**CODELANDCS BLOCKCHAIN DEVELOPMENT SYLLABUS**

**WEEK 7**

**DAY 1**

**INTRODUCTION TO MOCKS IN SMART CONTRACTS**

Smart contracts are self-executing contracts with the terms of the agreement directly written into code. They are usually deployed on decentralized platforms such as Ethereum and are tamper-proof, transparent, and verifiable. In the development of smart contracts, developers often use testing frameworks to ensure that their contracts function as expected. **One of the tools used in testing smart contracts is Mocks.** This lecture will discuss what mocks are, why they are important in smart contract testing, and how they can be used to improve the quality of smart contract code.

**WHAT ARE MOCKS?**

In software development, a **mock is a simulated version of a real object** that is used to test the functionality of other objects. Mocks are used in testing to create a controlled environment where the **behavior of the real object can be simulated.** In smart contract testing, mocks are used to **simulate the behavior of external contracts** and to test the interaction between different parts of a smart contract.

Mocks are **created using a testing library or framework,** and they can be **programmed to return specific values or trigger specific events** when certain conditions are met. For example, a mock for an external contract that returns a token balance can be programmed to return a specific balance when called. This allows the developer to test the functionality of their contract **without actually interacting with the external contract.**

**Why are Mocks Important in Smart Contract Testing?**

The use of mocks in smart contract testing has several benefits, including:

**Simulating External Contracts:** External contracts are an important part of smart contract functionality, but interacting with them can be difficult and costly. By using mocks to simulate the behavior of external contracts, developers can test their contracts **without incurring the cost of interacting with the real contrac**t.

**Testing Edge Cases:** Mocks can be used to test edge cases and unusual scenarios that may be difficult to replicate in the real world. For example, a mock can be used **to simulate a network failure** or a scenario where an external contract **returns an unexpected value**.

**Improving Test Coverage:** By using mocks, developers can test the interaction between different parts of a smart contract without relying on external contracts. This can improve test coverage and reduce the likelihood of bugs.

**Improving Test Efficiency:** By simulating the behavior of external contracts, developers can run tests more efficiently and reduce the time and resources required for testing.

**How to Use Mocks in Smart Contract Testing?**

To use mocks in smart contract testing, developers need to follow these step**s:**

**Identify External Contracts:** Identify the external contracts that the smart contract interacts with and determine the functions that need to be mocked.

**Write Mocks:** Write mocks for the external contracts using a testing framework such as Truffle or Hardhat. The mocks should simulate the behavior of the external contracts and return expected values when called.

**Test the Contract:** Use the mocks to test the smart contract by calling its functions and verifying the results. The tests should cover all edge cases and scenarios that the contract may encounter.

**Refactor the Contract:** If any issues are discovered during testing, refactor the smart contract code and repeat the testing process.

**Conclusion**

In conclusion, mocks are an essential tool in smart contract testing that can help developers improve the quality of their code, reduce the cost of testing, and increase test coverage. By simulating the behavior of external contracts, developers can test edge cases and unusual scenarios, improve test efficiency, and reduce the time and resources required for testing. To use mocks effectively, developers need to identify the external contracts that their smart contract interacts with, write mocks that simulate the behavior of those contracts, test the contract thoroughly, and refactor the code as needed.

**HARDHAT DEPLOY PACKAGE**

Normally, **after compilation is to deploy** and previously we used the scripts folder which contains the deploy file to deploy the contract. The deploy file here is not really keeping track of our deployments in any file. Here we would use the **hardhat-deploy package** (replicable deployment and easy testing). Run ***yarn add --dev hardhat-deploy*** and afterwards require/import into the configuration file (delete the previous deploy script). Make a deploy-folder which would be responsible for holding our deployment files and folders (ideal to identify them correctly or numbered). The folder would be where the **hardhat module looks to deploy code.** Since we are still using ethers, ***run yarn add --dev @nomiclabs/hardhat-ethers@npm:hardhat-deploy-ethers ethers***.

Here hardhat-deploy-ethers package would override hardhat- ethers package to add extra features to our code. It would enable ethers to keep track of all deployments. When we run yarn hardhat deploy, it would call an anonymous function(s) inside all files in the deploy folder. These **functions automatically would take in the hardhat runtime environment** as a parameter (or we can just destructure the things we need which are getNamedAccounts, deployments, getChainId, getUnnamedAccounts). The Hardhat Runtime Environment, or HRE for short, is an object containing all the functionality that Hardhat exposes when running a task, test or script.

**What is getNamedAccounts method?**

A way to get named account from the configuration file. We can get our accounts based off the number in the account section of each network. Instead add a section at the bottom, **namedAccounts** so we can name each spot in the accounts array.

We should bear two thitwo things :- identify an account with its default number and then position accross different chains.

The deployments object can be used to get two functions which are deploy and log. Next grab the chainId from the network object.