**Introduction**

**Building the TSender UI: A Gas-Optimized ERC20 Airdrop Frontend**

Welcome to this next stage of your web3 development journey. We're leveling up in complexity as we build a practical and powerful frontend application: the TSender UI. This project focuses on creating a user interface for a highly gas-optimized smart contract designed for efficiently airdropping ERC20 tokens to multiple recipients simultaneously.

Throughout this build, remember a crucial principle of web3 development: security and correctness are paramount. Unlike traditional web development, mistakes in web3 can lead to irreversible loss of funds. We must be diligent to avoid potential vulnerabilities.

Our goal is to build a static frontend application using React and Next.js that replicates the core functionality of the live t-sender.com tool. This UI will interact with the blockchain, specifically with ERC20 token contracts and the underlying TSender smart contract, which is engineered for maximum gas savings using the Huff language.

**Understanding the TSender UI Functionality**

The TSender UI provides a streamlined interface for performing batch ERC20 token transfers. Let's break down the target functionality, mirroring the t-sender.com application:

1. **Wallet Connection:** Users need to connect their web3 wallet (like MetaMask) to the application. We will integrate libraries like Rainbow Kit to provide a smooth and standard connection experience.
2. **Network Management:** The UI must support multiple EVM-compatible blockchain networks (e.g., Ethereum Mainnet, Optimism, Arbitrum, Base, ZK Sync, Sepolia testnet, and local testnets like Anvil). Users will be able to easily switch between supported networks directly within the interface. Our development will often utilize ZK Sync and Anvil.
3. **Input Gathering:** The core interface requires several inputs from the user:
   * **Token Address:** The contract address of the specific ERC20 token the user wishes to airdrop (e.g., USDC on ZK Sync).
   * **Recipient Addresses:** A list of wallet addresses to receive the tokens. These can be entered separated by commas or new lines in a dedicated text area.
   * **Token Amounts:** A corresponding list of token amounts (specified in wei, the smallest unit of the token) to be sent to each recipient. These must also be separated by commas or new lines, and the number of amounts must exactly match the number of recipient addresses.
4. **Dynamic Transaction Preview:** Based on the inputs, the UI will automatically fetch data from the blockchain and calculate:
   * **Token Information:** The name (e.g., "USDC") and decimal precision of the selected ERC20 token.
   * **Total Amount (wei):** The sum of all individual amounts entered.
   * **Total Amount (Formatted):** The total amount converted from wei to the standard token representation (e.g., 3,000,000 wei of USDC displayed as "3.00 USDC").
5. **Airdrop Execution:** A primary action button (e.g., "Send Tokens") initiates the two-step transaction process required for the airdrop:
   * **Approval (**approve**):** First, the user must sign a transaction granting the TSender smart contract permission to spend the specified total amount of the chosen ERC20 token from their wallet.
   * **Transfer (**airdrop**):** Second, the user signs the main transaction that calls the TSender smart contract function, which then efficiently distributes the tokens to all listed recipients.

The UI will also include toggles for different modes, such as "Safe Mode" and "Unsafe Mode", though the specifics of these modes are beyond this introductory overview.

**Core web3 Concepts in This Project**

Building the TSender UI involves several fundamental web3 concepts:

* **ERC20 Token Standard:** We'll interact heavily with ERC20 contracts – fetching data like name and decimals, and understanding the approve mechanism required for delegated transfers.
* **Token Airdrops:** The primary use case is batch token distribution, a common operation in web3 for rewards, community building, etc.
* **Gas Optimization:** While we are building the *frontend*, it's crucial to understand that the underlying TSender smart contract is written in Huff (a low-level EVM language) specifically to minimize the transaction costs (gas fees) associated with airdrops. Our UI leverages this efficiency.
* **Multi-Chain Architecture:** Modern web3 applications often need to function across various blockchain networks. Our frontend must be designed to handle network switching and interactions on different chains.
* **Wallet Interaction:** Securely connecting user wallets and prompting them to sign transactions (approve and the main airdrop call) is fundamental. Libraries like Wagmi and Rainbow Kit abstract much of this complexity.
* **web3 Security:** The constant need for vigilance against potential exploits. Every interaction with user funds or smart contracts requires careful implementation.

**Technology Stack for TSender UI**

This project utilizes a modern frontend stack tailored for web3 development:

* **React:** A popular JavaScript library for building user interfaces with a component-based architecture.
* **Next.js:** A framework built on React, providing features like static site generation (SSG), routing, and optimizations, making it ideal for performant web applications. We will be leveraging its SSG capabilities.
* **TypeScript:** Adds static typing to JavaScript, helping catch errors during development and improving code maintainability, especially in larger projects.
* **Wagmi:** A collection of React Hooks simplifying blockchain interactions. It provides tools for connecting wallets, reading contract data, executing transactions, and more. (wagmi.sh)
* **Rainbow Kit:** Builds upon Wagmi to provide polished UI components for wallet connection and network switching, enhancing the user experience.
* **Anvil / Foundry:** Used for local blockchain development and testing, providing a fast and configurable local testnet environment.
* **Fleek:** A platform for deploying static websites to decentralized storage solutions like IPFS, allowing for censorship-resistant and potentially free hosting. (fleek.xyz)

**Exploring the Codebase**

The codebase for this project, found at https://github.com/Cyfrin/ts-sender-ui-cu, represents a significant step up in scale compared to previous lessons. It's structured as a typical Next.js application with directories like src, components, utils, test, and configuration files such as next.config.js and tsconfig.json.

As an example of configuration, the rainbowKitConfig.tsx file sets up the wallet connection UI and specifies the supported chains using imports from Wagmi:

// Example structure from rainbowKitConfig.tsx

import { getDefaultConfig } from '@rainbow-me/rainbowkit';

import { mainnet, optimism, arbitrum, base, zksync, sepolia, anvil } from 'wagmi/chains';

​

// Note: Actual implementation will require secure handling of Project ID

const walletConnectProjectId = process.env.NEXT\_PUBLIC\_WALLETCONNECT\_PROJECT\_ID || "";

​

export const config = getDefaultConfig({

appName: 'TSender',

projectId: walletConnectProjectId, // Use WalletConnect Cloud Project ID

chains: [mainnet, optimism, arbitrum, base, zksync, sepolia, anvil],

ssr: false, // Typically set to false for static exports or client-heavy apps

});

It's also important to note the repository for the *underlying* smart contract: https://github.com/Cyfrin/TSender. While we won't be writing Huff code, understanding that the core logic resides in src/protocol/TSender.huff helps appreciate the gas optimization aspect of the tool we're interfacing with.

**Deployment and Final Thoughts**

Our final product will be a static website generated by Next.js. This means it consists of pre-built HTML, CSS, and JavaScript files. This static output can be efficiently hosted on platforms like Fleek, often for free, leveraging decentralized storage networks.

Remember, this project integrates multiple powerful tools and concepts. Pay close attention to how React, Next.js, Wagmi, and Rainbow Kit work together to create a seamless user experience for complex blockchain interactions. Always prioritize security and thorough testing as you build. Let's begin constructing the TSender UI.