Environment Setup and Project Initialization

Checking Environment Readiness

- Ensure your **environment is properly set up** to avoid security risks and other problems.
- Today's lesson focuses on running tests on `hello.sol`.
- Proper setup is crucial for the next six lessons.
- The goal is to replicate actions without issues.

Review of VS Code Setup

- A short review on setting up the coding environment with VS Code will be provided in the last 10 minutes of the lesson.

Addressing Node Version Issues

- NVM (Node Version Manager) is recommended for managing Node versions.
  - NVM allows quick swapping of Node versions.
  - Example: `$ NVM use 20` to switch to Node version 20.
- If encountering errors related to Node version compatibility with Hardhat:
  - The error might be a warning and Hardhat can still be used.
- The lecture uses Node version 22, same as a student in the class.

Creating a New Project with Hardhat

1. Create a new folder (e.g., `project2`).
2. Enter the new project folder.
3. Initialize npm: `$ npm init`
4. Install Hardhat as a dev dependency: `$ npm install --save-dev hardhat`
5. Run Hardhat initialization: `$ npx hardhat`

Installing Hardhat Locally

- Hardhat should be installed locally for each project to allow different projects to use different Hardhat versions.

Testing with Hello.sol

Skipping OpenZeppelin and Task Parts

- Skipping the OpenZeppelin and task parts from the previous lesson.
- Cleaning up the project to start fresh for the new lesson.
- Instructions on how to start a new project will be given at the beginning of every lesson.

Selecting an Option

- Create an **empty** hardhat.config.js
- **Do not** choose create a basic sample project
- **Do not** choose create an advanced sample project

Why Ethers over Web3js

- Ethers is used over Web3js because the decentralized application built later in the boot camp with React will use Ethers.

Setting Up a New Project with Hardhat

These are the steps for setting up a new project with **Hardhat**:

1. Install Hardhat.
2. Initialize the project.
    - It should not give any errors, but it might give warnings.
3. Create `mara.rc` file.
    - Remember to put the dot and everything, using your preferred text editor.
4. Test if everything is working by running `npx hardhat compile`.
    - This was slow in yesterday's lesson.
5. Run `npx hardhat test`.

If you want to have another project for today's lesson, you can create a new project and go along. Alternatively, you can use yesterday's project folder.

Common Issues and Solutions

Conflicting Peer Dependency Errors

Start a new project, creating a new folder, initialize npm there, and then install Hardhat.

Error H203

> This error indicates that you are trying to initiate a project inside an existing Hardhat project, referring to a path with `hardhat.config.ts`.

To fix this, pick a directory in another place (e.g., the desktop) and ensure that your folder is not inside any other project folder on your device.

Package.json Hierarchy Issue

Ensure that your project folder is not nested within a directory containing a `package.json` file. The project folder should not be inside another project.

Testing the Setup

To test if everything is correct, run the command:

```
npx hardhat compile
```

If it compiles one file or says nothing to compile, you're good to go.

Using a Text Editor

`nano` is a text editor used for quick edits, but you can use any text editor you prefer.

Hardhat Configuration

The `tsconfig.json` file should be in the root directory, together with the `package.json` and `hardhat.config.ts` files.

Hardhat Runtime Environment (HRE)

When we build a smart contract inside the Hardhat project and place it in the correct folder, Hardhat can use it. Smart contracts are written in Solidity and are compiled to target the Ethereum Virtual Machine (EVM). Hardhat uses the **Hardhat Runtime Environment (HRE)** to emulate all the workings of blockchains:

- Blocks
- Transactions
- Accounts

All of this happens in a very lightweight client-side EVM client, which allows you to:

- Run scripts
- Run tests
- Connect to testnets
- Connect to mainnet

All inside one environment. It's a very complete tooling that can serve a lot of different purposes, such as:

- Building smart contracts
- Security
- Helping you build decentralized applications on top of it

Hardhat uses TypeScript as the major language, which is convenient for front-end development.

Understanding the Lock Smart Contract

This section breaks down the functionality of a smart contract called "Lock" and the associated testing process.

Lock Contract Overview

The Lock smart contract includes elements like events, a payable constructor, and requirements. Reading the Solidity code can be complex, but the provided TypeScript test file clarifies its behavior.

Test Scenarios

The test file includes several scenarios:

- Deployment should set the correct unlock time.
- Deployment should set the correct owner.
- Should receive and store funds to lock.
- Should fail if the unlock time is not in the future.
- Withdrawal should fail if called too soon or by another account.
- Withdrawal should succeed under the correct conditions.

Functionality Explained

The Lock smart contract works like this:

1. It receives funds.
2. It locks the funds until a specified time.
3. After the unlock time, the owner can withdraw the funds.

For example, if you lock $1,000 in the contract until Christmas, neither you nor anyone else can withdraw it before then, ensuring the money is saved for a specific purpose.

Benefits of Testing

Testing provides peace of mind during development by clearly stating what the smart contract does in a human-readable way. This helps to avoid paranoia and ensures the code functions as expected.

Project Initialization and Setup Review

The following steps review how to initialize a project:

1. `npm init`
2. `npm install`
3. `npx hardhat`
4. Use Nano to create and edit files.

File Configuration

- moarc.json: Create this file and include the necessary content. This configuration is essential for using the test suite with the Mara test explorer, enabling features like running individual tests through a graphical interface. Without this file, the testing interface will not function correctly.

```
{
  // Configuration content here
}
```

- .env: Fill this file with your actual API keys. It is needed for Thursday's lesson.

> Important: Do not use private keys from wallets containing real funds on the mainnet. Instead, use a new, empty development wallet for testing.

Testing Environment Setup

1. Install the recommended extension: Mocha Test Explorer for the Test Book.
2. Enable the extension and trust the project.
3. Refresh tests to see them in the testing pane.

Project Cleanup

To start with a clean project:

1. Remove all files from the `contracts` folder.
2. Remove all files from the `ignitions` folder.
3. Remove all files from the `tests` folder.
4. Run `npx hardhat clean` to clear artifacts and cache.

Using Nano

Nano can be used to create and edit files directly in the terminal.

> Alternatively, you can use any text editor you prefer, such as Notepad or VS Code.

Clearing Initial Project Files

- When starting a new project, it's common to clear out initial files in directories like contracts, ignitions, and test.
- This helps keep the project organized.
- This can be done manually by deleting the files directly or copying the correct files back and forth.
- Copying the correct files back and forth creates a boiler plate or cookie cutter project.

Lesson 6: Understanding the Testing Environment

Objectives

- Understand the environment and what happens under the hood during tests.
- Understand how Solidity interacts with the EVM (Ethereum Virtual Machine) through TypeScript.
- Understand the VM library used for testing.
- Use the VM library to interact with the blockchain in a composable and type-safe manner.
- Create unit tests targeting execution with VM inside the Hardhat Runtime Environment (HRE).
- Use the same syntax for tests as for scripts.
- Learn about Solidity data structures like mappings, arrays, and structs.
- Debug and use the environment as a sandbox for testing.

The Hardhat Sandbox

- Hardhat provides a well-prepared sandbox for automated testing, unlike manual processes with Remix.
- Hardhat is your lab for you to test things freely.

Mocha Testing Suite

- Mocha is a widely-used testing suite for Node.js, not exclusive to smart contracts.
- Despite seeming complex, testing is presented as a simplified process, akin to an enhanced console log.

Test Driven Development (TDD) Methodology

- A testing methodology where unit tests are written to fail first, then code is written to make them pass.
- Helpful because smart contract development is more similar to building hardware because smart contracts are difficult to roll back, fix, or add features to after deployment.

TDD for Smart Contracts

- For every hour spent developing smart contracts, spend approximately five hours testing.
- Agile development methodologies can be used, but avoid releasing with known issues for later fixes.
- Writing unit tests first helps with collaboration, progress measurement, and problem identification.

Creating a Hello Contract

1. Create a new file named `hello.sol` in the `contracts` folder.
2. Copy the content from the previous week's project into the file.

Dealing with Initial Project Files

- When you initialize a new Hardhat project, it includes some pre-existing files.
- These files can be removed to start with a clean slate.

  Metaphor: This process is compared to buying shoes with paper inside; you can either remove the paper or keep it depending on your needs.

Solidity Extension in VS Code/Cursor

- The speaker recommends using the Solidity extension from Hardhat for correct styling.
- You can search for solidity in the extensions tab.

Testing Smart Contracts

Testing with TypeScript

- Instead of manually deploying smart contracts to a test net and interacting with them, Hardhat provides a full suite for testing, especially when integrating with apps.
- Create a TypeScript file (e.g., `helloWorld.ts` or `test.ts`) inside the `test` folder. The file name does not matter.
- Initially, the test file is just a script written in TypeScript.

```
console.log("hello");
```

- When executed using `npx hardhat test`, the script runs, and the output is displayed.
- The test suite (Mocha) provides an output indicating the number of passing and failing tests.

Rudimentary Testing

- Tests can be written with basic conditional checks.

```
const a = Math.random();
if (a > 0.5) {
  console.error("My test failed");
}
```

- If the condition fails, an error is thrown. However, this approach is messy and lacks organized output.

Assertion Libraries

- Assertion libraries like Chai are used to organize test outputs and provide structured assertions.
- Assertions define requirements that must be met.
  - Example: `expect(answer).to.equal(42);`

Parallel Testing Environments

- Parallel testing environments ensure that tests do not interfere with each other.
- Each test runs in a separate environment with new instances of objects, preventing one test from influencing others.
- Metaphor: Testing different aspects of a car (speed, air conditioning, comfort, crash resistance). Testing crash resistance first would influence the subsequent comfort and functionality tests if the same car were used.

Test Structure

- Use the `describe` scenario to structure tests, as suggested by Mocha.
- Each `describe` block represents a specific scenario.
- Tests within each block are isolated to prevent interference.

```
describe("Array", function () {
  describe("#indexOf()", function () {
    it("should return -1 when the value is not present", function ()
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

Example Test

- Here's an example illustrating the use of `describe` and `it` blocks to test array functionality:

```
describe("Array", function() {
  describe("#indexOf()", function() {
    it("should return -1 when the value is not present", function() {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

    - This structure helps in organizing tests and ensuring each test operates independently.

Testing Smart Contracts

Importance of Testing

- In Solidity, once a smart contract is deployed, it's difficult to iterate or fix errors.
- Any errors in a deployed smart contract can lead to significant financial losses.
- Deploying smart contracts is analogous to shipping hardware; recalling faulty smart contracts can be very costly.
- Testing is crucial to identify and rectify issues before public release.
- Testing on testnets before deploying to the mainnet is advisable.

Benefits of Using Tests

- Aids in learning Solidity.
- Useful for future smart contract development.

Test Structure Explained

Test Scenario Example

- Recall that after deploying a simple "Hello World" smart contract, it should return "Hello World".
- A test scenario (e.g., line 27 in the provided script) tests this behavior.
- Use a test suite or test explorer to run tests.
- The test suite will:
    - Bootstrap the virtual machine blockchain.
    - Deploy the smart contract.
    - Call the relevant function.
    - Check if the expected value ("Hello World") is returned.

Demonstration of a Failing Test

1. Modify the smart contract code (e.g., change "Hello World" to "Hello there").
2. Compile the contract.
3. Run the test again.
    - The test should now fail, indicating that the expected value ("Hello World") does not match the actual value ("Hello there").

Developer Experience

- The goal is to create a workflow where changes in Solidity code can be quickly tested using TypeScript.
- TypeScript is used for building front-end applications and scripts.
- The ability to navigate and communicate between Solidity and TypeScript is crucial.

Common Questions

Why Hardhat Over Foundry?

- Hardhat is preferred because of its TypeScript integration.
- Skills learned in TypeScript can be reused for building applications later.

Contract Owner is of Type Unknown Error

- This error often occurs when the smart contracts haven't been compiled.
- Compile the smart contracts to resolve this issue.

No Tests Have Been Found Error

- Ensure the `.mocharc.json` file exists in the project.
- Verify that VS Code is opened in the correct project folder.
- Refresh the test explorer to detect the tests.
- The `.mocharc.json` file instructs the test explorer where to find the test files.

Compiling After Changes

- Whenever changes are made to the Solidity code, the contract must be recompiled.
- Compiling from the command-line interface (CLI) typically compiles automatically.
- When using the Market Test Explorer, you need to manually compile.

Testing Smart Contracts

Bridging Solidity with Testing Suites

The lecture addresses how to bridge Solidity contracts with testing suites, focusing on the use of the Hardhat Runtime Environment (HRE).

- Solidity is designed for interacting with the Ethereum Virtual Machine (EVM), not for writing tests.
- Tools like Foundry allow testing with Solidity.

The HRE facilitates communication with the EVM, enabling actions like deploying contracts and managing wallet clients.

Troubleshooting Mocha Test Explorer Issues

- Some students may encounter issues with the **Mocha Test Explorer** in VS Code or similar environments like Cursor.
- Problems can often be resolved by ensuring the project is open in the correct folder, that the `package.json` file is present, and that the `mocha.rc.json` file is properly configured.
- If issues persist, refreshing the test discovery may help.
- Testing can also be performed via the command line using `npx hardhat test`, which compiles the contracts before running tests.

Compiling Smart Contracts in VS Code

Smart contracts can be compiled directly within VS Code or Cursor.

- Open the command palette using `Ctrl+Shift+P` (or `Cmd+Shift+P` on macOS).
- Type "Compile" and select the "Hardhat: Compile Project" option.
- This requires the Hardhat extension to be installed.

Understanding the Hardhat Runtime Environment (HRE)

The **Hardhat Runtime Environment (HRE)** is crucial for composing a local blockchain environment.

> The **Hardhat Runtime Environment (HRE)** is a tool to link and bridge a working blockchain environment to base scripts and tests.

Key functionalities of the HRE include:

- Connecting to a local or forked blockchain.
- Accessing pre-configured accounts.
- Deploying smart contracts.
- Interacting with public blockchain data.

The HRE is not related to the Holy Roman Empire.

Load Fixture Function

The **load fixture** function is used to reset the state of the blockchain for each test, preventing interference between tests. This ensures each test runs in a clean environment, simulating going back in time for each test.

Debugging with Breakpoints

Breakpoints are used to debug tests and inspect variables.

- Set breakpoints by clicking in the margin next to the line numbers in your code.
- Use the debugger to step through the code, inspect variable values, and modify them on the fly.
- Debugging requires the Hardhat Test Explorer, as it is not possible to do it directly from the command line interface.
- Click "Debug Test" to start the debugging session.

Public Client

The **public client** is an object within the HRE that allows you to fetch information from the public JSON-RPC API of a blockchain node.

With the Public Client you can:

- Get chain information (ID, URLs).
- Estimate gas.
- Get account balances.
- Get blocks and last blocks.
- Get transactions.## Debugging with Breakpoints
- Breakpoints can be placed in the code to pause execution at specific lines.
- To initiate debugging:
    - Set breakpoints in the desired lines.
    - Right-click on a test scenario and select "Debug Test".
    - Alternatively, use the debug tab to select a task and click "Debug Test".
- This allows you to step through the code and inspect variables.

Examining Variables During Debugging

- While debugging, you can inspect the state of variables at different points in the code.
- For example, you can check the contents of the `alner` object, including account addresses.
- The **pre-funded accounts**, like the ones with addresses starting with `0xf3` and `0x7099`, are the same accounts used when running the `task accounts`.
- You can see the **deployed contract's address and ABI**, which includes the constructor and view functions.
- You can interact with the environment due to pre-coded functionality.

Interacting with Smart Contracts During Tests

- During tests, you can interact with smart contracts as if they were local objects.
- You can read values from the contract using commands like `read`.
- This allows you to easily retrieve information from the smart contract deployed in the Hardhat runtime environment.

Testing Environment and Setup

- The testing environment creates a testnet from scratch, starting at block zero.
- It can also be configured to fork a testnet and pick balances.
- To see tests in the Test Explorer, ensure:
    - The test file is in the same folder as the configuration.
    - The `.mocharc.json` file is configured.
    - Tests are located inside the `test` folder.
    - Use the refresh tests button in the test explorer to see your tests

Running Tests and Analyzing Results

- All tests can be run at once to quickly validate scenarios.
- Tests can check various aspects, such as:
  - Correct deployment of a smart contract
  - Setting the owner to the deployer account
  - Preventing unauthorized access to functions like `transferOwnership`
- You can observe transaction hashes when writing comments or transferring ownership.
- After a transaction is included in the blockchain, you can verify the changes.

Example Test Scenarios

- Example tests to consider:
  - Testing that the smart contract returns "Hello World" upon deployment.
  - Ensuring that only the owner can call `transferOwnership`.
  - Validating that a call to `transferOwnership` from a non-owner account is rejected.
  - Testing the `transferOwnership` function to ensure it works correctly.

Applying Test Scenarios to Homework

- Create a test scenario that prevents anyone other than the owner from changing the text.
- Create a test scenario to ensure the text changes correctly

Understanding Test Scripts

- The test scripts can guide you to understand operations performed manually.
- The same operations done manually, such as transferring ownership or setting text, can be automated with tests.

Homework and Further Learning

- Focus on reading the referenced pages rather than exhaustively clicking every link.
- Reading through the hardhat runtime environment page is recommended.
- Use GitHub issues to ask questions and report any issues encountered.
- The tools learned will be used to explore a new smart contract related to **Ballot**, including **mappings, arrays, and structs**.