**Sending Contract Transactions Setup**

**Recap and Goal: Setting Up the Fund Function**

In our previous step, we successfully connected a user's wallet (like MetaMask) to our minimal HTML/JS application using Viem's createWalletClient, the custom transport, and requestAddresses. Now, we'll build upon that foundation.

Consulting our project plan (our README.md), the next logical step is to implement the "Buy Coffee" functionality. This involves:

1. Adding the necessary HTML elements (a button and likely an input field).
2. Writing JavaScript logic to interact with a specific function on a deployed smart contract.
3. Sending some Ether (ETH) along with that function call to simulate funding the contract.

This lesson focuses on setting up the frontend structure and the initial JavaScript logic required to prepare for sending this transaction.

**Understanding the Target: The FundMe Smart Contract**

Our frontend application needs to interact with a backend smart contract. For this example, we'll be interacting with the FundMe contract, the code for which resides in a separate repository (cyfrin/foundry-fund-me-cu). You can view the source code (FundMe.sol) there.

The specific function we need to call for our "Buy Coffee" button is the fund() function within the FundMe contract:

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

// ... imports ...

contract FundMe {

// ... state variables, modifiers, constructor ...

​

/\*\*

\* @notice Funds our contract based on the ETH/USD price

\*/

function fund() public payable { // <-- Our target function

require(msg.value.getConversionRate(s\_priceFeed) >= MINIMUM\_USD, "You need to spend more ETH!");

s\_addressToAmountFunded[msg.sender] += msg.value;

s\_funders.push(msg.sender);

}

​

// ... other functions like withdraw ...

}

Key points about this fund function:

* It's public, meaning it can be called from outside the contract (e.g., from our frontend application).
* It's payable, meaning it's designed to receive Ether (ETH) when it's called.

Calling this function essentially sends ETH from the user's wallet to the smart contract's address, effectively funding it.

Because smart contracts can hold balances (just like regular blockchain accounts), we'll also add a "Get Balance" button to check how much ETH our FundMe contract holds after being funded.

**Frontend Perspective: What You Need to Interact**

As a frontend developer, you don't necessarily need deep expertise in Solidity (the language smart contracts are often written in) to interact with a contract. Your primary focus is on understanding:

1. **Which functions** are available to call.
2. **What parameters** (inputs) each function requires.
3. **What the function does** at a high level (e.g., "this function sends funds").
4. **The contract's address** on the blockchain.
5. **The contract's ABI** (Application Binary Interface - a JSON file describing the contract's functions and events).

You can typically get this information by collaborating with the smart contract development team, reading documentation, using analysis tools, or inspecting the contract code if you're comfortable doing so.

**Building the UI: HTML for Funding and Balance Checks**

Let's add the necessary elements to our index.html file.

First, add a button to check the contract's balance:

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

Next, create a simple form structure to handle the funding ("Buy Coffee") action. This will include a label, an input field for the user to specify the ETH amount, and the button itself.

<!-- Form - "buy coffee" or "fund" -->

<label for="ethAmount">ETH Amount</label>

<input id="ethAmount" placeholder="0.1" />

<button type="button" id="fundButton">Buy Coffee</button>

Breaking down the funding elements:

* The <label> improves accessibility and user experience by associating text with the input field.
* The <input> with id="ethAmount" allows users to enter the amount of ETH they wish to send. The placeholder provides a visual hint.
* The <button> with id="fundButton" will trigger our JavaScript fund function. We use type="button" to prevent default browser form submission behavior, ensuring our JavaScript handles the click.

After adding these elements, your frontend should display the new "Get Balance" button, the "ETH Amount" label and input field, and the "Buy Coffee" button below the existing "Connect" button. For now, these new buttons won't do anything when clicked.

**JavaScript Setup: Connecting Buttons to Functions**

Now, let's connect these new HTML elements to our JavaScript code in index-js.js.

First, get references to the new elements using their IDs, typically near the top where you select other elements:

// Add these lines with your other element selections

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

const ethAmountInput = document.getElementById("ethAmount");

// You'll also need the balanceButton eventually

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

Next, define an empty async function that will eventually hold the logic for funding the contract:

async function fund() {

// We will add logic here soon

}

Finally, attach this fund function to the onclick event of the fundButton. Add this line near where you assigned the connect function:

// Event Listeners

connectButton.onclick = connect;

fundButton.onclick = fund; // Add this line

// balanceButton.onclick = getBalance; // Will add later

**Initial Fund Function: Reading User Input**

Let's add some basic logic inside the fund function. We need to get the value the user entered into the input field and log it to the console to confirm it's working.

Modify the fund function like this:

async function fund() {

const ethAmount = ethAmountInput.value; // Get value from the input field

console.log(`Funding with ${ethAmount}...`); // Log the value using template literals

​

// More logic will follow...

}

Here, ethAmountInput.value retrieves the text entered by the user. We use template literals (backticks `) for the console.log statement, which allows us to easily embed the ethAmount variable directly within the string using ${ethAmount}.

Test this by entering a value (e.g., "0.1" or "1") into the "ETH Amount" input field on your webpage and clicking the "Buy Coffee" button. You should see the message "Funding with 0.1..." (or whatever value you entered) printed in your browser's developer console.

**Ensuring Wallet Readiness within the Fund Function**

A crucial consideration is that the user might click "Buy Coffee" *before* clicking "Connect", or their connection state might somehow be lost. Our fund function needs access to the walletClient (which we initialized in the connect function) to interact with the user's wallet.

To ensure the walletClient is ready, we need to perform the same connection check and initialization steps within the fund function *before* attempting any transaction logic.

Copy the relevant code block from your connect function into the fund function:

async function fund() {

const ethAmount = ethAmountInput.value;

console.log(`Funding with ${ethAmount}...`);

​

// Ensure wallet is connected and client is initialized

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

// Re-initialize or confirm walletClient

// Note: We assume 'walletClient' is declared globally (e.g., 'let walletClient;')

walletClient = createWalletClient({

transport: custom(window.ethereum),

});

// Request account access (important step!)

const [address] = await walletClient.requestAddresses();

console.log("Wallet connected, Account:", address);

​

// Now we can proceed with transaction logic...

​

} else {

// Handle the case where MetaMask (or other provider) is not installed

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

// Consider disabling the button or updating its text here

// e.g., fundButton.innerHTML = "Please Install MetaMask";

}

}

This block checks for window.ethereum, re-initializes the walletClient (assigning it to our globally declared variable), and crucially calls requestAddresses(). This call not only retrieves the user's connected address but also prompts the user to connect via MetaMask if they aren't already connected to our site. This ensures we have an active wallet connection and the necessary client object before proceeding. We also capture the returned address, as we'll need it later.

*(Note: We removed the lines that updated the connectButton.innerHTML as that logic belongs specifically to the connect function).*

**Best Practice: Simulating Transactions with Viem**

Before actually sending a transaction to the blockchain (which requires the user's confirmation via MetaMask and costs gas fees), it's a strong best practice to *simulate* the transaction first.

**Why Simulate?**  
Simulation allows Viem to check if the transaction is likely to succeed *without* actually broadcasting it to the network or costing any gas. It runs the transaction locally against the current blockchain state. This provides several benefits:

* **Early Error Detection:** Catches potential issues (like insufficient funds, incorrect parameters, contract logic errors) before the user is prompted to sign.
* **Better User Experience:** Prevents users from paying gas for transactions that are destined to fail.
* **Gas Estimation:** Simulation often provides gas estimates.

**Viem's**simulateContract**:**  
Viem provides the simulateContract function specifically for this purpose. Looking at the Viem documentation, we see that this function requires a Public Client.

**Public Client vs. Wallet Client:**

* Wallet Client**:** Used for actions requiring a private key signature (sending transactions, signing messages). It interacts directly with the user's wallet (e.g., MetaMask). We created this using createWalletClient.
* Public Client**:** Used for read-only interactions with the blockchain (fetching data, reading contract state) and for simulating transactions. It doesn't need private keys and can interact directly with a public blockchain node (via an RPC URL) or through the wallet provider.

**Creating a**Public Client**:**  
Let's set up a publicClient in our index-js.js.

1. **Import:** Add createPublicClient to your Viem import statement:

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

1. **Declare Globally:** Declare a global variable for it, similar to walletClient:

let walletClient;

let publicClient; // New global variable

1. **Initialize:** Inside the fund function, after successfully initializing the walletClient and getting the address, create the publicClient. In this setup, we can conveniently use the same custom(window.ethereum) transport provided by MetaMask:

async function fund() {

const ethAmount = ethAmountInput.value;

console.log(`Funding with ${ethAmount}...`);

​

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

walletClient = createWalletClient({

transport: custom(window.ethereum),

});

const [address] = await walletClient.requestAddresses();

console.log("Wallet connected, Account:", address);

​

// Create Public Client after Wallet Client is ready

publicClient = createPublicClient({

transport: custom(window.ethereum)

});

console.log("Public Client Initialized");

​

// Now we can use publicClient for simulation...

​

} else {

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

}

}

**Preparing for Simulation: Required Parameters**

Now that we have our publicClient, we can prepare to call simulateContract. Based on the Viem documentation, the simulateContract function expects an object with several key parameters:

* address: The blockchain address of the smart contract we want to interact with (the FundMe contract address).
* abi: The Application Binary Interface (ABI) of the smart contract. This is a JSON array that describes the contract's functions, events, and parameters, telling Viem how to encode the function call.
* functionName: A string specifying the exact name of the function to call (in our case, "fund").
* account: The address of the user initiating the transaction (the address we obtained from requestAddresses).
* value: The amount of ETH to send with the transaction. This needs to be provided as a BigInt value representing Wei (the smallest unit of Ether). We'll need to parse the user's input (ethAmount) into this format.
* args (Optional): An array of arguments for the function, if it requires any. Our fund() function doesn't require arguments.

Our next step will be to obtain the FundMe contract's address and abi, correctly parse the ethAmount into Wei, and then call publicClient.simulateContract within a try...catch block to handle potential simulation errors.

// Inside the fund function's 'if' block, after initializing publicClient:

try {

// We need to define contractAddress and contractAbi first!

// We also need to parse ethAmount into Wei (e.g., using viem's parseEther)

​

console.log("Attempting simulation...");

const simulationResult = await publicClient.simulateContract({

address: undefined, // TODO: Add deployed contract address

abi: undefined, // TODO: Add contract ABI

functionName: 'fund',

account: address, // Use the address obtained from requestAddresses

value: undefined, // TODO: Add parsed ETH amount in Wei

});

​

console.log("Simulation successful:", simulationResult);

// If simulation succeeds, simulationResult.request contains the prepared transaction details

// We can then pass this to walletClient.writeContract() to send the actual transaction

​

} catch (error) {

console.error("Simulation failed:", error);

// Handle simulation errors appropriately (e.g., display message to user)

}

We have now successfully set up the HTML structure, the basic JavaScript function, ensured wallet connectivity, and prepared the groundwork for simulating the fund transaction using Viem's publicClient. The next phase involves providing the contract specifics (address, ABI) and actually performing the simulation and subsequent transaction sending.