**Read Contract Wagmi**

**Reading Smart Contract Data with Wagmi**

This lesson focuses on the crucial first step in interacting with smart contracts that handle ERC20 tokens: reading data from the blockchain. Specifically, we'll implement the initial part of an airdrop function in a React application, using the wagmi library to check how many tokens a user has approved our airdrop contract to spend on their behalf.

**Understanding the ERC20 Approval Flow**

Before diving into the code, it's essential to understand the standard ERC20 approval mechanism. When you want a smart contract (like our tsender airdrop contract) to transfer ERC20 tokens *owned by a user*, the user must first grant permission. This is a two-step security measure:

1. **Approve:** The token owner calls the approve function on the *token contract* itself. They specify the address of the contract they are authorizing (the "spender" – in our case, the airdrop contract) and the maximum amount of tokens that spender is allowed to withdraw.
2. **TransferFrom:** The spender contract (our airdrop contract), when triggered by an action (like the user submitting the airdrop form), can then call the transferFrom function on the *token contract*. This function allows the spender to move tokens from the owner's address to another address, but only up to the amount previously approved.

Therefore, before our airdrop contract can execute the transferFrom logic (which happens inside its airdropERC20 function), our frontend application must:

1. **Check the current allowance:** Read the amount the user (token owner) has already approved for our airdrop contract (spender).
2. **Request approval if needed:** If the current allowance is less than the total amount required for the airdrop, prompt the user to execute an approve transaction.
3. **Execute the airdrop:** Once sufficient allowance is confirmed, call the function on the airdrop contract to perform the token transfers.

This lesson focuses on the first step: checking the current allowance.

**Wagmi Hooks vs. Core Actions**

Wagmi is a powerful React Hooks library that simplifies blockchain interactions. It offers two primary ways to interact with contracts:

1. **Wagmi Hooks (**use...**)**: Functions like useReadContract, useWriteContract, useAccount, useChainId, etc. These are designed specifically for React components. They integrate with React's state management, automatically update your UI when blockchain data changes, and handle caching and deduplication. They are ideal for displaying data that needs to be reactive (e.g., showing the user's balance).
2. **Wagmi Core Actions (**@wagmi/core**)**: Lower-level functions like readContract, writeContract, getAccount, getChainId, etc. These perform the underlying blockchain operations without being tied to React's component lifecycle or state. They are suitable for one-off actions triggered by events, like button clicks within an event handler (handleSubmit), where you need to perform an action immediately but don't necessarily need the result to be automatically reactive in the UI state for that specific call.

For checking the allowance within our handleSubmit function (which runs once when the button is clicked), using the readContract core action from @wagmi/core is more appropriate than the useReadContract hook.

**Managing Configuration with**constants.ts

Hardcoding contract addresses and Application Binary Interfaces (ABIs) directly into components makes applications brittle and hard to maintain, especially when deploying to multiple networks (each having different contract addresses). A best practice is to store this static configuration in a central location.

We'll create a src/constants.ts file to hold:

* A mapping of chain IDs to our deployed tsender contract addresses.
* The ABI for standard ERC20 functions (allowance, approve, balanceOf, etc.).
* The ABI for our specific tsender airdrop contract.

// src/constants.ts (Example Structure)

interface ContractsConfig {

[chainId: number]: {

tsender: string;

// Other contract addresses if needed

};

}

​

// Map chain IDs to deployed tsender contract addresses

export const chainsToTsSender: ContractsConfig = {

31337: { // Example: Anvil local network

tsender: "0x5FbDB2315678afecb367f032d93F642f64180aa3",

},

1: { // Example: Ethereum Mainnet

tsender: "0x...",

},

// Add other supported chains

};

​

// Standard ERC20 ABI (partial example)

export const erc20Abi = [

{ constant: true, inputs: [{ name: "\_owner", type: "address" }, { name: "\_spender", type: "address" }], name: "allowance", outputs: [{ name: "remaining", type: "uint256" }], type: "function" },

{ constant: false, inputs: [{ name: "\_spender", type: "address" }, { name: "\_value", type: "uint256" }], name: "approve", outputs: [{ name: "success", type: "bool" }], type: "function" },

// ... other ERC20 functions

] as const; // Using 'as const' for better type inference with wagmi/viem

​

// ABI for the tsender contract (partial example)

export const tsenderAbi = [

{ type: "function", name: "airdropERC20", inputs: [/\*...\*/], outputs: [], stateMutability: "payable" },

// ... other tsender functions

] as const;

Remember to import these into your component:

// src/components/AirdropForm.tsx

import { chainsToTsSender, erc20Abi } from "@/constants";

import { useChainId, useConfig, useAccount } from 'wagmi';

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

import { useState } from 'react';

// ... other imports

**Implementing the Allowance Check**

Now, let's modify the AirdropForm.tsx component to perform the allowance check within the handleSubmit function.

**1. Get Required Data with Wagmi Hooks:**

Inside the AirdropForm component, we use wagmi hooks to get essential dynamic data: the connected wallet's account address, the current chain ID, and the wagmi configuration object (needed for core actions).

function AirdropForm() {

const [tokenAddress, setTokenAddress] = useState('');

const [recipients, setRecipients] = useState('');

const [amounts, setAmounts] = useState('');

​

// Get dynamic data using wagmi hooks

const account = useAccount();

const chainId = useChainId();

const config = useConfig(); // Required for core actions like readContract

​

// ... rest of the component

}

**2. Define the**handleSubmit**Function:**

This asynchronous function will be triggered when the form is submitted. Our initial goal is to get the tSenderAddress for the current chain and then call a helper function to check the allowance.

async function handleSubmit() {

console.log("Form submitted");

console.log("Token Address:", tokenAddress);

console.log("Recipients:", recipients);

console.log("Amounts:", amounts);

​

// Get the tsender contract address for the current chain

const tSenderAddress = chainsToTsSender[chainId]?.tsender;

console.log("Current Chain ID:", chainId);

console.log("TSender Address for this chain:", tSenderAddress);

​

// Basic validation

if (!account.address) {

alert("Please connect your wallet.");

return;

}

if (!tSenderAddress) {

alert("TSender contract not found for the connected network. Please switch networks.");

return;

}

if (!tokenAddress || !/^0x[a-fA-F0-9]{40}$/.test(tokenAddress)) {

alert("Please enter a valid ERC20 token address (0x...).");

return;

}

// Add validation for recipients and amounts later...

​

// --- Step 1: Check Allowance ---

try {

// Using 'as `0x${string}`' for type assertion required by wagmi/viem

const approvedAmount = await getApprovedAmount(

tSenderAddress as `0x${string}`,

tokenAddress as `0x${string}`,

account.address

);

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

​

// TODO: Compare approvedAmount with the total amount needed for the airdrop

// TODO: If allowance is insufficient, call the approve function

// TODO: If allowance is sufficient, call the airdrop function on tsender contract

​

} catch (error) {

console.error("Error during submission process:", error);

alert("An error occurred. Check the console for details.");

}

}

**3. Create the**getApprovedAmount**Helper Function:**

This function encapsulates the logic for calling the allowance function on the specified ERC20 token contract using readContract from @wagmi/core.

// Helper function to check ERC20 allowance

async function getApprovedAmount(

spenderAddress: `0x${string}`,

erc20TokenAddress: `0x${string}`,

ownerAddress: `0x${string}`

): Promise<bigint> { // Return type should be bigint for uint256

​

console.log(`Checking allowance for token ${erc20TokenAddress}`);

console.log(`Owner: ${ownerAddress}`);

console.log(`Spender: ${spenderAddress}`);

​

try {

const allowance = await readContract(config, {

abi: erc20Abi,

address: erc20TokenAddress, // The address of the ERC20 token contract

functionName: 'allowance',

args: [ownerAddress, spenderAddress], // Arguments: owner, spender

});

​

console.log("Raw allowance response:", allowance);

// The response from 'allowance' is typically a BigInt

return allowance as bigint; // Assert type if necessary based on ABI return type

} catch (error) {

console.error("Error fetching allowance:", error);

// Rethrow or handle error appropriately

throw new Error("Failed to fetch token allowance.");

}

}

​

// ... inside the AirdropForm component ...

// Make sure to include the form JSX with an onSubmit handler pointing to handleSubmit

// e.g., <form onSubmit={(e) => { e.preventDefault(); handleSubmit(); }}> ... </form>

**Key points in**getApprovedAmount**:**

* It's async because readContract is asynchronous.
* It takes the config object (from useConfig), the erc20Abi, the token contract address, the functionName ('allowance'), and the required args (owner address, spender address).
* We use erc20Abi because allowance is a standard ERC20 function, even though we're interacting *for* our tsender contract.
* The address parameter is the address of the *token contract* you want to check the allowance for (which the user entered in the form).
* The args array requires the owner (the connected user's address) and the spender (the address of our tsender contract).
* Error handling is included with a try...catch block.
* The return type is Promise<bigint> as uint256 values are represented as BigInts in JavaScript/TypeScript.
* We use TypeScript's 0x${string} template literal type for addresses, as expected by viem (which wagmi uses internally). We use as for type assertion where needed if the initial state or input types don't perfectly match.

**Setting Up and Testing Locally with Anvil**

To test this interaction without spending real gas or tokens, use a local development blockchain like Anvil (part of the Foundry toolkit).

1. **Get Anvil State:** Obtain the tsender-deployed.json file (from the course repository or your deployment process). This file contains a snapshot of a blockchain state where your tsender contract and potentially some mock ERC20 tokens are already deployed. Place this file in your project's root directory.
2. **Add Anvil Script:** Add a script to your package.json to easily start Anvil with the saved state:

// package.json

"scripts": {

"anvil": "anvil --load-state tsender-deployed.json",

"dev": "next dev --turbo"

// ... other scripts

}

1. **Run Anvil:** Open a terminal and run pnpm anvil (or npm run anvil / yarn anvil). Anvil will start, load the state from the file, and print out local node details (RPC URL, private keys).
2. **Configure MetaMask:**
   * Add a new network in MetaMask.
   * Set the RPC URL to http://127.0.0.1:8545 (Anvil's default).
   * Set the Chain ID to 31337 (Anvil's default).
   * Import one of the accounts listed by Anvil using its private key. This account will be funded with test ETH and likely owns the mock tokens if included in the state file.
3. **Get Mock Token Address:** Find the address of the deployed mock ERC20 token (e.g., from the Anvil logs, the state file, or project documentation like a CONTRIBUTING.md). Let's assume it's 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512.
4. **Import Token (Optional but Recommended):** Add the mock token address as a custom token in MetaMask on the Anvil network so you can see its balance.

**Testing Steps:**

1. Ensure Anvil is running (pnpm anvil).
2. Run your frontend application (pnpm run dev).
3. Open the application in your browser.
4. Connect MetaMask, ensuring it's set to the Anvil network (Chain ID 31337) and using the imported Anvil account.
5. Enter the mock ERC20 token address (0xe7f...) into the "Token Address" field.
6. Enter some recipient addresses and amounts (e.g., your own address and 100).
7. Open your browser's developer console.
8. Click the "Send Tokens" button.
9. **Verify Console Output:** You should see logs indicating:
   * The current Chain ID (31337).
   * The tSenderAddress corresponding to chain 31337 from your constants.ts.
   * Messages from getApprovedAmount showing the owner, spender, and token address being checked.
   * The "Raw allowance response" (likely 0n or a similar BigInt representation of zero if you haven't approved tokens yet).
   * The final "Current allowance: 0" message (or the corresponding BigInt value).
   * Crucially, there should be no errors related to the readContract call itself.

You have now successfully implemented the logic to read the ERC20 token allowance from the blockchain using wagmi's core readContract action within a React component's event handler. The next steps involve comparing this allowance to the required amount and implementing the approve transaction if necessary.