Week 3 Overview: Tokenization and Standards

This week focuses on solidifying your understanding of token standards and their practical applications. We'll be working with ERC20 and ERC721 tokens, utilizing testing and scripting tools like Hardhat and TypeScript. The goal is to prepare you for next week's full-stack development, which will involve backend calls.

Environment Setup and Syntax

Ensure your environment is properly configured, and you're comfortable with the syntax and tools (Hardhat, EVM, libraries). This week offers a chance to resolve any lingering issues before the more intensive development next week.

Tokens in the Ecosystem

Common Uses of Tokens

Tokens are smart contracts representing ownership of assets, both fungible and non-fungible.

> Fungible Assets: Assets that are interchangeable and have the same value.
>
> Non-Fungible Assets: Unique assets that cannot be interchanged, each having a distinct value.

Utility of Network Tokens vs. Other Tokens

- Network Tokens are necessary for acquiring block space and processing transactions.
- Other Tokens are used for various purposes:
  - Stablecoins (e.g., USDC)
  - Meme Coins
  - Utility Tokens
  - NFTs

Token Standards

Token standards define how these tokens are created and managed.

Ethereum Improvement Proposals (EIPs)

EIPs Defined

> **Ethereum Improvement Proposals (EIPs)**: Community-driven discussions and proposals that guide the development and evolution of the Ethereum ecosystem.

EIPs cover specifications, standards, and potential upgrades. These proposals start as discussions and evolve based on community feedback.

Governance of Ethereum

Ethereum is managed by its community, primarily through:

- **Node Operators**: They decide which protocol upgrades to support by upgrading or not.
- **Developers**: They follow standards in smart contract and wallet design.

This contrasts with centralized applications where feature changes are controlled by a central authority (e.g., Zoom).

Application-Layer Standards

These standards, like token standards, define ways to represent asset ownership.

Navigating the Token Standard Landscape

There are numerous standards beyond **ERC20** and **ERC721**. The goal is to learn how to read and apply these standards effectively. Even if a token doesn't strictly follow an ERC standard, the code will still compile. It just won't be compatible with applications that assume that the interface.

Learning to Apply Standards

The goal is to create a "sandbox environment" to learn and apply new standards independently. This approach will be useful for understanding future concepts like flash swaps, flash minting, and oracles.

Practical Application: Creating a New Project

To reinforce learning, it's helpful to create a new project and apply the standards. You can quickly duplicate an existing project structure, excluding the `node_modules`, `artifacts`, and `cache` folders, and potentially the `package-lock.json` file.

To create a new project quickly:

1. Copy everything from the old project except `node_modules`, `artifacts`, and `cache`.
2. Paste into your new project folder.
3. Install dependencies.## □ ERC-20 Token Standard

When writing a smart contract that should be compatible with other platforms, such as decentralized exchanges, lending protocols, or marketplaces, it's important to adhere to the **ERC-20 token standard**.

Functions in ERC-20

The ERC-20 standard requires the implementation of specific functions:

- **Optional Functions**:
  - `name`
  - `symbol`
  - `decimals`
- **Mandatory Functions**:
  - `totalSupply`: Returns the total token supply in circulation.
  - `balanceOf`: Accepts an address and returns the token balance for that address.

Challenges of Coding ERC-20 From Scratch

Coding an ERC-20 token from scratch involves:

- Reading the ERC-20 standard definition.
- Implementing a transfer mechanism with proper authorization.
- Complying with the required structure.

This process can be challenging, error-prone, and less maintainable. It is generally not recommended to code ERC-20 tokens from scratch.

OpenZeppelin Contracts Library

Instead of coding from scratch, the **OpenZeppelin Contracts library** offers a collection of reusable smart contracts for secure smart contract development.

> OpenZeppelin Contracts library is a library where you can import pieces of smart contracts for secure smart contract development.

Benefits of Using OpenZeppelin

- **Community-vetted code**: The library contains community-vetted code used for billions of dollars worth of transactions.
- **Reusability**: Solidity components can be reused like building blocks.
- **Standards implementation**: Includes implementations for ERC-20, ERC-721, and more.

How to Use OpenZeppelin

To use OpenZeppelin:

1. Import the desired contract, such as `ERC20` or `ERC721`, from the OpenZeppelin contracts library.

2. Inherit the contract in your own smart contract.

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract GLDToken is ERC20 {
    // Custom logic here
}
```

Inheritance

Inheritance allows a contract to copy the code of another contract and extend or override its functionality.

> Inheritance is a way that we can make a contract copy the code of another contract and append it or override it.

Key Aspects of Inheritance

- A contract can inherit state variables and functions from another contract.
- Overriding is possible for functions marked as `virtual`.
- Accessibility:
  - `public`, `external`, and `internal` members can be accessed.
  - `private` members cannot be accessed.
- Additional functions can be added on top of the inherited contract.
- Inheritance is about copying code at the developer level, not affecting deployed contracts.

Inheritance Graph

Understanding the inheritance graph is crucial, especially when dealing with multiple levels of inheritance.

For example, if contracts `D` and `E` inherit from both `B` and `C`, the order of inheritance matters. If `D` and `E` inherit from `B` first and then from `C`, they will pick up functions from `C`, potentially overwriting those inherited from `B`. Contract `F`, which only inherits from `B`, will pick up functions directly from `B`.

Installing OpenZeppelin

Instead of cloning the OpenZeppelin repository, it's recommended to install it as an npm package.

```
npm install @openzeppelin/contracts
```

This approach allows you to lock a specific version of OpenZeppelin and avoid using the main branch directly.

Creating Tokens with OpenZeppelin

When creating tokens, such as an ERC20 token (e.g., "MyToken") or an ERC721 token (e.g., "MyNFT"), OpenZeppelin contracts can be used. To do so, you can create a contract that imports from the relevant OpenZeppelin contract. For example:

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symb
        _mint(msg.sender, 1000 * 10 ** decimals());
    }
}
```

This contract imports the ERC20 contract from OpenZeppelin, defines a constructor that takes a name and a symbol, and mints tokens to the message sender.

OpenZeppelin Contracts Explained

OpenZeppelin provides pre-built, tested, and audited smart contracts for various use cases, including token standards like ERC20 and ERC721.

Key Components in ERC20 Contracts:

- Total Supply: The total number of tokens in existence.
- Name: The name of the token (e.g., "MyToken").
- Symbol: The symbol of the token (e.g., "MTK").
- Allowances: A mapping that allows token holders to approve other addresses to spend their tokens.
- Balances: A mapping of account addresses to their token balances.

Using OpenZeppelin

1. **Installation**: Install OpenZeppelin contracts using a package manager like npm or yarn.

```
npm install @openzeppelin/contracts
```

2. **Import**: Import the necessary contracts into your Solidity file. For example:

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

3. **Inheritance**: Inherit from the OpenZeppelin contract in your contract.

```
contract MyToken is ERC20 {
    // ...
}
```

Explicit Imports

When importing contracts, explicitly name what you are importing to improve maintainability and readability. For example:

```
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

This makes it clear what you are importing from the file, especially when a file contains multiple contract definitions.

Locating OpenZeppelin Contracts

After installing OpenZeppelin contracts, you can find the source code in the `node_modules` directory under the `@openzeppelin` folder. This directory contains all the contracts, including **ERC20**, **ERC721**, **ERC1155**, and their extensions.

Testing Smart Contracts

When developing smart contracts, it is important to test their functionality. You can write unit tests in TypeScript to test functionalities such as:

- Minting tokens
- Transferring tokens
- Checking balances

Extending OpenZeppelin Contracts

OpenZeppelin contracts are designed to be composable, allowing you to extend and customize their functionality. You can inherit multiple contracts and override functions as needed.

Role-Based Access Control

Role-Based Access Control (RBAC) is a mechanism to manage permissions in a smart contract by assigning roles to different addresses.

> Definition: Role-Based Access Control (RBAC) is a security mechanism that restricts network access based on a person's role within an organization. It involves defining roles and assigning permissions to those roles. Users are then assigned to specific roles, granting them the permissions associated with that role.

Key Concepts:

- Roles: Define different levels of authorization (e.g., moderator, minter, admin).
- Permissions: Assign specific actions or functions to each role.
- Access Control: Check if a user has the required role before allowing them to execute a function.

Implementing RBAC:

1. Define Roles: Define the roles you need for your smart contract.
2. Assign Roles: Assign roles to addresses.
3. Check Roles: In each function, check if the caller has the required role before executing the function.

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";

contract MyContract is AccessControl {
    bytes32 public const MINTER_ROLE = keccak256("MINTER_ROLE");

    constructor() {
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _setupRole(MINTER_ROLE, msg.sender);
    }

    function mint(address to, uint256 amount) public onlyRole(MINTER_ROLE)
        // Mint tokens
    }
}
```

In this example, the `AccessControl` contract from OpenZeppelin is used to implement RBAC. The `MINTER_ROLE` is defined, and the constructor sets up the default admin role and assigns the minter role to the deployer. The `mint` function can only be called by addresses with the `MINTER_ROLE`.

Building Smart Contracts with OpenZeppelin

OpenZeppelin can guide you in building your own smart contracts.

Creating a Token with RS Access Control

The lecture demonstrates creating a token that is minable with RS access control. Note that OpenZeppelin versions change, so code might need adjustments when transitioning between major versions.

When creating ERC20, ERC721, or ERC1155 tokens, the OpenZeppelin wizard can be used to quickly pick desired options.

Running Scripts in Hardhat

- Scripts will be run using the Hardhat runtime environment.
- The syntax from previous lectures is necessary.

Here's an example script structure:

```
// Example script
async function main() {
  // Script logic here
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

The script reuses familiar elements:

- VM with Hardhat
- Structure of a `main` function
- Calling the `main` function
- Public client
- Pre-funded accounts inside the Hardhat environment

Deploying a Contract

When a contract like `MyToken` is deployed to an address using the Hardhat VM, it initially has minimal code (constructor and inheritance).

If you attempt to call a non-existent function (e.g., `playGame`), it will result in an error.

Inheritance and Function Calls

If a function is not defined in your contract but is part of an inherited contract (like ERC20), the call goes up the inheritance chain.

> Inheritance allows you to use functions from parent contracts without redefining them.

Even if a function (e.g., `totalSupply`) is private in the parent contract, it can be accessed through a public function in the parent.

Initial Total Supply

Running `npx hardhat run <script>` compiles and runs the script. Initially, the `totalSupply` is zero if not defined in the constructor.

Minting Tokens

By default, the ERC20 standard doesn't have a public `mint` function. The `_mint` function is internal and updates the balance of the recipient.

> Every ERC20 contract should implement its own minting logic based on specific conditions (e.g., games, lotteries, voting).

A centralized approach is to implement an initial supply, where the deployer controls the total supply.

```
Deployer Address --> Constructor --> 10 * (10 ^ decimals) tokens
```

Working with blockchain involves dealing with decimals. Tokens have decimal parts (like Satoshi in Bitcoin or Wei in Ethereum).

JavaScript/TypeScript Syntax

For writing scripts, basic JavaScript/TypeScript syntax is needed:

- Variables
- Definitions
- Function calls

A tutorial like "Learn TypeScript in Y Minutes" can help grasp the necessary syntax. For application development, a deeper understanding of JavaScript/TypeScript (especially React or backend) is required.

Token Deployment and Access Control

Initial Supply and Centralization Risks

- When deploying a smart contract, you can add an initial supply to put all the tokens in your possession.
- However, in production, it is generally not advisable to rely on a single address for handling all tokens due to the risk of losing access or theft.
- Role-based access control can be used to distribute power and combine different functions among multiple addresses.
    - Different people can be assigned roles such as approvers, minters, and auditors, with a control structure requiring multiple approvals for actions like minting.
- The lecture uses a simplified pattern where the deployer is granted the Minter role, similar to having a chairperson who can grant voting rights.

Combining ERC20 and Access Control

- The lecture demonstrates how to combine ERC20 and Access Control functionalities into a single smart contract.
    - This allows you to create a token that is both an ERC20 token and has access control features.

```
// Importing ERC20 and AccessControl
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/AccessControl.sol";

// Joining ERC20 and AccessControl in a single smart contract
contract MyToken is ERC20, AccessControl {
    // ...
}
```

- The resulting token is an ERC20 token with a modified mint function that can only be called by those with the Minter role.

Role-Based Access Control Details

Role-Based Access Control: Grants different roles like moderator, minter, and admin to different people. Anyone can check if a caller has a specific role by applying the onlyRole modifier to a function.

- The `onlyRole` modifier can be used similarly to `onlyOwner`.

Setting Multiple Roles

- You can set multiple `onlyRole` modifiers for a function.

```
function myfunction() public onlyRole(ROLE_1) onlyRole(ROLE_2) {
    // Function logic here
}
```

- Alternatively, you can create a new role that combines the permissions of two other roles.
    - This can be useful if you want to remove a person from one role but not the other.

Risks of Mixing Contracts

- Altering and combining contracts increases risk.
- It is crucial to understand the implications of combining different components.
- Analogy: Assembling a bomb. Each component is safe on its own, but combining them incorrectly can lead to dangerous consequences.

Gas Considerations

- Adding more logic to constructors increases gas costs and makes deployment more expensive.
- Everything you inherit from other contracts adds to the bytecode and increases deployment costs.
- More complex functions cost more gas to call.

Demonstrating Access Control in Scripts

- The lecture demonstrates fetching a role code from a smart contract in a script.
- The script attempts to deploy the smart contract and call the mint function from a different account.
- It showcases deploying with one account and calling from another.
- Attempting to mint from an account without the Minter role results in an Access Control unauthorized account error.

Error Handling and Role Management

When an error occurs, such as an "unauthorized account" error, it indicates that the account trying to perform an action, like minting tokens, is missing a required **role**. In the example, `account 0x3c44` was missing the role `0x9f2d`.

- **Gas Efficiency**: Using a **hash** or **code** (like `0x9f2d`) for roles instead of a string is more **gas efficient**. While strings are more readable, they cost more gas. Hashes are difficult to collide, allowing roles to be translated to a 32-bit representation.

To resolve the authorization issue:

1. Grant the necessary role to the account before executing the minting transaction.
2. The deployer of the smart contract can grant roles using a function like `grantRole(roleCode, accountAddress)`.

Batch Data Retrieval with `Promise.all`

`Promise.all` allows you to fetch multiple pieces of information from a smart contract in a single call. This is more efficient than making multiple individual calls.

- Example: Fetching the name, symbol, decimals, and total supply of a smart contract in one go.

Benefits:

- Reduces the number of calls to the blockchain.
- Improves efficiency.
- Useful in front-end development for gathering necessary data at once.

```
Promise.all([
    contract.name(),
    contract.symbol(),
    contract.decimals(),
    contract.totalSupply()
]).then(([name, symbol, decimals, totalSupply]) => {
    console.log({ name, symbol, decimals, totalSupply });
});
```

⁉ Decimal Precision in Token Minting

When minting tokens, the **decimal precision** is crucial. Minting "10 tokens" without considering decimals can lead to unexpected results.

> **Example**: Minting 10 units with 18 decimals will result in a fraction of a token, not 10 whole tokens.

- **Analogy**: Minting dollars vs. minting cents. 1 million cents is only $10,000, not 1$ million.

Token Creation and the Zero Address

When tokens are minted, they are created "out of thin air."

> The tokens don't come from any existing balance but are rather created by the minting process.

- The `from` value in the transfer event is set to the **address zero** (`0x0...0`).
- This doesn't imply a transfer of balance from address zero; instead, it signifies the creation of new tokens.

Transferring Tokens and Checking Balances

After transferring tokens, you can check account balances to confirm the transaction.

- A function like `balanceOf(accountAddress)` can be used to get the balance of an account.

Example: Transfer tokens from the deployer to `account1` and verify the updated balance.

Formatting Token Balances with `formatEther`

Reading raw token balances (especially with many decimals) can be difficult.

- The `formatEther` function (imported from `ethers`) converts raw balances into a more readable format.

```
import { ethers } from "ethers";

// Assume 'balance' is a BigNumber representing the raw balance
const formattedBalance = ethers.utils.formatEther(balance);
console.log(formattedBalance); // Output: e.g., "18.0 MTK"
```

- Caution: Be aware of the number of decimals defined in the smart contract. If the contract uses a different number of decimals (e.g., 8 instead of 18), use `formatUnits` instead.

Scripts for Testing Smart Contracts

- You can write scripts to test the functions of smart contracts:
  - `allowance`
  - `burn`
- These scripts help you understand how the functions work.
- Optional homework: write scripts to test the functions of ERC20 and ERC721 tokens.
- You can use TypeScript for quick testing.

Understanding Values in Smart Contracts

- Constants: Values, like decimal values, are constant after deployment.
- Display Purposes: These values are mainly for display.
- Arithmetic: Calculations are done in actual decimal values, not display values.
- You can mint tokens to your MetaMask wallet if you deploy a token with your personal MetaMask wallet address.
- The code for public and local testnets should be interchangeable.

Events in Smart Contracts

- Events are crucial for front-end development.
- They allow you to listen to information from smart contracts in front-ends or trading bots.

Native Tokens vs. ERC20 Tokens

- **Native Tokens**: Normal transaction data (sender, receiver, gas) is included in a block.
- **ERC20 Tokens**: Additional information isn't indexed in the block.
    - The value is zero, so you don't have the logging abilities that exist for native tokens.
    - Need to create events to index the data.

How Events Work

1. **Smart Contract Call**: A smart contract is called, but the blockchain sees it as a zero ether transaction.

2. **Event Emission**: Code an event in the smart contract (e.g., `emit`).

    ```
    emit Transfer(address from, address to, uint256 value);
    ```

3. **Log Creation**: When the event is emitted, it creates a log inside the node's structure.

    - Linked with blockchain data and the contract's address.
    - Incorporated into the blockchain.

4. **Accessibility**: The log is not accessible within the contract itself.

    - More useful for off-chain applications, front-ends, and back-ends.

Limitations of Events

Events are not on-chain data that you can read from smart contracts.

- The logging functionality proves that those logs happened, and then you can listen for them off-chain.
- You can't use events for historical averages or triggering reactions between contracts on-chain.
- Need to build off-chain applications for this.

Example of Testing Events

1. Create a test file: `eventTest.ts` in the test folder.

2. Write a test:

```
// Example test scenario
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("Token", function () {
  it("Should trigger the Transfer event with correct parameters", asy
    const [owner, addr1, addr2] = await ethers.getSigners();

    const Token = await ethers.getContractFactory("MyToken");
    const hardhatToken = await Token.deploy();

    await hardhatToken.deployed();

    // Transfer tokens
    await expect(hardhatToken.transfer(addr1.address, 100))
      .to.emit(hardhatToken, 'Transfer')
      .withArgs(owner.address, addr1.address, 100);
  });
});
```

3. Run the test: `npx hardhat test`

Benefits of events

- Events are a way to pack and structure data when we do transactions.
- Events can be coded for various actions:
    - Birthdays
    - Account creation

Events in Solidity

Indexed Members and Data Values

- **Events** in Solidity allow you to log activities that occur within your smart contract.
- You can specify which parts of the events are **indexed** to enable listening, triggering watch events, and executing back-end and front-end operations.
- **Indexed members** allow filtering for specific events.
- You can also have normal **data values** within an event.
- A limitation: you can only have **four indexed members**.
- The **topic** serves as one of the indexed parts, containing event data.
- Additional data that doesn't fit into the indexed parts can be included as normal data. Note that this data is not searchable or watchable but is visible as a packet structured inside the event log.

Event Usage Timeline

- Events will not be used this week.
- Events will be used in front-end applications next week.
- Understanding **ERC20** events is essential.