**Simulations**

**Simulating Contract Interactions with**viem

Before sending a transaction that modifies the blockchain state and costs gas, it's crucial to verify if it's likely to succeed. This process, known as transaction simulation, allows you to predict the outcome without incurring costs or making permanent changes. This lesson demonstrates how to use the viem JavaScript library's simulateContract function to simulate a fund transaction for a "Buy Me a Coffee" style decentralized application. Simulations are typically performed using a publicClient, which handles read-only operations and doesn't require transaction signing.

**Organizing Contract Details: Address and ABI**

To interact with a specific smart contract, your frontend needs two key pieces of information: the contract's deployment address and its Application Binary Interface (ABI). Storing these in a dedicated constants file enhances code organization and reusability.

First, create a constants file (e.g., constants-js.js) and export the necessary values:

// constants-js.js

​

// Replace "0x..." with your actual deployed contract address

export const contractAddress = "0x...";

​

// The ABI defines the contract's functions and structures

export const coffeeAbi = [

// ... (Paste your contract's full ABI array here)

// Example structure:

// {

// "inputs": [],

// "name": "fund",

// "outputs": [],

// "stateMutability": "payable",

// "type": "function"

// },

// ... other functions and events

];

Next, import these constants into your main JavaScript file (e.g., index-js.js) where you'll be calling the simulation function:

// index-js.js

​

// Import the contract address and ABI

import { contractAddress, coffeeAbi } from "./constants-js.js";

// Note: Renaming 'abi' to 'coffeeAbi' during import or in the constants file improves clarity.

These imported values will be used as parameters when calling simulateContract. Inside your interaction function (e.g., async function fund()):

// Inside async function fund()

​

// ... other code ...

​

await publicClient.simulateContract({

address: contractAddress, // Specifies which contract to simulate on

abi: coffeeAbi, // Provides the contract interface definition

// ... other parameters ...

});

​

// Note: If the variable name matches the key (e.g., if you imported `abi`),

// JavaScript allows the shorthand `abi,` instead of `abi: abi`.

// Using the explicit form `abi: coffeeAbi` can be clearer.

**Identifying the Transaction Sender: Fetching the Account**

Even though a simulation doesn't execute on-chain, it mimics an actual transaction. Therefore, it needs to know *which account* is pretending to send the transaction. This context is crucial because contract logic might depend on the sender's address (msg.sender).

We use the walletClient (which handles interactions requiring a connected wallet) to retrieve the user's currently connected account address. The requestAddresses() method returns an array of addresses; typically, we only need the first one. JavaScript array destructuring provides a concise way to extract this first element.

// Inside async function fund()

​

// Ensure walletClient is initialized (e.g., during connection)

// Example: walletClient = createWalletClient({ transport: custom(window.ethereum) });

​

// Fetch the connected accounts

const accounts = await walletClient.requestAddresses();

// Use array destructuring to get the first account

const [connectedAccount] = accounts;

​

// ... later in the simulateContract call ...

​

await publicClient.simulateContract({

// ... other parameters ...

account: connectedAccount, // The account context for the simulation

functionName: "fund", // Specify the contract function to simulate

// ... other parameters ...

});

Remember the distinction: walletClient is used for actions requiring wallet connection/signing (like getting addresses or later sending transactions), while publicClient is used for read-only operations and simulations that query blockchain state or predict outcomes.

**Specifying the Network: Defining the Chain**

Simulations need to occur within the context of a specific blockchain network (e.g., Ethereum Mainnet, Sepolia testnet, or a local development network like Anvil or Hardhat). While viem has built-in definitions for common public networks, custom or local networks require explicit definition using the defineChain utility.

To manage this, we can use a helper function, potentially sourced from project examples (like the html-ts-coffee-cu GitHub repository mentioned in the video), to dynamically define the chain based on the connected wallet's current network.

First, ensure defineChain is imported from viem:

// index-js.js (top-level imports)

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

// Note: The video initially missed this import, causing a ReferenceError.

Then, include the helper function in your code:

// index-js.js (helper function, potentially near the bottom)

​

async function getCurrentChain(client) {

// Get the chain ID from the connected wallet client

const chainId = await client.getChainId();

​

// Define the chain parameters using viem's defineChain

const currentChain = defineChain({

id: chainId,

name: "Local Devnet", // Provide a descriptive name (e.g., Anvil, Hardhat)

nativeCurrency: {

name: "Ether",

symbol: "ETH",

decimals: 18,

},

rpcUrls: {

// Use the RPC URL of your local node

default: { http: ["http://localhost:8545"] },

// public: { http: ["http://localhost:8545"] }, // Optional: specify public RPC if different

},

// Add other chain-specific details if needed (e.g., blockExplorers)

});

return currentChain;

}

Now, call this helper function within your fund function and pass the resulting chain object to simulateContract:

// Inside async function fund()

​

// Get the defined chain object using the walletClient

const currentChain = await getCurrentChain(walletClient);

​

// ... later in the simulateContract call ...

​

await publicClient.simulateContract({

// ... other parameters ...

chain: currentChain, // Pass the defined chain object for network context

// ... other parameters ...

});

Defining the chain explicitly ensures viem knows how to interact with your target network, especially crucial for local development environments.

**Handling Transaction Value: Converting Ether to Wei**

Many smart contract functions, particularly those involving payments like our fund function, expect cryptocurrency to be sent along with the call (msg.value). User interfaces typically display amounts in Ether (e.g., "0.1 ETH"), but smart contracts operate using the smallest unit, Wei (1 Ether = 1018 Wei). Therefore, conversion is mandatory.

viem provides the parseEther utility function to easily convert a string representation of Ether into its equivalent Wei value as a BigInt, which is the required format for contract interactions.

First, ensure parseEther is imported from viem:

// index-js.js (top-level imports)

import { ..., parseEther, defineChain, ... } from "https://esm.sh/viem";

Inside your fund function, retrieve the user input (assumed to be in Ether) and use parseEther to convert it before passing it as the value parameter to simulateContract.

// Inside async function fund()

​

// Assume ethAmountInput is your HTML input element for the ETH amount

const ethAmount = ethAmountInput.value; // Get the amount string (e.g., "0.01")

​

// You can verify the conversion (optional):

// console.log(`Converting ${ethAmount} ETH to Wei:`, parseEther(ethAmount));

// Inputting "1" would log: 1000000000000000000n

​

// ... later in the simulateContract call ...

​

await publicClient.simulateContract({

// ... other parameters ...

value: parseEther(ethAmount), // Convert the Ether string to Wei BigInt

});

Using parseEther correctly handles the large numbers involved in Wei calculations and ensures the value parameter is formatted correctly for the simulation.

**Simulating the Transaction: Putting It All Together**

Having prepared all the necessary parameters – contract address, ABI, function name, simulating account, network chain definition, and the transaction value in Wei – we can now assemble the complete simulateContract call. It's best practice to wrap asynchronous calls like this in a try...catch block to handle potential simulation errors gracefully.

Here is the structure of the fund function incorporating all the steps:

// Assumes publicClient and walletClient are already created and available

// Assumes ethAmountInput and connectButton are references to HTML elements

// Assumes constants (contractAddress, coffeeAbi) are imported

// Assumes helper function getCurrentChain is defined

​

async function fund() {

const ethAmount = ethAmountInput.value; // Get ETH amount from input

console.log(`Attempting to simulate funding with ${ethAmount} ETH...`);

​

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

try {

// 1. Get the connected account

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

​

// 2. Define the current chain

const currentChain = await getCurrentChain(walletClient);

​

// 3. Call simulateContract with all parameters

publicClient = createPublicClient({

transport: custom(window.ethereum),

});

console.log("Preparing simulation parameters...");

await publicClient.simulateContract({

// Contract details:

address: contractAddress, // From constants

abi: coffeeAbi, // From constants

functionName: "fund", // Target function on the contract

​

// Transaction context:

account: connectedAccount, // Account performing the simulated transaction

chain: currentChain, // Network definition for the simulation

​

// Transaction parameters:

value: parseEther(ethAmount), // ETH amount converted to Wei

// args: [], // Include if your function takes arguments

});

​

// If the above line doesn't throw an error, the simulation is successful

console.log("Simulation successful! Transaction is likely to succeed.");

​

// Next logical step: Allow the user to actually send the transaction

// e.g., await walletClient.writeContract({ address, abi, functionName, account, chain, value });

​

} catch (error) {

// Log simulation errors for debugging

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

// Provide user feedback (e.g., update UI)

alert(`Simulation failed: ${error.message || error}`);

}

} else {

// Handle case where MetaMask or other provider isn't installed

connectButton.innerHTML = "Please install a web3 provider!";

console.log("No web3 provider detected.");

}

}

By successfully executing simulateContract, you gain confidence that the actual transaction, when sent using walletClient.writeContract with the same parameters, is likely to succeed on the blockchain. This pre-flight check is an essential part of building robust and user-friendly dApps.