**Review Pt.2**

**Review: Building the TSender Frontend with Next.js and web3 Tools**

This lesson serves as a comprehensive review of the frontend development phase for our TSender application. We've covered significant ground, integrating modern web technologies with web3 libraries to build a functional decentralized application (dApp) interface. Let's recap the key technologies, concepts, and philosophies underpinning this section.

\*\*Core Philosophy: Practice, Deployment, and Vigilance\*\*

Before diving into the specifics, remember these guiding principles:

1. \*\*Repetition is the Mother of Skill:\*\* Building and deploying applications repeatedly is the most effective way to master the tools and concepts involved in web3 development. Don't just follow along; rebuild, experiment, and deploy often.

2. \*\*Deployment Matters:\*\* Theory is one thing, but practical experience is invaluable. Ensure you have deployed your TSender frontend, perhaps using Fleek as demonstrated. This step solidifies your understanding and prepares you for real-world scenarios.

3. \*\*Testing is Non-Negotiable:\*\* Given the potential financial implications of bugs in web3, rigorous testing is essential. We explored both unit and end-to-end testing to ensure our application behaves as expected.

4. \*\*AI as a Supervised Tool:\*\* While AI tools like ChatGPT and Copilot significantly accelerated UI development and boilerplate generation, they are not infallible. Treat AI as a helpful but potentially flawed assistant. \*\*You must review, understand, and verify every line of AI-generated code\*\*, especially when dealing with user funds or smart contract interactions. Blind trust can lead to critical security vulnerabilities.

\*\*Technology Stack Deep Dive\*\*

We leveraged a powerful stack to create a robust and maintainable frontend:

1. \*\*Next.js & React:\*\*

\* \*\*Rationale:\*\* Chosen over simpler HTML/JS setups to build a production-ready, modular application. This framework allows us to blend TypeScript/JavaScript logic seamlessly with HTML structure within reusable components.

\* \*\*Architecture:\*\* We adopted a component-based structure, organizing code into directories like `src/app`, `src/components`, and `src/utils`.

\* \*\*Key Components:\*\*

\* `Header.tsx`: Managed the application title and wallet connection elements.

\* `AirdropForm.tsx`: Served as the core component, handling user inputs (token address, recipients, amounts) and orchestrating the airdrop logic.

\* `InputField.tsx`: A reusable UI component, potentially generated by AI, for handling text inputs and text areas.

\* `HomeContent.tsx`: Acted as a container for the main page content.

\* \*\*React Hooks:\*\* Utilized extensively for state management and handling asynchronous operations:

\* `useState`: Managed the state of form inputs.

\* `useMemo`: Employed for optimizing potentially expensive calculations, such as deriving the total amount to be sent (`calculateTotal`).

2. \*\*Wagmi:\*\*

\* \*\*Purpose:\*\* This library provides a collection of React Hooks specifically designed for Ethereum Virtual Machine (EVM) chain interactions, greatly simplifying tasks like wallet connection, reading chain data, and sending transactions.

\* \*\*Hooks Used:\*\*

\* `useChainId`, `useConfig`, `useAccount`: Provided essential information about the connected blockchain, configuration, and the user's account details.

\* `useWriteContract`: The cornerstone for initiating on-chain transactions. It was used for both the `approve` and `airdropERC20` calls, returning transaction data, hash, pending status (`isPending`), and the crucial `writeContractAsync` function to trigger the transaction.

\* \*\*Core Functions:\*\*

\* `readContract`: Used for querying data from smart contracts without sending a transaction (e.g., checking the current ERC20 token allowance).

\* `waitForTransactionReceipt`: Essential for confirming that a submitted transaction (initiated via `writeContractAsync`) has been successfully mined and included in a block.

3. \*\*RainbowKit:\*\*

\* \*\*Purpose:\*\* Provided a polished, user-friendly "Connect Wallet" button and modal experience.

\* \*\*Functionality:\*\* Handled the complexities of wallet selection, connection status display, showing the user's address and balance, and facilitating network switching, all configured within `rainbowKitConfig.tsx`.

4. \*\*TypeScript:\*\* Employed throughout the frontend codebase to enhance type safety, reduce runtime errors, and improve developer experience.

\*\*Understanding the ERC20 Approve/Airdrop Flow\*\*

A critical piece of logic involved interacting with ERC20 tokens. Since our `TSender` smart contract needs to spend tokens \*on behalf of the user\*, we implemented the standard two-step ERC20 interaction pattern:

1. \*\*Check Allowance:\*\* Before attempting the airdrop, we use `readContract` (via Wagmi) to check how many tokens the user has currently authorized (approved) the `TSender` contract to spend for the specific ERC20 token.

2. \*\*Approve (if necessary):\*\* If the current allowance is less than the total amount required for the airdrop, the user must first grant permission. This is done by calling the `approve` function on the \*ERC20 token contract\* itself, specifying the `TSender` contract address as the spender and the required amount. This interaction uses `writeContractAsync`. We then use `waitForTransactionReceipt` to confirm the approval transaction is successful.

```typescript

// Conceptual snippet within AirdropForm.tsx's submission logic

const currentAllowance = await readContract(config, {

abi: erc20Abi,

address: tokenAddress as `0x${string}`,

functionName: 'allowance',

args: [userAddress as `0x${string}`, tSenderAddress as `0x${string}`], // Check allowance for TSender

});

if (currentAllowance < totalToSendBigInt) {

console.log("Approval required...");

const approvalHash = await writeContractAsync({

abi: erc20Abi,

address: tokenAddress as `0x${string}`,

functionName: 'approve',

args: [tSenderAddress as `0x${string}`, totalToSendBigInt], // Grant approval to TSender

});

// Wait for approval transaction confirmation

const approvalReceipt = await waitForTransactionReceipt(config, { hash: approvalHash });

console.log("Approval confirmed:", approvalReceipt.status);

if (approvalReceipt.status !== 'success') {

throw new Error("Approval transaction failed");

}

}

```

3. \*\*Execute Airdrop:\*\* Once sufficient allowance is confirmed (either pre-existing or newly approved), we call the `airdropERC20` function on our `TSender` smart contract, again using `writeContractAsync`. This function takes the token address, processed lists of recipient addresses and amounts, and potentially the total amount.

```typescript

// Conceptual snippet following successful approval (if needed)

console.log("Executing airdrop...");

const airdropHash = await writeContractAsync({

abi: tsenderAbi, // ABI of your TSender contract

address: tSenderAddress as `0x${string}`, // Address of your deployed TSender contract

functionName: 'airdropERC20',

args: [

tokenAddress as `0x${string}`,

processedRecipients, // Your array of recipient addresses

processedAmounts, // Your array of corresponding amounts (as BigInts)

]

});

// Wait for airdrop transaction confirmation

const airdropReceipt = await waitForTransactionReceipt(config, { hash: airdropHash });

console.log("Airdrop transaction status:", airdropReceipt.status);

// Handle success or failure UI feedback

```

\*\*Implementing a Robust Testing Strategy\*\*

We emphasized a two-pronged testing approach:

1. \*\*Unit Testing (Vitest):\*\*

\* \*\*Goal:\*\* To test individual functions or modules in isolation, ensuring core logic is correct.

\* \*\*Example:\*\* We created tests for the `calculateTotal` utility function (`src/utils/calculateTotal/calculateTotal.test.ts`) to verify it accurately sums amounts from strings containing various delimiters and handles potential errors or edge cases gracefully.

\* \*\*Execution:\*\* `pnpm run test:unit`

2. \*\*End-to-End Testing (Playwright + Synpress):\*\*

\* \*\*Goal:\*\* To simulate real user interactions within a browser, testing the entire application flow, including crucial wallet interactions.

\* \*\*Tools:\*\*

\* \*\*Playwright:\*\* A powerful browser automation framework for E2E testing.

\* \*\*Synpress:\*\* A specialized wrapper around Playwright designed for dApp testing. It automatically sets up a browser (like Chromium) with MetaMask installed and provides commands to programmatically interact with the wallet (e.g., connect, approve transactions).

\* \*\*Example Tests:\*\*

\* Verifying the application title is correct.

\* Ensuring the Airdrop form only becomes visible \*after\* the user successfully connects their wallet.

\* (Potentially) Automating the entire airdrop flow, including wallet confirmations handled by Synpress.

\* \*\*Execution:\*\* `pnpm exec playwright test` (This command launches the browser, runs the tests, and interacts with the Synpress-managed MetaMask instance).

\*\*Deployment to the Decentralized Web with Fleek\*\*

To make our TSender frontend accessible, we deployed it using Fleek:

\* \*\*Purpose:\*\* Fleek facilitates hosting static websites (like our Next.js build output) on the InterPlanetary File System (IPFS), providing decentralized and censorship-resistant hosting.

\* \*\*Method:\*\* We primarily used the Fleek CLI for a streamlined deployment process, although the web UI is also an option.

\* \*\*Result:\*\* A live, publicly accessible website hosted decentrally, with a URL like `your-unique-name.on-fleek.app` or linked to a custom domain (e.g., `t-sender.com`).

\*\*Conclusion: Consolidating Your Frontend Skills\*\*

Congratulations on completing this intensive frontend section! You've integrated React, Next.js, TypeScript, Wagmi, and RainbowKit to build a sophisticated interface for a web3 application. You've also learned the critical importance of the approve/transfer pattern, implemented robust unit and E2E tests (including wallet interactions via Synpress), and deployed your application to the decentralized web using Fleek.

Remember the power of repetition and the necessity of understanding every line of code, especially when leveraging AI tools. The jump in complexity from basic web development is significant, but by building, testing, and deploying, you are solidifying the skills needed for modern dApp development. Take a moment to appreciate your progress before moving forward.