned successfully. Directory listing shows:
`contracts/ test/ scripts/ frontend/ hardhat.config.js package.json .env.example`

Step 2: Install Dependencies and Configure Environment

Objective: Install all Node.js dependencies and configure the environment.

1. Run `npm install` to install all project dependencies
2. Verify Hardhat installation by checking its version
3. Copy the environment example file and configure placeholder values

```
# Install all dependencies
npm install

# Verify Hardhat installation
npx hardhat --version
# Expected output: 2.x.x (e.g., 2.19.4)

# Copy environment example file
cp .env.example .env
# Edit .env with placeholder values:
# PRIVATE_KEY=your_wallet_private_key_here
# ALCHEMY_API_KEY=your_alchemy_api_key_here
# Note: .env is NOT required for local development.
```

Install Dependencies and Configure

Expected Output

```
added 350+ packages in 45s
npx hardhat -version
2.19.4
```

Never commit your `.env` file to version control. It contains sensitive keys. For this lab, we primarily use the local Hardhat network, which does not require real private keys.

Part B: Compile & Test Smart Contracts

Step 3: Compile All Smart Contracts

Objective: Compile all Solidity contracts and review the generated artifacts.

1. Run the Hardhat compile command
2. Review the compilation output for errors or warnings
3. Examine the generated `artifacts/` directory (ABI and bytecode)
4. Open `contracts/SimpleStorage.sol` — a basic contract with `store(uint256)` and `retrieve()` functions plus a `ValueChanged` event

```
# Compile all Solidity contracts
npx hardhat compile
```

Compile Contracts

Expected Output

```
Compiling 5 Solidity files with 0.8.x
Compilation finished successfully
```

Artifacts generated in `artifacts/contracts/` for each contract (SimpleStorage, BasicToken, Escrow, Voting, NFTMarketplace).

Each artifact JSON file contains the contract ABI (Application Binary Interface) and bytecode. The ABI defines how external applications interact with the contract—listing all public functions, parameters, and return types.

Step 4: Run the Complete Test Suite

Objective: Execute all test files to validate smart contract functionality.

1. Run the Hardhat test command to execute all test files
2. Review pass/fail status for each contract
3. Note gas consumption reported for each operation

```
# Run all tests
npx hardhat test

# Tests cover: SimpleStorage (store/retrieve, events),
# BasicToken ERC-20 (mint, transfer, approve, transferFrom),
# Escrow (deposit, release, refund), Voting (proposals,
# delegation, vote counting), NFTMarketplace (mint, list, buy)

```

```
# Run tests with gas reporting
npx hardhat test --gas-reporter
```

Run Test Suite

Expected Output

```
SimpleStorage
✓ Should store and retrieve a value (42ms)
✓ Should emit ValueChanged event (28ms)
BasicToken
✓ Should mint initial supply to deployer (35ms)
✓ Should transfer tokens between accounts (52ms)
✓ Should approve and transferFrom (61ms)
✓ Should fail with insufficient balance (22ms)
Escrow
✓ Should accept deposits / release / refund
Voting
✓ Should create proposals / delegate / count votes
NFTMarketplace
✓ Should mint / list / execute purchase
15 passing (8s)
```

Screenshot 1

What to paste: Project directory structure (showing contracts/, test/, scripts/, frontend/ folders) alongside the Hardhat compilation output showing all 5 contracts compiled successfully.

Paste your screenshot here

Screenshot 2

What to paste: Terminal output showing the complete test suite results for all contracts (SimpleStorage, BasicToken, Escrow, Voting, NFTMarketplace) with all tests passing.

Paste your screenshot here

Step 5: Deep Dive into BasicToken.sol (ERC-20)

Objective: Understand the ERC-20 token standard interface and security patterns.

1. Open `contracts/BasicToken.sol` in your code editor
2. Identify each ERC-20 interface function and its purpose
3. Review security patterns: reentrancy guards, access control (Ownable)
4. Understand the `approve` + `transferFrom` delegated transfer pattern

```
# ERC-20 Standard Interface:
# totalSupply()      -> total tokens in existence
# balanceOf(addr)    -> token balance of address
# transfer(to, amt) -> transfer from caller to recipient
# approve(spender, amt) -> allow spender to withdraw
# transferFrom(from, to, amt) -> delegated transfer
# allowance(owner, spender) -> remaining allowance

# Security Patterns:
# ReentrancyGuard - prevents recursive calls
# Ownable - restricts mint() to contract owner
# Built-in overflow checks (Solidity 0.8.x)
# Event emission: Transfer, Approval for transparency
```

ERC-20 Interface Functions in BasicToken.sol

The ERC-20 `approve/transferFrom` pattern is critical for DeFi protocols. When you approve a DEX contract to spend your tokens, it uses `transferFrom` to execute swaps. Always verify the spender address and amount before approving.

Part C: Local Deployment

Step 6: Deploy to Hardhat Local Network

Objective: Start a local Ethereum node and deploy all contracts.

1. **Terminal 1:** Start the Hardhat local blockchain node
2. Observe the 20 test accounts generated with 10000 ETH each
3. Note the local RPC URL: `http://127.0.0.1:8545`
4. **Terminal 2:** Run the deployment script against the local network
5. Record the deployed contract addresses from the output

```
# TERMINAL 1: Start local Hardhat blockchain node
npx hardhat node
# Shows 20 accounts with private keys and
# starts JSON-RPC server at http://127.0.0.1:8545/

# TERMINAL 2: Deploy all contracts to localhost
npx hardhat run scripts/deploy.js --network localhost
# Deploys: SimpleStorage, BasicToken (1M tokens),
# Escrow, Voting (with proposals), NFTMarketplace
```

Deploy to Local Network

Expected Output

```
Deploying contracts with account:  0xf39F...2266
SimpleStorage deployed to:  0x5FbD...1234
BasicToken deployed to:  0xe7f1...5678
Escrow deployed to:  0x9fE4...9abc
Voting deployed to:  0xCf7E...def0
NFTMarketplace deployed to:  0xa513...4567
All contracts deployed successfully!
```

Keep Terminal 1 running throughout the lab session. The local blockchain node must be active for all subsequent steps. If you close it, all deployed contracts and state will be lost.

Screenshot 3

What to paste: Terminal output showing the local Hardhat node running (with test accounts) and the deployment script output showing all contract addresses.

Paste your screenshot here

Step 7: Interact with Deployed Contracts

Objective: Call functions on deployed contracts to store/retrieve values, mint and transfer tokens.

1. Run the interaction script against the local network
2. Observe function calls: store a value, retrieve it, mint tokens, transfer tokens
3. Check Terminal 1 for transaction logs confirming each operation

```
# Run the interaction script
npx hardhat run scripts/interact.js --network localhost

# The script performs:
# SimpleStorage: store(42), retrieve() -> 42
# BasicToken: totalSupply(), balanceOf(), transfer(1000)
# Escrow: deposit(1 ETH), getBalance()
# Voting: getProposals(), vote(0)
```

Interact with Contracts

Expected Output

```
=== SimpleStorage ===
Storing value:  42 | Retrieved value:  42
=== BasicToken ===
Total supply:  1000000.0 BTK
Transferring 1000 BTK to Account1...  Transfer successful!
Account1 balance:  1000.0 BTK | Deployer:  999000.0 BTK
=== Escrow ===
Depositing 1.0 ETH...  Escrow balance:  1.0 ETH
=== Voting ===
Proposals:  ["Proposal A", "Proposal B", "Proposal C"]
Voted for:  Proposal A | Votes:  1
```

Screenshot 4

What to paste: Terminal output showing the contract interaction script results—function calls and returned values for SimpleStorage, BasicToken, Escrow, and Voting.

Paste your screenshot here

Part D: Frontend Integration & Testnet Deployment

Step 8: Connect Frontend to Smart Contracts via MetaMask

Objective: Interact with deployed contracts through a browser-based UI connected to MetaMask.

1. Ensure the Hardhat node is still running (Terminal 1)
2. Configure MetaMask to connect to Hardhat localhost: Network Name: `Hardhat Local`, RPC URL: `http://127.0.0.1:8545`, Chain ID: `31337`, Symbol: `ETH`
3. Import a Hardhat test account into MetaMask using its private key
4. Open `frontend/index.html` in your browser (or serve via `npx serve frontend/`)
5. Click "Connect Wallet" to link MetaMask
6. Use the UI to store a value, view token balances, and transfer tokens

Expected Output

Browser displays the Smart Contract DevKit frontend:
- Connected address: `0xf39F...2266`
- ETH Balance: 9998.xx ETH (less due to gas from deployments)
- BTK Token Balance: 999000.0 BTK
- SimpleStorage current value: 42
- Token transfer form functional

If MetaMask shows "Internal JSON-RPC error" or transactions fail after restarting the Hardhat node, reset the account: Settings > Advanced > Clear Activity Tab Data. This resets the nonce counter to match the fresh blockchain state.

Step 9: (Optional) Deploy to Sepolia Testnet

Objective: Deploy a contract to the Ethereum Sepolia test network and verify on Etherscan.

1. Configure MetaMask for Sepolia testnet (built-in network)
2. Get free testnet ETH from `https://sepoliafaucet.com` or `https://faucets.chain.link`
3. Update `.env` with your wallet private key and Alchemy API key
4. Deploy and verify the contract

```
# Deploy SimpleStorage to Sepolia
npx hardhat run scripts/deploy.js --network sepolia
```

```

# Verify contract on Etherscan
npx hardhat verify --network sepolia <DEPLOYED_ADDRESS>
# View: https://sepolia.etherscan.io/address/<ADDRESS>
```

Deploy to Sepolia Testnet

Testnet deployments cost no real money but simulate the full Ethereum deployment experience. Sepolia is the recommended testnet as Goerli has been deprecated.

Screenshot 5

What to paste: Browser showing the frontend application connected to MetaMask wallet, displaying the connected address, ETH balance, token balance, and contract interaction interface.

Paste your screenshot here

Conceptual Background

Blockchain Fundamentals

A blockchain is a distributed, immutable ledger that records transactions across a network of computers.

- **Decentralization:** No single authority controls the network; consensus is achieved among distributed nodes
- **Immutability:** Once recorded, data cannot be altered without consensus of the network majority
- **Consensus Mechanisms:** Protocols (Proof of Work, Proof of Stake) ensuring all nodes agree on ledger state
- **Blocks & Transactions:** Batches of signed messages cryptographically linked in a chain

Ethereum and the EVM

- **EVM (Ethereum Virtual Machine):** Turing-complete runtime executing smart contract bytecode on every node
- **Gas:** Computational fee for executing operations; prevents infinite loops and spam
- **Accounts:** Externally Owned Accounts (EOA, controlled by private keys) and Contract Accounts (controlled by code)
- **Smart Contracts:** Self-executing programs that automatically enforce rules and agreements
- **State:** World state stores account balances, contract storage, and code for every address

Solidity Language Basics

- **Data Types:** `uint256`, `int256`, `address`, `bool`, `string`, `bytes`, `mapping`, arrays
- **Functions:** Visibility (`public`, `private`, `internal`, `external`) and mutability (`view`, `pure`, `payable`)
- **Modifiers:** Reusable conditions (e.g., `onlyOwner`, `nonReentrant`)
- **Events:** Logged data for external listeners (e.g., `Transfer`, `Approval`)
- **Inheritance:** Contracts inherit using `is` keyword (e.g., `contract BasicToken is ERC20, Ownable`)

Token Standards

Standard	Type	Use Cases
ERC-20	Fungible Tokens	Cryptocurrencies, utility tokens, stablecoins
ERC-721	Non-Fungible Tokens	Digital art, collectibles, identity, real estate
ERC-1155	Multi-Token	Gaming items, batch transfers, mixed token types

Smart Contract Security

1. **Reentrancy:** External contract calls back before state updates. Mitigation: ReentrancyGuard, checks-effects-interactions pattern.
2. **Overflow/Underflow:** Arithmetic exceeds type limits. Mitigation: built-in in Solidity 0.8.x; SafeMath for older versions.
3. **Access Control:** Unauthorized privileged function calls. Mitigation: Ownable, role-based AccessControl, multi-sig.
4. **Front-Running:** Bots observe and front-run pending transactions. Mitigation: commit-reveal schemes, time-locks.
5. **Denial of Service:** Unbounded loops or reliance on external calls. Mitigation: pull-over-push patterns.

Development Workflow

Write (Solidity contracts) → **Compile** (bytecode + ABI) → **Test** (Chai/Mocha on Hardhat) → **Deploy** (local/testnet/mainnet) → **Verify** (Etherscan) → **Monitor** (events, balances)

Hardhat Framework

- **Tasks:** Extensible CLI commands for compile, test, deploy, and custom workflows
- **Networks:** Built-in local blockchain plus configurable testnet/mainnet connections
- **Console:** Interactive JavaScript console for live contract interaction
- **Testing:** Mocha test runner with Chai assertions and ethers.js integration
- **Plugins:** Gas reporter, contract verification, coverage analysis

India Context: Blockchain in Indian Financial Ecosystem

RBI Central Bank Digital Currency (CBDC) – Digital Rupee

- **e-Rupee (Retail):** Pilot launched December 2022 with SBI, ICICI, Yes Bank, IDFC First
- **e-Rupee (Wholesale):** For interbank settlements and government securities
- **Technology:** Blockchain-based distributed ledger managed by RBI
- **Goal:** Reduce dependence on physical currency, enable programmable money, improve financial inclusion

SEBI Tokenization and Regulatory Sandbox

- Tokenization of unlisted commercial paper and bonds
- Blockchain-based KYC processes and securities settlement (T+0)
- IFSCA blockchain sandbox for testing blockchain-based financial products

Real-World Indian Blockchain Applications

- **WazirX/CoinDCX:** Smart contracts for token listing, escrow, and P2P trading
- **Land Registry:** Andhra Pradesh and Telangana piloting blockchain for tamper-proof land records
- **Supply Chain:** Coffee Board of India using blockchain for farm-to-cup traceability

Assessment & Deliverables

Assessment Questions

Q1. Explain the difference between a blockchain and a traditional centralized database. What properties make blockchain suitable for financial applications?

Q2. Describe the six functions defined in the ERC-20 token standard. Why is the `approve/transferFrom` pattern necessary for DeFi?

Q3. What is a reentrancy attack? Describe how ReentrancyGuard prevents it, and give a real-world example (e.g., The DAO hack).

Q4. Write a Solidity function that accepts ETH payments and emits an event. Explain `payable` and `msg.value`.

Q5. Why is testing critical for smart contracts compared to traditional software? What happens if a bug is found after mainnet deployment?

Q6. Explain the purpose of gas in Ethereum. How does gas pricing affect smart contract design?

Q7. Describe the RBI Digital Rupee (e-Rupee) initiative. How does it differ from decentralized cryptocurrencies like Ether?

Q8. What is the IFSCA blockchain sandbox? How can Indian FinTech startups benefit from it?

Deliverables Checklist

Item	Description	Type	Status
Screenshot 1	Project structure + compilation output	Paste	☐
Screenshot 2	Test suite results (all contracts)	Paste	☐
Screenshot 3	Local deployment with contract addresses	Paste	☐
Screenshot 4	Contract interaction via scripts	Paste	☐
Screenshot 5	Frontend + MetaMask interaction	Paste	☐
Addresses	All deployed contract addresses	Text	☐
Test Report	Pass/fail count and gas usage	Text	☐
Questions	All 8 assessment questions answered	Text	☐

Verification Checklist

- ☐ Repository cloned and project structure verified
- ☐ All npm dependencies installed; Hardhat version confirmed (2.x.x)
- ☐ All 5 contracts compiled without errors
- ☐ All 15 tests passing in the test suite
- ☐ Hardhat local node running on port 8545
- ☐ All contracts deployed with addresses recorded
- ☐ Contract interaction script executed with correct outputs
- ☐ MetaMask connected to Hardhat localhost; frontend loaded
- ☐ Token transfer completed via frontend
- ☐ All 5 screenshots captured and pasted; all 8 questions answered

Grading Rubric

Criteria	Description	Points	Score
Environment Setup	Repo cloned, deps installed, Hardhat verified	10	___/10
Compilation	All contracts compiled without errors	10	___/10
Testing	All tests passing, gas report reviewed	15	___/15
ERC-20 Understanding	Deep dive analysis of BasicToken.sol	10	___/10
Deployment	Contracts deployed to local network	15	___/15
Interaction	Script-based contract calls successful	10	___/10
Frontend	MetaMask connected, UI interaction working	10	___/10
Screenshots	All 5 screenshots submitted clearly	10	___/10
Assessment	All 8 questions answered thoroughly	10	___/10
	TOTAL	**100**	___/100

Appendix A: Solidity Quick Reference

Data Types	Description	Visibility	Modifiers
`uint256`	Unsigned integer	`public` – internal + external	`view` – reads state only
`int256`	Signed integer	`private` – same contract	`pure` – no state access
`address`	20-byte Ethereum address	`internal` – contract + derived	`payable` – receives ETH
`bool`	Boolean (true/false)	`external` – outside only	`onlyOwner` – owner access
`string`	Dynamic UTF-8 string		`nonReentrant` – reentrancy guard
`mapping`	Key-value hash table		
`bytes32`	Fixed-size byte array		

Appendix B: ERC-20 Interface Reference

```
interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool
        );
    function allowance(address owner, address spender) external view
        returns (uint256);
    function approve(address spender, uint256 amount) external returns (
        bool);
    function transferFrom(address from, address to, uint256 amount)
        external returns (bool);
    event Transfer(address indexed from, address indexed to, uint256
        value);
    event Approval(address indexed owner, address indexed spender,
        uint256 value);
}
```

IERC20 Standard Interface

Appendix C: Hardhat Commands Reference

```
npx hardhat compile                # Compile all contracts
npx hardhat test                   # Run all tests
npx hardhat test --gas-reporter    # Tests with gas reporting
npx hardhat node                   # Start local blockchain
npx hardhat run scripts/deploy.js --network localhost   # Deploy
npx hardhat console --network localhost   # Interactive console
npx hardhat clean                  # Clean artifacts
npx hardhat verify --network <net> <addr>   # Verify on Etherscan
```

```
npx hardhat coverage              # Test coverage report
npx hardhat help                  # Show all available tasks
```

Essential Hardhat CLI Commands

Appendix D: MetaMask Setup Guide

1. Install MetaMask from `https://metamask.io/download` (Chrome or Firefox)
2. Create a new wallet, set a strong password, back up Secret Recovery Phrase offline
3. Add Hardhat Local network: Name: `Hardhat Local`, RPC: `http://127.0.0.1:8545`, Chain ID: `31337`, Symbol: `ETH`
4. Import test account: Account icon > Import Account > paste private key from Hardhat node output
5. Imported account should show 10000 ETH on the Hardhat Local network

Appendix E: Common Solidity Errors

Error	Cause	Solution
`ParserError: Expected ';'`	Missing semicolon	Add `;` at end of line
`TypeError: undeclared identifier`	Variable not declared	Declare variable or fix typo
`revert: Ownable: caller is not the owner`	Non-owner calling restricted function	Call from owner account
`revert: ERC20: insufficient allowance`	transferFrom without approval	Call approve() first
`revert: transfer amount exceeds balance`	Insufficient token balance	Check balance before transfer
`out of gas`	Gas limit too low	Increase gas limit or optimize code

Appendix F: Troubleshooting Guide

Solutions:

1. Verify Node.js version: `node -version` (must be 18+)
2. Clear npm cache: `npm cache clean -force`
3. Delete `node_modules` and `package-lock.json`, then retry
4. Try: `npm install -legacy-peer-deps`

Solutions:

1. Check Solidity version in `hardhat.config.js` matches pragma in contracts
2. Ensure OpenZeppelin is installed: `npm install @openzeppelin/contracts`
3. Run `npx hardhat clean` then compile again
4. Verify import paths are correct (case-sensitive on Linux/Mac)

Solutions:

1. Verify Hardhat node is running in Terminal 1
2. Confirm Chain ID is `31337` in MetaMask network settings
3. Reset account: Settings > Advanced > Clear Activity Tab Data
4. Try `http://localhost:8545` instead of `http://127.0.0.1:8545`
5. Re-import test account private key if balance shows 0

Appendix G: Resources

Official Documentation

→ Solidity: `https://docs.soliditylang.org`

→ Hardhat: `https://hardhat.org/docs`

→ OpenZeppelin: `https://docs.openzeppelin.com/contracts`

→ ethers.js: `https://docs.ethers.org`

→ Ethereum: `https://ethereum.org/en/developers`

→ ERC-20 (EIP-20): `https://eips.ethereum.org/EIPS/eip-20`

→ RBI Digital Rupee: `https://www.rbi.org.in`

→ IFSCA Sandbox: `https://ifsca.gov.in`

Tools Used in This Practical

Tool	Purpose	Cost
Node.js	JavaScript runtime for development tools	Free
Hardhat	Smart contract development framework	Free (open source)
Solidity	Smart contract programming language	Free
ethers.js	Ethereum interaction library	Free (open source)
MetaMask	Browser-based Ethereum wallet	Free
OpenZeppelin	Audited smart contract library	Free (open source)
Etherscan	Blockchain explorer and verification	Free

—END OF LAB MANUAL—

Document Version: 1.0

IT Management & Audits – Practical Lab Series