**Wagmi Send A Tx**

**Sending ERC20 Tokens with Wagmi: The Approve and Transfer Flow**

This lesson guides you through implementing ERC20 token transfers, specifically focusing on scenarios like airdrops where a smart contract sends tokens on behalf of a user. We'll use the Wagmi library in a React application to handle the crucial two-step process: approving the contract to spend tokens and then executing the transfer.

**Understanding the ERC20 Approve/Transfer Mechanism**

Before a smart contract (let's call it the "spender," like our tsender contract example) can transfer ERC20 tokens owned by a user, the user must first grant permission. This is done by calling the approve function on the specific ERC20 token contract. The user approves the spender contract to withdraw up to a certain amount of tokens.

Therefore, sending tokens via a contract involves two potential blockchain interactions:

1. **Approval:** If the spender contract doesn't have sufficient allowance, the user must first send an approve transaction to the token contract.
2. **Transfer:** Once the necessary allowance is confirmed, the user can trigger the function on the spender contract (e.g., airdropERC20) that performs the actual token transfers.

**Step 1: Checking Existing Allowance**

First, determine if an approve transaction is necessary. Calculate the total amount of tokens required for the intended operation (e.g., the sum of all airdrop amounts). Then, read the current allowance the spender contract already has.

You can use Wagmi Core's readContract function to call the allowance function of the ERC20 token contract. The allowance function typically takes the owner's address and the spender's address as arguments and returns the approved amount.

// Assuming 'total' (BigInt) is calculated based on user input

// Assuming 'config', 'erc20Abi', 'tokenAddress', 'userAddress', 'tSenderAddress' are defined

​

import { readContract } from '@wagmi/core';

import { erc20Abi } from './abi'; // Your ERC20 ABI import

​

const approvedAmount = await readContract(config, {

abi: erc20Abi,

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

functionName: 'allowance',

args: [userAddress as `0x${string}`, tSenderAddress as `0x${string}`],

});

​

// Conditional check

if (approvedAmount < total) {

// Need to perform an approve transaction

} else {

// Sufficient allowance, can proceed directly to the main transaction

}

**Step 2: Initiating Transactions with**useWriteContract

When integrating contract interactions within a React UI, Wagmi's React hooks are highly beneficial as they manage transaction state (loading, success, error) and trigger UI updates. For write operations like approve and airdropERC20, we use the useWriteContract hook.

Import the hook and instantiate it in your component:

import { useWriteContract } from 'wagmi';

import { waitForTransactionReceipt } from '@wagmi/core'; // Also needed

import { config } from './wagmi'; // Your Wagmi config import

​

// Inside your React component

const { data: hash, isPending, error, writeContractAsync } = useWriteContract();

​

// Key return values:

// - writeContractAsync: Function to call to initiate the transaction. Returns a Promise resolving with the tx hash.

// - data (renamed to hash): Stores the transaction hash upon successful submission.

// - isPending: Boolean flag, true while the transaction is being sent to the wallet and submitted.

// - error: Contains error details if the submission fails.

Note: useWriteContract automatically accesses the Wagmi configuration provided via context, so you don't need to pass config directly to the hook itself.

**Step 3: Implementing the**approve**Transaction**

Inside the if (approvedAmount < total) block, use the writeContractAsync function returned by the hook to trigger the approve transaction on the ERC20 token contract.

// Inside the component, within the function handling the send logic

​

if (approvedAmount < total) {

try {

console.log(`Approval needed: Current ${approvedAmount}, Required ${total}`);

// Initiate Approve Transaction

const approvalHash = await writeContractAsync({

abi: erc20Abi, // ERC20 token ABI

address: tokenAddress as `0x${string}`, // ERC20 token address

functionName: 'approve',

args: [tSenderAddress as `0x${string}`, BigInt(total)], // Spender address and total amount

});

console.log("Approval transaction hash:", approvalHash);

​

// ---> Next: Wait for confirmation

​

} catch (err) {

console.error("Approval failed:", err);

// Handle UI feedback for error

return; // Stop the process if approval fails

}

} else {

console.log(`Sufficient allowance: ${approvedAmount}`);

// ---> Proceed directly to airdrop logic

}

​

**Important Notes:**

* **ABI & Address:** Provide the ABI and address of the *ERC20 token contract* you're interacting with.
* **Arguments (**args**):** The approve function requires the spender's address (tSenderAddress) and the amount (total) to approve.
* **BigInt:** Solidity's uint256 type maps to JavaScript's BigInt. Ensure numeric amounts passed as arguments are converted to BigInt.
* **Type Casting:** Use `0x${string}` to cast address strings for type compatibility with Viem/Wagmi.

**Step 4: Waiting for Transaction Confirmation**

Sending a transaction returns a hash almost immediately, but the transaction isn't confirmed (mined) yet. If the next step (the actual transfer) depends on the approve being successful, you *must wait* for the transaction receipt. Use the waitForTransactionReceipt function from @wagmi/core.

// Continuing inside the `if (approvedAmount < total)` block, after getting approvalHash

​

// Wait for the transaction to be mined

console.log("Waiting for approval confirmation...");

const approvalReceipt = await waitForTransactionReceipt(config, { // Pass config here!

hash: approvalHash,

});

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

​

// Optional: Check receipt status for success

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

console.error("Approval transaction failed:", approvalReceipt);

// Handle UI feedback for failed transaction

return;

}

​

// ---> Approval successful, now proceed to the airdrop logic

​

**Key Points:**

* waitForTransactionReceipt requires the Wagmi config object and the hash of the transaction to wait for.
* This function returns a promise that resolves with the transaction receipt once the transaction is included in a block.
* Checking receipt.status === 'success' is crucial to ensure the transaction didn't revert.

**Step 5: Implementing the Main Contract Call (e.g.,**airdropERC20**)**

Once the necessary approval is confirmed (either because it existed initially or the approve transaction just succeeded), you can call the main function on your spender contract (tsender contract in this example) using the same writeContractAsync function.

This call will happen in two places:

1. Inside the else block (if approvedAmount >= total).
2. Inside the if block, *after* waitForTransactionReceipt confirms the approve transaction succeeded.

// Define the airdrop function call logic (can be placed in a helper function or directly)

const executeAirdrop = async () => {

try {

console.log("Executing airdropERC20...");

// Prepare arguments - requires parsing user input

const recipientAddresses = recipients // Assuming 'recipients' is a string like "addr1, addr2\naddr3"

.split(/[, \n]+/) // Split by comma, space, or newline

.map((addr) => addr.trim()) // Remove whitespace

.filter(addr => addr !== '') // Remove empty entries

.map(addr => addr as `0x${string}`); // Cast to address type

​

const transferAmounts = amounts // Assuming 'amounts' is a string like "10, 20\n30"

.split(/[, \n]+/)

.map((amt) => amt.trim())

.filter(amt => amt !== '')

.map(amount => BigInt(amount)); // Convert amounts to BigInt

​

if (recipientAddresses.length !== transferAmounts.length) {

throw new Error("Mismatch between number of recipients and amounts.");

}

​

// Initiate Airdrop Transaction

const airdropHash = await writeContractAsync({

abi: tsenderAbi, // Spender contract's ABI

address: tSenderAddress as `0x${string}`, // Spender contract's address

functionName: 'airdropERC20',

args: [

tokenAddress as `0x${string}`, // 1. Token being sent

recipientAddresses, // 2. Array of recipient addresses

transferAmounts // 3. Array of amounts (BigInt)

]

});

console.log("Airdrop transaction hash:", airdropHash);

​

// Optional: Wait for airdrop confirmation if needed for further UI updates

console.log("Waiting for airdrop confirmation...");

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

console.log("Airdrop confirmed:", airdropReceipt);

// Update UI based on success/failure

​

} catch (err) {

console.error("Airdrop failed:", err);

// Handle UI feedback for error

}

};

​

// --- Integration into the main logic ---

​

if (approvedAmount < total) {

try {

// ... (Approval code as above) ...

const approvalHash = await writeContractAsync({ /\* ... approve args ... \*/ });

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

​

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

console.log("Approval successful, proceeding to airdrop.");

await executeAirdrop(); // Call airdrop AFTER successful approval

} else {

console.error("Approval transaction failed.");

// Handle UI feedback

}

} catch (err) {

console.error("Approval process error:", err);

// Handle UI feedback

}

} else {

console.log("Sufficient allowance, proceeding directly to airdrop.");

await executeAirdrop(); // Call airdrop directly

}

​

**Key Considerations:**

* **Input Parsing:** Robustly parse user input (like comma or newline-separated lists of addresses and amounts). Ensure you trim whitespace, filter empty entries, and convert data to the expected types (address and BigInt). Consider abstracting this parsing logic into reusable utility functions.
* **Contract Details:** Use the correct ABI and address for the spender contract (tsenderAbi, tSenderAddress).
* **Arguments:** Ensure the arguments passed match the airdropERC20 function signature in your Solidity contract (token address, array of recipient addresses, array of amounts).
* **Error Handling:** Implement try...catch blocks around writeContractAsync and check receipt status to handle potential transaction failures gracefully in the UI.

**Summary**

By combining readContract (or alternative allowance checks), the useWriteContract hook, and waitForTransactionReceipt, you can reliably implement the common ERC20 approve-and-transfer pattern in your React dApp. This involves:

1. Checking the current allowance.
2. If insufficient, prompting the user for an approve transaction using useWriteContract.
3. Waiting for the approve transaction to confirm using waitForTransactionReceipt.
4. Prompting the user for the main contract interaction (e.g., airdropERC20) using useWriteContract, either directly (if allowance was sufficient) or after successful approval.
5. Handling transaction states (isPending, error, receipt status) to provide feedback to the user.

This ensures that your contract interaction logic respects the ERC20 standard and provides a clear flow for users interacting with token contracts.