**Review**

**Reviewing Your First dApp Frontend: Connecting, Reading, and Writing with Viem**

Congratulations! You've successfully built a minimal decentralized application (dApp) frontend using HTML and JavaScript, allowing users to interact with a Solidity smart contract deployed on a local Anvil testnet. This lesson recaps the key functionalities you implemented, the technologies involved, and the core Viem concepts used to bridge the gap between your web interface and the blockchain.

**Technology Stack Recap**

To build this initial project, we utilized a specific set of tools and technologies:

* **Frontend:** HTML and Vanilla JavaScript (with a brief look at TypeScript)
* **Blockchain Interaction:** Viem library
* **Wallet:** MetaMask browser extension
* **Local Blockchain:** Anvil (from the Foundry toolset), running with a pre-loaded state (fundme-anvil.json)
* **Development Environment:** VS Code with the Live Server extension and the integrated terminal
* **AI Assistance (Optional):** Tools like DeepSeek, Claude, Cursor, or GitHub Copilot can aid development.

**Core Functionality Walkthrough**

Our minimal frontend successfully implemented the essential interactions needed for many dApps:

**Connecting Your Wallet**

The first step for any dApp user is connecting their wallet.

* **Goal:** Link the user's MetaMask wallet to the web application.
* **Action:** Clicking the "Connect" button.
* **Mechanism:** We check for the presence of window.ethereum (injected by MetaMask). If found, we use Viem's createWalletClient function, configured with the window.ethereum provider, and call walletClient.requestAddresses() to prompt the user for connection permission via MetaMask. On success, the UI is updated.

**Reading Contract Data: Getting the Balance**

Once connected, users often need to view information stored on the blockchain or within a smart contract.

* **Goal:** Display the current ETH balance held by our deployed smart contract.
* **Action:** Clicking the "Get Balance" button.
* **Mechanism:** This interaction reads public data and doesn't require a signed transaction. We use Viem's createPublicClient, again configured with the window.ethereum transport. The publicClient.getBalance() function is called, passing the contractAddress whose balance we want to fetch. The returned value (in Wei) is then formatted into Ether using formatEther for readability before being logged. This demonstrates a read-only operation.

**Writing to the Contract: Funding (Sending ETH)**

Sending funds or changing the state of a smart contract requires writing to the blockchain.

* **Goal:** Allow the user to send a specified amount of ETH to the smart contract's fund function.
* **Action:** Entering an ETH amount and clicking the "Buy Coffee" (Fund) button.
* **Mechanism:** This is a write operation involving value transfer.
  1. **Clients:** We need both a WalletClient (for signing the transaction) and a PublicClient (for simulation).
  2. **Value Parsing:** The user's input amount is converted from Ether to Wei using Viem's parseEther.
  3. **Simulation:** Before sending, we use publicClient.simulateContract(). This crucial step asks the node to simulate the transaction (fund function call with the specified value). It checks for potential errors (like reverts) and estimates gas *without* requiring the user to sign or spend gas yet. It returns a prepared request object if successful.
  4. **Execution:** If the simulation succeeds, we pass the request object to walletClient.writeContract(). This prompts the user in MetaMask to review and sign the actual transaction (including the amount, gas fees, etc.). Once confirmed, the transaction is sent to the network, and its hash is returned.

**Writing to the Contract: Withdrawing Funds**

Withdrawing involves calling a contract function that changes state (the contract's balance) and potentially transfers funds *from* the contract.

* **Goal:** Trigger the smart contract's withdraw function to send its balance back to the owner (the connected user).
* **Action:** Clicking the "Withdraw" button.
* **Mechanism:** This follows the same simulate-then-write pattern as funding:
  1. **Clients:** WalletClient and PublicClient are used.
  2. **Simulation:** publicClient.simulateContract() is called, targeting the withdraw function name in the contract's ABI. No value field is typically needed here, as we aren't sending ETH *to* the contract with this call.
  3. **Execution:** walletClient.writeContract() is called with the request from the successful simulation, prompting MetaMask confirmation for the state-changing transaction (which still costs gas).

**Core Viem Concepts Reviewed**

Understanding these Viem concepts is fundamental for frontend blockchain development:

* WalletClient**:** Used for operations requiring user authentication and signing, such as sending transactions (writeContract) or signing messages. It needs access to a wallet provider like window.ethereum. Created with createWalletClient.
* PublicClient**:** Used for reading public blockchain data (getBalance, readContract) or simulating transactions (simulateContract). It doesn't necessarily require a wallet connection for read operations. Created with createPublicClient.
* **Reading vs. Writing:** Reading data is generally free (no gas cost) and doesn't require a transaction. Writing data (changing state or transferring value) always requires a signed transaction, which consumes gas.
* simulateContract**:** A vital best practice before writing. It preemptively checks if a transaction is likely to succeed, estimates gas, and prepares the transaction request, improving user experience and preventing wasted gas on failed transactions.
* writeContract**:** The function used with a WalletClient to submit a signed transaction to the blockchain network, using the validated request data obtained from simulateContract.
* **ABI (Application Binary Interface):** The JSON representation of your smart contract's interface. Viem uses the ABI (like our imported coffeeAbi) to correctly encode function calls and decode return data when interacting with the contractAddress.
* **Contract Address:** The unique identifier on the blockchain where your smart contract code resides. Essential for targeting interactions.

**Key Code Structures (Simplified Examples)**

Let's revisit the basic structure of our interaction functions using Viem:

**Connecting the Wallet**

// Assuming createWalletClient, custom, defineChain are imported from Viem

// And window.ethereum is available

​

async function connect() {

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

try {

const walletClient = createWalletClient({

chain: anvil, // Replace with your chain definition

transport: custom(window.ethereum)

});

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

console.log("Connected account:", address);

// Update UI (e.g., button text)

} catch (error) {

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

}

} else {

console.log("MetaMask not detected.");

// Update UI to prompt installation

}

}

* **Highlights:** Checks for MetaMask, creates WalletClient, requests accounts via requestAddresses().

**Reading the Contract Balance**

// Assuming createPublicClient, custom, defineChain, formatEther are imported

// And contractAddress is defined

​

async function getBalance() {

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

try {

const publicClient = createPublicClient({

chain: anvil, // Replace with your chain definition

transport: custom(window.ethereum)

});

const balanceWei = await publicClient.getBalance({

address: contractAddress,

});

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

} catch (error) {

console.error("Failed to get balance:", error);

}

} // Handle no MetaMask case

}

* **Highlights:** Creates PublicClient, uses getBalance() with the contract address, formats the result.

**Funding the Contract (Simulate & Write)**

// Assuming createWalletClient, createPublicClient, custom, defineChain,

// parseEther, contractAddress, coffeeAbi are available/imported

​

async function fund(ethAmountString) {

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

const walletClient = createWalletClient({ /\* ... config ... \*/ });

const publicClient = createPublicClient({ /\* ... config ... \*/ });

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

​

try {

console.log(`Simulating fund with ${ethAmountString} ETH...`);

// 1. Simulate

const { request } = await publicClient.simulateContract({

account,

address: contractAddress,

abi: coffeeAbi,

functionName: 'fund',

value: parseEther(ethAmountString),

chain: anvil, // Replace with your chain definition

});

​

console.log("Simulation successful. Sending transaction...");

// 2. Write

const txHash = await walletClient.writeContract(request);

console.log("Funding transaction sent:", txHash);

// Optional: Wait for transaction confirmation

// await publicClient.waitForTransactionReceipt({ hash: txHash });

// console.log("Funding transaction confirmed.");

​

} catch (error) {

console.error("Funding failed:", error); // Catches simulation or execution errors

}

} // Handle no MetaMask case

}

* **Highlights:** Creates both clients, gets account, uses parseEther, calls simulateContract with function details and value, then calls writeContract with the simulation result (request).

**A Note on JavaScript vs. TypeScript**

While we focused on Vanilla JavaScript for simplicity in this review, you also saw a brief conversion to TypeScript (index-ts.ts). TypeScript adds static typing, which can help catch errors during development. However, browsers run JavaScript, so TypeScript code must be compiled (transpiled) into JavaScript before it can be used in an HTML file. Modern frontend frameworks and build tools (which we'll explore later) often handle this compilation process automatically.

**Using AI Assistants in Development**

Tools like DeepSeek (shown in the video), Claude, Cursor, and GitHub Copilot can be powerful allies. Use them to:

* Generate boilerplate code.
* Explain complex concepts or library functions (like Viem's).
* Help debug errors.
* Learn new technologies, including version control with Git.

**Build Your Portfolio: The Importance of GitHub**

This is crucial advice: **Start pushing your code to GitHub now.** Even small projects like this demonstrate your ability to write functional code and interact with web3 technologies. A populated GitHub profile is one of the most effective ways to showcase your skills to potential employers or collaborators in the web3 space. If you're unfamiliar with Git, use AI assistants or online resources to learn the basic commands (git init, git add, git commit, git push). Don't wait until your projects are "perfect" – start building your public coding history today.

**Next Steps**

You've successfully reviewed the fundamentals of creating a dApp frontend that interacts with a smart contract. Take a well-deserved break! The next sections will introduce more advanced concepts and tooling, building upon the foundation you've established here.