**Getbalance**

**How to Fetch Smart Contract Balances Using viem**

This lesson demonstrates how to implement a feature that allows users to check the Ether (ETH) balance of a specific smart contract directly from a web frontend. We'll cover using the viem library for read-only blockchain interactions, handling balance formatting, and understanding the nuances of block confirmations in local development environments like Anvil.

**Understanding Read vs. Write Operations**

Before diving into the code, it's crucial to understand the difference between reading from and writing to the blockchain:

1. **Write Operations:** These actions change the state of the blockchain. Examples include sending Ether, transferring tokens, or calling smart contract functions that modify storage. Write operations require a transaction to be signed by a user's private key (usually via a wallet like MetaMask) and mined into a block. For these, you typically use a WalletClient.
2. **Read Operations:** These actions query the current state of the blockchain without changing it. Examples include checking an account's balance, reading a value from a smart contract's public variable, or fetching transaction details. Read operations don't require signing and can be performed by anyone using a connection to a blockchain node. For these, a PublicClient is sufficient.

Fetching a smart contract's balance is a read operation. Therefore, we'll use viem's PublicClient.

**Implementing the Get Balance Feature**

Let's add a button to our HTML and wire it up with JavaScript to fetch and display the contract's balance.

**1. HTML Button**

First, ensure you have a button in your index.html file that the user can click:

<button id="balanceButton">Get Balance</button>

**2. JavaScript Setup**

In your JavaScript file (e.g., index-js.js), get a reference to this button and attach an event listener:

// Assuming imports are handled, especially for viem and contractAddress

import { createPublicClient, custom, formatEther } from "https://esm.sh/viem";

import { contractAddress } from "./constants.js"; // Your contract address constant

​

// Get the button element

const balanceButton = document.getElementById("balanceButton");

​

// Attach the getBalance function to the button's click event

balanceButton.onclick = getBalance;

​

// The getBalance function definition will follow

**3. The**getBalance**Function**

Now, define the asynchronous getBalance function that will perform the actual work:

async function getBalance() {

// Check if a browser Ethereum provider (like MetaMask) is available

if (typeof window.ethereum !== "undefined") {

// Create a Public Client using viem

// This client is used for read-only interactions

const publicClient = createPublicClient({

// Connects viem to the browser's Ethereum provider (e.g., MetaMask)

transport: custom(window.ethereum)

});

​

try {

// Use the publicClient to fetch the balance of the specified address

const balance = await publicClient.getBalance({

address: contractAddress // The address of the smart contract

});

​

// The balance is returned in Wei as a BigInt

// Format it into Ether for user-friendly display

const formattedBalance = formatEther(balance);

​

// Log the formatted balance to the console

console.log(`Contract Balance: ${formattedBalance} ETH`);

// You could update a UI element here instead of logging

​

} catch (error) {

// Handle potential errors during the asynchronous call

console.error("Error getting balance:", error);

}

} else {

// Inform the user if MetaMask or another provider isn't installed

console.log("Please install MetaMask!");

// Update the UI to prompt installation if desired

}

}

**Key Code Explanations:**

* typeof window.ethereum !== "undefined": Checks if an Ethereum wallet provider (like MetaMask) is injected into the browser's window object.
* createPublicClient({...}): Instantiates a viem client configured for public (read-only) operations.
* transport: custom(window.ethereum): Tells viem how to communicate with the blockchain via the provider injected by the user's wallet.
* publicClient.getBalance({ address: contractAddress }): This is the core viem method call. It takes an object with the address whose balance you want to query and returns the balance as a JavaScript BigInt value representing Wei.
* contractAddress: This should be the checksummed address of the smart contract you deployed (imported from a constants file or defined directly). Ensure correct spelling.

**Handling Wei and Ether:**formatEther

Blockchain balances are typically handled in Wei, the smallest denomination of Ether (1 ETH = 1 x 1018 Wei). This allows for precise calculations without decimals. However, displaying a massive number like 1000000000000000000 Wei isn't user-friendly.

viem provides utility functions for conversion:

* formatEther(balanceInWei): Converts a BigInt representing Wei into a string representing Ether (e.g., "1.0"). We use this to display the balance.
* parseEther(amountInEther): Converts a string representing Ether (e.g., "1.0") into a BigInt representing Wei. This is useful when *sending* funds.

**The Challenge of Block Confirmations**

A common point of confusion arises when you perform a write operation (like funding the contract) and immediately try to read the updated state (like getting the balance).

1. You click "Fund" and confirm the transaction in MetaMask.
2. You immediately click "Get Balance".
3. You might see the *old* balance (e.g., "0" ETH) instead of the new balance.

**Why?** Blockchain state updates are not instantaneous. Your funding transaction must be included in a block that is then mined and added to the chain. The getBalance function reads the state from the *latest confirmed block*. If your transaction hasn't been mined yet, getBalance retrieves the state *before* your transaction was applied.

**Local Development Solutions with Anvil**

When using a local development blockchain like Anvil (part of the Foundry toolkit), you have control over block mining, which helps manage this:

1. **Default Behavior (Manual Mining on Transaction):** By default, Anvil usually mines a block only when it receives a new transaction.
   * *Workaround:* After funding, send another (tiny) transaction (e.g., send 0.0001 ETH to yourself). This forces Anvil to mine a block, confirming both the tiny transaction *and* your preceding funding transaction. Then, clicking "Get Balance" will show the updated state.
2. **Automatic Block Mining (**--block-time**):** A more convenient approach is to configure Anvil to mine blocks automatically at regular intervals.
   * Stop your current Anvil instance (Ctrl+C in the terminal).
   * Restart Anvil with the --block-time flag, specifying the interval in seconds:

anvil --block-time 5

# Or, if using state loading:

# anvil --load-state your-state-file.json --block-time 5

* + Now, after funding the contract, wait slightly longer than the block time (e.g., 5-6 seconds). Anvil will automatically mine a block, confirming the transaction. Clicking "Get Balance" will then reflect the updated balance without needing manual intervention.

**Anvil State Resets**

Remember that stopping and restarting Anvil typically resets the blockchain state to its default (e.g., balances go back to initial values). Even if you use flags like --load-state to reload deployment scripts and ensure your contract exists at the same address, it usually doesn't persist arbitrary state changes (like balances updated via UI interactions) from previous runs unless more advanced state management is configured. This reset behavior is often beneficial for ensuring clean test environments.

By understanding the difference between read/write operations, using publicClient and formatEther correctly, and managing block confirmations during development, you can reliably implement features like getBalance in your decentralized applications.