**Sending The Tx**

**Sending a State-Changing Transaction with viem**

This lesson guides you through the process of sending a state-changing transaction to a smart contract from a web frontend. We'll use viem to interact with a local Anvil blockchain node via MetaMask. Specifically, we'll focus on funding a smart contract, implementing the crucial "simulate-then-write" pattern for a safer and more robust user experience.

**Prerequisites: Anvil and MetaMask Setup**

Before initiating transactions, ensure your local development environment is correctly set up:

1. **Running Anvil Node:** A local blockchain node, like Anvil, must be running. If you have a saved state (e.g., containing deployed contracts), start Anvil using a command similar to anvil --load-state fundme-anvil.json in your terminal. This ensures your contract is available and the blockchain retains any previously saved state. Remember, keeping this node running is essential throughout the development process unless you explicitly save its state.
2. **MetaMask Connection:** Your browser's MetaMask wallet needs to be configured and connected:
   * **Network:** Select the custom RPC network pointing to your local Anvil node (e.g., http://127.0.0.1:8545). Ensure your web application has permission to interact with this network.
   * **Account:** Select the specific account you intend to use for transactions. This account should typically be one provided by Anvil (often imported into MetaMask) and hold sufficient test ETH. Grant your web application permission to access this specific account. Properly configuring these permissions prevents ambiguity and ensures transactions originate from the correct source on the intended network.

**Step 1: Simulating the Transaction Before Sending**

Sending a transaction directly can lead to failures if conditions aren't met (e.g., insufficient funds, incorrect arguments, contract logic errors). These failed transactions still consume gas and provide a poor user experience. Best practice dictates simulating the transaction first.

Simulation executes the transaction logic locally on the node *without* broadcasting it to the network or creating a permanent state change. It's a dry run to catch potential errors beforehand. viem provides the simulateContract function for this purpose, typically used with a publicClient.

**Understanding**publicClient**vs.**walletClient**:**

* publicClient**:** Used for read-only operations and simulations that don't require transaction signing (interacting directly with the blockchain node's public RPC).
* walletClient**:** Used for operations requiring user interaction and signing via their private key (sending transactions, signing messages). It interacts with the user's wallet (e.g., MetaMask).

**Simulating the**fund**Function:**

To simulate calling a fund function on our contract, which accepts ETH, we use publicClient.simulateContract:

// Assuming publicClient, contractAddress, coffeeAbi, connectedAccount,

// currentChain, and ethAmount (as a string) are already defined.

// parseEther is imported from viem.

​

try {

const { request } = await publicClient.simulateContract({

address: contractAddress, // Target contract address

abi: coffeeAbi, // Contract's ABI

functionName: "fund", // Function to simulate

account: connectedAccount, // Account initiating the call

chain: currentChain, // Chain information object

value: parseEther(ethAmount), // Amount of ETH (in Wei) to send

});

​

console.log("Simulation successful. Prepared request:", request);

// Proceed to sending the transaction using this 'request' object...

} catch (error) {

console.error("Transaction simulation failed:", error);

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

}

**Explanation:**

* We provide the contract details (address, abi, functionName), the transaction parameters (account, chain, value), converting the ETH amount string to Wei using parseEther.
* If the simulation is successful (i.e., the node determines the call would likely succeed if actually sent), simulateContract returns an object.
* We destructure the request property from this result. This request object contains all the necessary parameters, validated and formatted by viem and the node, ready to be passed to writeContract for actual execution.
* If the simulation fails (e.g., due to a require statement in the contract), an error is thrown, which should be caught and handled appropriately.

**Step 2: Executing the Transaction via MetaMask**

Once the simulation succeeds and we have the validated request object, we can confidently ask the user to sign and send the actual transaction. This is done using the walletClient and its writeContract function.

**Sending the Transaction:**

The request object obtained from simulateContract is passed directly to walletClient.writeContract:

// Assuming walletClient is defined and 'request' was successfully

// obtained from the simulation step above.

​

try {

const hash = await walletClient.writeContract(request);

console.log("Transaction sent! Hash:", hash);

// Transaction submitted, now wait for confirmation (covered in later lessons)

} catch (error) {

console.error("Failed to send transaction:", error);

// Handle errors during sending (e.g., user rejected in MetaMask)

}

**User Interaction and Confirmation:**

1. Calling walletClient.writeContract(request) triggers MetaMask (or the connected wallet) to display a confirmation pop-up.
2. This pop-up shows the user the details of the transaction they are about to sign (recipient address, function being called, ETH value, estimated gas fees).
3. The user must explicitly click 'Confirm' to sign the transaction with their private key and authorize it to be broadcast to the network. If they click 'Reject', the writeContract call will throw an error.
4. Upon confirmation, writeContract resolves and returns the transaction hash. This hash is the unique identifier for this transaction on the blockchain.
5. Simultaneously, you should see output in your Anvil node's terminal confirming it received and mined the transaction, also displaying the hash, gas used, and block number.

This simulate-then-write pattern ensures that the transaction the user approves in MetaMask is the exact same one that was successfully simulated moments before, significantly reducing the chances of unexpected failures on-chain.

**Important: Maintaining Your Local Blockchain State**

When working with a local development node like Anvil, remember that its state (account balances, contract storage) is typically ephemeral.

* **Stopping Anvil:** If you stop the Anvil process (e.g., by closing the terminal or killing the process), any state changes made during that session (like the ETH funded to the contract in this example) will be lost unless you explicitly configured state saving.
* **State Persistence:** Using flags like --dump-state on exit and --load-state on startup allows you to preserve the blockchain state between sessions. Ensure you are using these if you need persistence.
* **Restarting Development:** If you stop and restart Anvil *without* loading a saved state, you'll need to re-run any setup transactions (like funding the contract) to bring your local blockchain back to the expected state before proceeding with dependent functionality. Always ensure your contract is in the correct state on your *current* Anvil instance after restarting your environment.