**Challenge**

**Enhancing Your dApps App: Implementation Challenges**

At this stage, you should have a foundational dApps web application. It can accept a token contract address, lists of recipient addresses and corresponding amounts (in wei), and initiate the token transfer process via a smart contract using MetaMask when you click the "Send Tokens" button.

While functional, this version lacks the polish expected of a user-friendly application. The user experience can feel clunky, especially during asynchronous operations or if the user accidentally refreshes the page.

This lesson presents a series of challenges designed to significantly improve the dApps application's usability and provide valuable learning experiences in frontend web3 development. We won't walk through the implementation step-by-step; instead, the goal is for you to tackle these enhancements yourself, utilizing the hints and concepts provided.

**Challenge 1: Implement Loading/Pending Indicators**

**Goal:** Provide clear visual feedback to the user while the application is waiting for asynchronous operations to complete. This prevents confusion and reassures the user that the application is working.

**Specifics:**  
Modify the "Send Tokens" button or the surrounding UI to display loading indicators (like spinners) during two key phases:

1. **Waiting for Wallet Confirmation:** After the user clicks "Send Tokens" but before they have approved the transaction in their MetaMask wallet. Display a message like "Confirming in wallet..." alongside a spinner.
2. **Waiting for Transaction Mining:** After the user confirms the transaction in MetaMask, while the transaction is being processed and mined on the blockchain. Display a message like "Mining transaction..." or "Sending transaction..." with a spinner.

**Hints:**

* Use component state variables (e.g., isPending, isConfirming) to track the application's status.
* Implement conditional rendering within your button component. Based on the state variables, render either the default button text, an error message, or the appropriate loading indicator and text.
* Consider using a library for spinner components (like react-spinners or an icon library like react-icons which includes spinners, e.g., CgSpinner).

// Simplified conceptual structure for button content logic

function getButtonContent(state) {

if (state.isPending) {

return <div><Spinner /><span>Confirming in wallet...</span></div>;

}

if (state.isConfirming) {

return <div><Spinner /><span>Mining transaction...</span></div>;

}

if (state.error) {

return <span>Error occurred.</span>

}

if (state.isConfirmed) {

return <span>Transaction Confirmed!</span>

}

// Default button text

return "Send Tokens";

}

​

// Within your button component's render method:

<button disabled={state.isPending || state.isConfirming} onClick={handleSubmit}>

{getButtonContent(appState)}

</button>

**Challenge 2: Persist User Inputs with Local Storage**

**Goal:** Prevent users from losing their entered data (token address, recipient list, amount list) if they accidentally close the tab or refresh the page.

**Specifics:**  
Utilize the browser's local storage to save the content of the "Token Address", "Recipients", and "Amounts" input fields. When the application loads, check local storage for any saved data and automatically populate the input fields if found.

**Hints:**

* Use the localStorage.setItem('key', value) method to save data. This should typically happen whenever the input field's corresponding state variable changes.
* Use the localStorage.getItem('key') method to retrieve data. This should happen when the component mounts (loads for the first time).
* React's useEffect hook is ideal for managing these side effects:
  + A useEffect with an empty dependency array ([]) runs once when the component mounts – suitable for retrieving data from local storage.
  + Separate useEffect hooks with specific dependencies (e.g., [tokenAddress]) run whenever that particular piece of state changes – suitable for saving data to local storage.

// Conceptual structure using React hooks

​

// Retrieve data on component mount

useEffect(() => {

const savedAddress = localStorage.getItem('tSenderTokenAddress');

const savedRecipients = localStorage.getItem('tSenderRecipients');

const savedAmounts = localStorage.getItem('tSenderAmounts');

if (savedAddress) setTokenAddress(savedAddress);

if (savedRecipients) setRecipients(savedRecipients);

if (savedAmounts) setAmounts(savedAmounts);

}, []); // Empty dependency array: run only on mount

​

// Save data when tokenAddress state changes

useEffect(() => {

if (tokenAddress) { // Avoid saving initial empty state if desired

localStorage.setItem('tSenderTokenAddress', tokenAddress);

}

}, [tokenAddress]); // Dependency: run when tokenAddress changes

​

// Similar useEffect hooks needed for recipients and amounts states

**Challenge 3: Display Dynamic Token Transaction Details**

**Goal:** Provide the user with immediate, clear information about the token they are interacting with and the total amount they are preparing to send, based on their inputs.

**Specifics:**  
Add a new section or display area in your UI that shows the following details, updating dynamically as the user enters information:

1. **Token Name:** Read the name property directly from the ERC20 token contract specified in the "Token Address" input.
2. **Total Amount (wei):** Calculate the sum of all valid numerical values entered in the "Amounts" input field.
3. **Total Amount (tokens):** Convert the calculated total wei amount into the standard token unit display. This requires reading the decimals property from the token contract and using it to format the total wei value correctly (e.g., dividing by 10 \*\* decimals). This value should update as the user types in the amounts field.

**Hints:**

* You'll need to interact with the token contract on the blockchain to read its name and decimals.
* Use a library like Wagmi to simplify blockchain interactions.
* **Performance Tip:** Instead of making multiple separate calls (e.g., using useReadContract for name, then again for decimals), use the useReadContracts hook (plural) from Wagmi. This hook allows you to batch multiple read calls into a single request, improving performance.
* Parse the amounts input (remembering it can be comma or newline separated) and sum the values. Ensure robust error handling for non-numeric inputs.
* Use the fetched decimals value to format the total wei sum into a human-readable token amount (e.g., using ethers.utils.formatUnits or equivalent logic).

// Conceptual structure using Wagmi's useReadContracts

​

import { useReadContracts } from 'wagmi';

import { erc20Abi } from 'viem'; // Assuming you have the ERC20 ABI

​

// Inside your component...

const { data: tokenDetails, isLoading: isLoadingTokenDetails } = useReadContracts({

allowFailure: false, // Or true, depending on how you want to handle errors

contracts: [

{ // Call 1: Get token name

address: tokenAddress as `0x${string}`, // Ensure address is valid format

abi: erc20Abi,

functionName: 'name',

},

{ // Call 2: Get token decimals

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

abi: erc20Abi,

functionName: 'decimals',

},

// You could add other reads here, like user balance if needed

// {

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

// abi: erc20Abi,

// functionName: 'balanceOf',

// args: [userAddress]

// }

],

// Conditionally enable the hook, e.g., only run if tokenAddress is valid

// query: { enabled: isValidAddress(tokenAddress) }

});

​

// Access results (tokenDetails will be an array):

const tokenName = tokenDetails?.[0]; // Result of the 'name' call

const tokenDecimals = tokenDetails?.[1]; // Result of the 'decimals' call

​

// Calculate totalWei from the amounts input state

// const totalWei = calculateTotalWei(amounts);

​

// Format the total amount using fetched decimals

// const formattedTotalAmount = formatAmount(totalWei, tokenDecimals);

​

// Render tokenName, totalWei, and formattedTotalAmount in your UI,

// handling loading and error states.

**Guidance and Final Thoughts**

Successfully implementing these features will significantly elevate your dApps application. Don't hesitate to:

* **Use AI Assistants:** Tools like ChatGPT or GitHub Copilot can be excellent aids for generating boilerplate, debugging, or explaining concepts.
* **Consult Example Code:** Refer to the completed example codebase provided in the course materials (github.com/Cyfrin/ts-tsender-ui-cu) if you get stuck or want to see one possible solution.
* **Be Patient:** These challenges may take time – hours or even a couple of days. The learning process involved in researching, implementing, and debugging is the primary goal.
* **Embrace the Process:** Actively working through these problems, looking up documentation (like Wagmi's docs), using AI, and analyzing working code is how you build robust development skills. Remember: repetition is the mother of skill.

Take the time now to pause and attempt these challenges. Good luck!