Development Tooling Setup

Addressing Student Questions

- A student asked about issues with the **transfer ownership** function in their **Hello World contract**.
- The issue might stem from attaching the interface to the wrong smart contract address.
- Even with the correct function, attaching it to an older, incompatible smart contract deployment will not work.
- The solution is to ensure the interface is connected to the correct deployed instance of the **Hello World contract** that has the **transfer ownership** function.

Introduction to Development Tooling

- Today's lecture focuses on development tooling rather than new blockchain concepts.
- The goal is to transition from Remix to local development environments using tools like Hardhat.
- Emphasis will be placed on writing high-quality, maintainable, and auditable code.
- The lecture aims to familiarize students with JavaScript and TypeScript for front-end and back-end development.
- Students will also learn to write scripts for setting up environments and deploying smart contracts securely.
- Writing tests is crucial for ensuring code correctness, especially after encountering issues with modifier changes.

Programming Setup

IDE (Integrated Development Environment)

- A local IDE (Integrated Development Environment) is recommended.
- VS Code is suggested, but the professor uses VS Codium, a fork of VS Code.

    > VS Code is a source code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control and GitHub, syntax highlighting, intelligent code completion, snippets, and code refactoring. It is highly customizable, allowing users to change the theme, keyboard shortcuts, preferences, and install extensions that add additional functionality.

- While VS Codium provides the same functionality, using it is not mandatory.
- However, the professor will be using specific VS Code extensions, which may not be directly compatible with other IDEs like Eclipse or JetBrains.
- If using another IDE, finding equivalent extensions may be necessary, or they might not be available.

CLI Tools

- CLI (Command Line Interface) tools such as Yarn, npm, and Node will be used for running tasks locally.
- These tools should have been installed as part of Lesson Four's homework.

OpenZeppelin Contracts

- The lecture will start with the **OpenZeppelin Contracts** repository.
- Although not used directly today, **OpenZeppelin Contracts** will be cloned, installed, and used as a reference.

    > OpenZeppelin Contracts is a library for smart contract development, providing implementations of standards like ERC20 and ERC721, as well as reusable components to build secure smart contracts.

- **OpenZeppelin Contracts** is widely trusted within the Web3 community for providing safe, reusable, and modular smart contract definitions.
- Using smart contracts from **OpenZeppelin** inherits a high level of security and community-vetted code.
- The aim is to emulate the tools and practices of **OpenZeppelin** to achieve similarly secure results.

Prerequisites

- **Node.js** must be installed correctly.
- To check the installed version, open a terminal or console and type `node -v`.
- Ideally, the version should be version 20.

Setting Up Your Environment

Version Considerations

When setting up your environment, consider the following:

- Node.js Versions:
    - Versions around 20 or 22 should work.
    - Versions before 16 or 18, or after 24, might cause issues.
- Operating Systems:
    - The lesson is tested on Windows and Linux.
    - Mac should work similarly to Linux, but watch out for permission issues.
    - If you're using Homebrew, ensure everything is correctly installed to avoid headaches.

Node Version Manager (NVM)

If you don't have Node.js installed, or if you need to manage multiple Node.js versions, use Node Version Manager (NVM).

- What it does:
    - Lists installed Node.js versions.
    - Downloads new versions.
    - Upgrades existing versions.
- Installation:
    - Windows: Use NVM for Windows.
    - Other OS: Install NVM, then install Node.js.

If you install Node.js with NVM, npm (node package manager) should be installed automatically.

Cloning the Repository

Clone the OpenZeppelin Contracts repository to see how the tools are used.

```
git clone <repository_url>
```

Folder Structure and Organization

Ensure you place your projects in an organized folder structure.

- **Example**: `home/<user>/projects/code/`
- **Avoid**: Creating a project inside another project, as it can lead to problems.
- **Windows**: When opening a command prompt, ensure you are not inside the `system32` folder.
    - Use the `cd` command to navigate to your desired folder.
    - Alternatively, open Explorer/File Finder and open a terminal from there.

Git CLI Installation

You need **Git CLI** to download the repository.

- If you don't have it, install it.
- Alternatively, download the repository as a ZIP file from the green button on the repository page and extract it.

Installing Dependencies with npm

Navigate into the downloaded directory using the `cd` command.

```
cd <directory_name>
```

- **Note**: In Windows, you might encounter forward slashes instead of backslashes.
    - Use tab to autocomplete the directory name and avoid typos.

After navigating into the directory, use **npm** to install the project dependencies.

```
npm install
```

- npm (Node Package Manager):

    A package manager that installs all project dependencies defined in the `package.json` file.

- The `package.json` file contains:
    - Project name.
    - Description.
    - Version.
    - List of dependencies.

- Dependencies: Instead of installing each dependency individually, npm automates the process.

Post-Installation Notes

- Vulnerabilities: You might see some vulnerabilities reported, but these are usually in the repository and not in the smart contracts themselves. So it is not a problem.
- If `git clone` fails, download the ZIP file directly from the page and extract it.

The install command creates a `node_modules` folder containing all the necessary dependencies.

Compiling the Contracts

To check if everything is set up correctly, run the following command:

```
npm run compile
```

If everything is installed correctly, this command will download the compiler and process the script.

Package.json Scripts

The `package.json` file includes pre-programmed **scripts**. One such script is the **compile script**, which is executed using **npm** (Node Package Manager).

Node Version Warnings

- A warning about hardhat not being supported may appear.
- It's recommended to **update Node** when possible.

Accessing Resources

All necessary links for Node, Yarn, Git, VS Code, and OpenZeppelin contracts are available in the **readme** file for Lesson 5.

Compilation Output

A successful compilation should produce an output like "Compile 231," indicating successful smart contract compilation.

Folder Structure

Ensure you're in the correct directory: **openzeppelin-contracts**. This is crucial for running commands like `npm install` and `npm run compile`.

Dependency Installation

If you encounter a "solidity dog not installed" error, run `npm install` within the correct folder.

Node Version Management

If Node version 22 isn't working, switch back to version 20. Delete the **node_modules** folder and reinstall. NVM (**Node Version Manager**) allows quick switching between Node versions.

Common Issues and Solutions

Issue	Solution
Missing compile command	Ensure you are in the `openzeppelin-contracts` directory.
Incorrect folder	Navigate to the correct folder using `cd openzeppelin-contracts`. Use tab to autocomplete the folder name.
Hardhat compile instruction not working	Verify you are in the correct folder and run `npm run compile`.
No node_modules folder	Run `npm install` to install the dependencies.
Nothing to compile	This means you have already compiled. Compiled artifacts are in the **artifacts** folder. Delete this folder to recompile.
Errors due to forward slashes	When changing directories, type `cd` , then a space, and begin typing the folder name. Use the tab key to autocomplete the folder name. This helps avoid errors caused by incorrect forward slashes. This is especially helpful on Windows.

Artifacts Folder

The **artifacts** folder contains compiled contract artifacts. It is created after running the compile command. You can delete it to force recompilation.

Testing Smart Contracts with JavaScript

Running Test Scripts

- A JavaScript script can be used to test smart contracts.
- The test script is configured in the `package.json` file.
- The Hardhat tooling is used to compile contracts and run the test file.
- The test file, written by OpenZeppelin maintainers, tests the necessary behaviors of the smart contract.
- To run the test script:

```
npm test
```

Updating Node Version

- Sometimes, updating the Node version can resolve issues with running test scripts.

```
nvm install node_version
```

    - Where `node_version` represents the version to install.

- After updating Node, it may be necessary to reinstall the node modules:

```
npm install
```

Package.json Scripts

- The `package.json` file contains scripts for compiling and testing.
- The `compile` script compiles the smart contracts.
- The `test` script runs the JavaScript test file.

Example Test Output

- A successful test run should show all tests passing, with output similar to:

```
126 passing
```

- The test script checks various aspects of the contract, such as balance and name.

Making a Breaking Change

- A **breaking change** is a modification to the contract that causes existing tests to fail.
- To demonstrate this, the `decimals` function in the ERC20 contract can be modified.

Decimals Function

- The `decimals` function defines the number of decimals the token uses.
- It is typically set to 18.
- The function is defined as:

```
function decimals() public view virtual returns (uint8) {
    return 18;
}
```

- Changing the return value of the `decimals` function from `18` to `42` will cause tests to fail.

Observing Test Failures

- After modifying the `decimals` function and running the tests, errors should appear in the terminal.
- These errors indicate that the smart contract's output differs from what the test files expect.
- For example, the ERC20 and ERC20Mock approval tests may fail.

Ethers.js and the Link Between Solidity and JavaScript

- Ethers.js is a package that creates a link between the JavaScript environment and the smart contract.
- It allows describing functions and statements in JavaScript that interact with the smart contract.
- For example, a test can assert that the token decimals equal 18:

```
expect(token.decimals()).to.equal(18);
```

- This creates a connection between the JavaScript test and the Solidity contract.

Tools for Smart Contract Development

This section focuses on the tools used for smart contract development, emphasizing Hardhat for creating a JavaScript environment linked to smart contracts.

Linking JavaScript Environments and Smart Contracts

- The goal is to create a strong link between JavaScript environments and smart contracts.
- Hardhat will be used extensively for the next few weeks to understand smart contracts, tests, and decentralized applications (DApps).
- The link between JavaScript environments and smart contracts will be developed over the next few lessons.

Importance of Quality Code

- Focus on creating code with **composability**, **upgradeability**, **maintainability**, and **readability**.
- Tools, including tests and Test-Driven Development (**TDD**), will ensure a well-organized development flow.

    > TDD: Test Driven Development. A development process relying on tests to drive the development of the program, improving code quality and maintainability.

Alternative Tools

- Other tools, such as Foundry and Forge, can accomplish similar tasks.
- Hardhat is chosen for its close integration with JavaScript and TypeScript.

Transition to Hardhat

- The class will transition from using OpenZeppelin to Hardhat for new projects.
- The initial steps for future lessons will be based on Hardhat.

Hardhat Environment

- Hardhat provides an environment for running **Ethereum** and any **EVM-compatible chain**.

    > EVM: Ethereum Virtual Machine

- It includes components for editing, compiling, and debugging smart contracts.

VS Code Extensions

- Two VS Code extensions will be introduced and used in future lessons. Students should install them.

TypeScript Configuration

- Using TypeScript can help avoid problems associated with JavaScript's flexibility.
- TypeScript's **static type checks** are beneficial for newcomers.

Step-by-Step Project Creation

Pay close attention to the following steps, as missing a command can lead to failure.

1. **Exit the Current Directory**:
    - Exit the current working folder (e.g., OpenZeppelin project).
    - Choose a different folder on your computer.

```
cd ..
```

2. **Create a New Project Folder**:
    - Create a new folder for the project using `mkdir` or any preferred method.

```
mkdir hardhat_project
```

3. **Navigate into the New Folder**:
    - Enter the newly created project folder.

```
cd hardhat_project
```

4. **Initialize a Repository**:
    - Initialize a new repository using `npm init`.

```
npm init
```

    - Accept the default options by repeatedly pressing Enter or use `npm init -y` for the default "yes" option.
        - This command creates a `package.json` file with default settings.
5. **Install Hardhat Locally**:
    - Install Hardhat as a development dependency using `npm install --save-dev hardhat`.

```
npm install --save-dev hardhat
```

    - This command adds Hardhat to the `package.json` file and installs it in the `node_modules` directory.

6. Execute Hardhat Init Command:
    - Use `npx` to execute the Hardhat initialization command.

```
npx hardhat init
```

Setting Up Your Development Environment

Choosing the Right Project Type

When creating a new project, select the **"create a typescript project with VM"** option. This setup uses the VM package, which is designed to closely mirror front-end coding practices, instead of ethers, for smart contract interactions. This approach is intended to align with the coding practices used in front-end development.

Initializing Your Project

Use the following command to initialize your project, selecting the default options during setup:

```
npx hardhat
```

Dependency Installation Issues

- In some cases, Hardhat may face issues during dependency installation.
- If auto-installation fails, the output will include a list of dependencies that need to be manually installed.

Windows-Specific Configuration

- If you directly install Node.js on Windows, you must configure the system environment path to ensure command scripts work correctly.
- Alternatively, using Node Version Manager (NVM) for Windows handles path configuration automatically.

Recommended VS Code Extensions

Essential Extensions

The professor recommends the following VS Code extensions to help with testing, scripting, and smart contract interaction:

- Marqeta Test Explorer: A visual tool to run and debug test scripts.
- Solidity by Nomic Foundation (Hardhat): Offers utilities and features specifically for Hardhat projects, enhancing code readability and functionality.

Installing the Solidity Extension

When searching for the Solidity extension, two options are available:

- Solidity by Juan Blanco
- Solidity by Nomic Foundation for Hardhat The professor suggests installing the Solidity by Nomic Foundation for Hardhat extension for its Hardhat-specific features and helpful commands.

Configuring Test Environment

Creating `mocharc.json`

To properly utilize the Marqeta Test Explorer, create a file named `.mocharc.json` in the root directory of your project. This configuration file is not generated by default during Hardhat initialization.

Contents of `mocharc.json`

The `.mocharc.json` file should contain the following content:

```
{
  "require": "ts-node/register",
  "timeout": 10000,
  "spec": "test/**/*.ts"
}
```

Purpose of `mocharc.json`

This configuration file is necessary to set up the testing environment, specifying settings like timeout and test file paths.

Managing Environment Variables

Creating `.env` File

Create a `.env` file in the root directory of your project to store sensitive information such as private keys, mnemonics, and API keys.

Storing Credentials

Store your mnemonic, private key, and RPC provider API keys (such as Infura or Alchemy) in the `.env` file. For example:

```
PRIVATE_KEY=your_private_key
ALCHEMY_API_KEY=your_alchemy_api_key
```

Security Warning

Never use mainnet private keys or mnemonics. Always create a new wallet for testing purposes. Sharing your private key can lead to theft of your cryptocurrency.

Including `.env` in `.gitignore`

Ensure that the `.env` file is included in your `.gitignore` file to prevent sensitive information from being committed to your repository.

Testing Configuration

To verify that your setup is working, run the following commands:

```
npx hardhat compile
npx hardhat test
```

Hardhat Tooling Configuration

This section covers the configuration and usage of the Hardhat development environment for Ethereum smart contract development.

Setting Up a New Project

1. Ensure Hardhat is correctly configured on your computer.
2. Verify that Hardhat behaves as expected.
3. Create a new address for each new account; **do not share your private key or mnemonic.**

   > Sharing private keys or mnemonics can lead to security issues, especially with mainnet balances.

4. List all files in the directory using the command `ls -la`.
5. Create a new project using Hardhat. This project is configured and can compile and test smart contracts.

Hardhat Configuration File

- The `hardhat.config.ts` file is the configuration file for Hardhat.
- It organizes Hardhat settings, such as the Solidity version.

Testing the Setup

1. After setting up the new project, run the following commands to test the setup:

```
npx hardhat compile
npx hardhat test
```

   These commands should compile the contracts and run the tests.

Creating and Running Tasks

1. Create a new test file, e.g., `accounts.ts`.

2. Add a task definition inside the `hardhat.config.ts` file. This task uses the Hardhat Runtime Environment (HRE) to get wallet clients.

```
// Example task definition in hardhat.config.ts
task("accounts", "Prints the list of accounts", async (taskArgs, hre)
  const accounts = await hre.ethers.getSigners();

  for (const account of accounts) {
    console.log(account.address);
  }
});
```

3. Update the import statement or use the quick fix option in VS Code (Ctrl + . or Cmd + .) to resolve any import errors.

4. Run the task using the command:

```
npx hardhat accounts
```

   This command invokes the `accounts` task and prints the configured addresses.

Local Development Network

- Hardhat uses a **local development network** by default, which is easy to set up and destroy after running tests and scripts.
- This local network contains a list of pre-funded addresses that can be used for testing.
- Similar to Remix, Hardhat provides pre-funded accounts for quick tests.

Testing with Mocha and Solidity

- Use the **Mocha test explorer** to run testing scripts directly from the VS Code interface.
- Set breakpoints in the test files to debug and inspect variable values during test execution.

TypeScript Syntax for Hardhat

- A basic understanding of **TypeScript syntax** is needed for working with Hardhat.
- Suggested reading material: "Learn TypeScript in 5 Minutes."

Relevance of Remix IDE Tutorials

- For the current week, focus on Hardhat documentation instead of Remix IDE tutorials.

Key Concepts

Concept	Description
Hardhat	A development environment to compile, test, debug and deploy Ethereum software.
Private Key	A secret key that allows you to control your Ethereum account.
Mnemonic	A set of words that can be used to recover your private key.
Local Dev Network	A private Ethereum network running on your computer, used for testing and development.
Pre-funded wallets	Accounts on the local dev network that come with pre-loaded ETH for easy testing.
Task	A custom command that can be run using Hardhat.
TypeScript	A superset of JavaScript that adds static typing, often used for writing smart contract tests and scripts.
Mocha	A JavaScript test framework used for writing and running tests.