**Setup**

**Project Setup: Building the TSender Airdrop UI**

Welcome! In this lesson, we'll lay the groundwork for building a user interface (UI) for the TSender smart contract. This front-end application will allow users to interact with the contract's token airdropping functionality. We'll be using TypeScript, React, and Next.js for this project.

If you were following along with the previous 'html-ts-coffee-cu' project, make sure your Live Server extension (if used) is turned off before proceeding.

Let's begin by setting up our project directory. Open your terminal. If you're currently inside a previous project directory (like html-ts-coffee-cu), navigate up one level using cd ... Clear your terminal for better visibility with clear. Confirm your current location with pwd.

Now, create the dedicated directory for our new project:

mkdir ts-tsender-ui-cu

The name signifies 'TypeScript TSender UI - Cyfrin Updraft'.

Finally, open this new directory in VS Code. You can do this directly from the terminal (if VS Code is in your path) with:

code ts-tsender-ui-cu/

Alternatively, use the File -> Open Folder menu within VS Code.

Before we start coding, let's define our objective. We are building a front-end application to interact with the TSender smart contract. This contract facilitates airdropping ERC20 tokens to multiple recipients in a single transaction.

You can view the final code for this UI project at https://github.com/Cyfrin/ts-tsender-ui-cu and the associated smart contract code (including a reference implementation) at https://github.com/Cyfrin/TSender/. These resources are helpful if you want to see the end result or review the contract logic we'll be interacting with.

Let's create a README.md file in our new project root (ts-tsender-ui-cu/) to outline our goals.

The core functionality we need to build our UI around is the airdropERC20 function within the TSender contract. While the main TSender contract uses Huff (a low-level language), a reference implementation in Solidity (TSender/src/reference/TSenderReference.sol) helps illustrate the logic:

function airdropERC20(

address tokenAddress, // The address of the ERC20 token

address[] calldata recipients, // Addresses to receive the tokens

uint256[] calldata amounts, // Amounts each recipient gets

uint256 totalAmount // Sum of all amounts (for verification)

) external {

// ... implementation details transferring tokens ...

}

This function takes the address of the ERC20 token contract, lists of recipient addresses and corresponding amounts, and the total amount to be sent. Our UI will need to gather this information from the user and pass it to this function when calling the smart contract.

To interact with airdropERC20, we need to understand ERC20 tokens. An ERC20 token represents a fungible asset (like USDC, DAI, etc.) on an EVM-compatible blockchain. It's defined by a standard interface implemented as a smart contract deployed on the blockchain. This is different from the blockchain's native currency (like ETH or MATIC), which is used primarily for gas fees. Each ERC20 token *is* a smart contract and has its own unique address (you can find examples like USDC on block explorers like Etherscan). For a deeper dive, the first few minutes of the 'Create your own Blockchain ERC20 Token' video (linked in the main course repository) provide a good overview.

Crucially, the recipients and amounts parameters in airdropERC20 work as parallel arrays. The address at recipients[0] receives the amount specified at amounts[0], recipients[1] receives amounts[1], and so on. For example:

* recipients = [address\_A, address\_B]
* amounts = [50, 150]

This means address\_A gets 50 tokens, and address\_B gets 150 tokens.

The totalAmount parameter acts as a safeguard; it must equal the sum of all values provided in the amounts array for the transaction to succeed.

Now, let's outline the steps we'll take in our README.md to build this application:

1. **Create a basic React/Next.js application:** Set up the initial front-end project structure using Next.js, which provides a robust framework compared to plain HTML/JavaScript.
2. **Implement the**airdropERC20**interaction:** Write the core logic in our front end to collect user input (token address, recipient list, amounts list) and use libraries (like Viem/Wagmi) to construct and send the transaction to the airdropERC20 function on the deployed TSender contract.
3. **Deploy to Fleek:** Make our application publicly accessible by deploying the built Next.js application to decentralized storage (IPFS) via Fleek.

The goal here is significant: by the end of this section, you will have built and deployed a fully functional, full-stack decentralized application (dApp), like the example visible at t-sender.com. Having a user-friendly front end is essential for making smart contracts accessible and usable. Don't underestimate this achievement – deploying a live dApp is a major milestone worth celebrating!

With our environment set up and our plan defined, we're ready to start building. In the next lesson, we'll focus on creating the basic Next.js application structure.