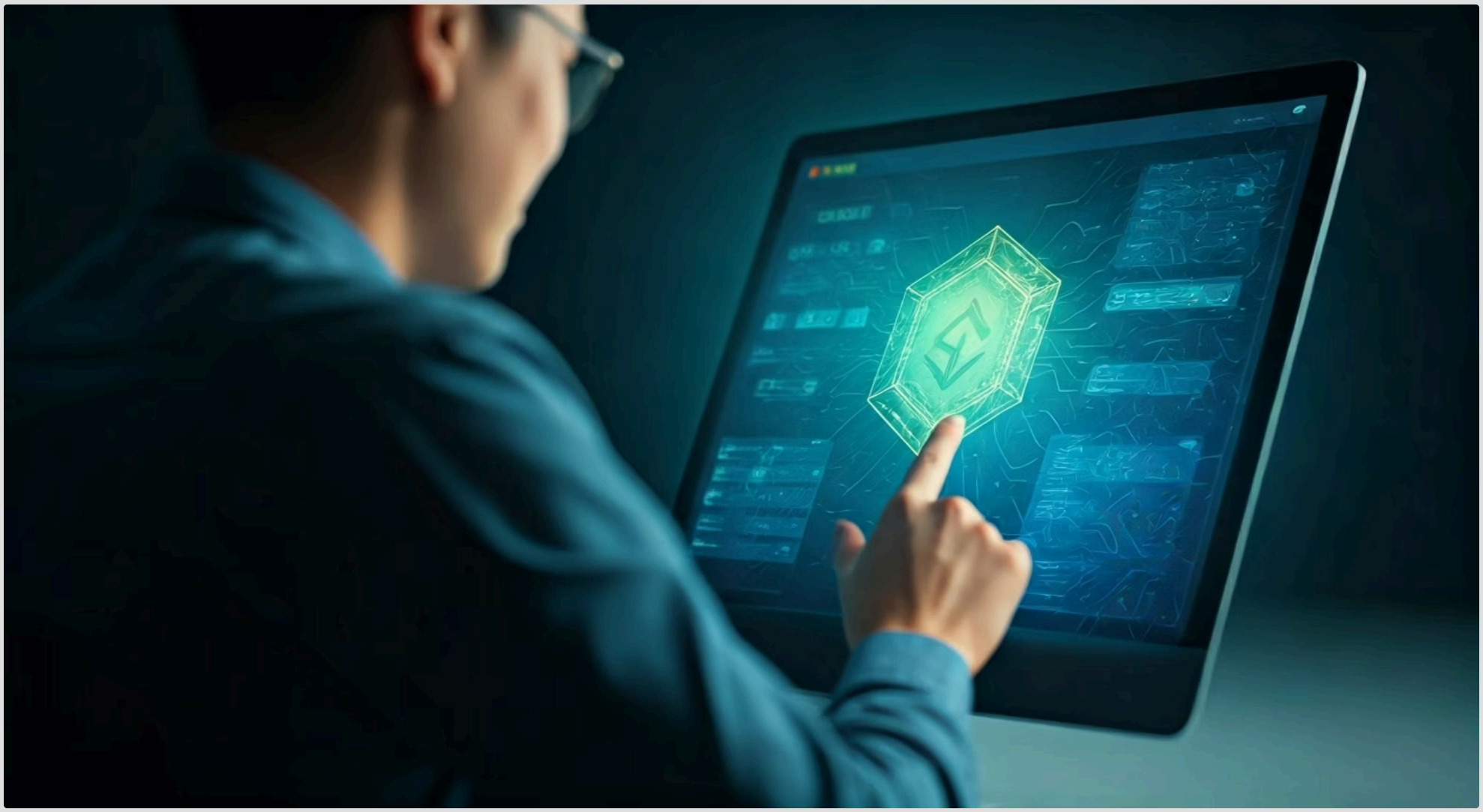
Aula 20 – Escrevendo Transações a partir do Frontend



Imagine um mundo onde você pode interagir diretamente com a lógica de um programa, não apenas lendo informações, mas também alterando seu estado, tudo isso de forma transparente e segura. No universo dos smart contracts, essa interação é a essência das aplicações descentralizadas (DApps). Até agora, exploramos como os smart contracts funcionam por dentro, mas a verdadeira magia acontece quando os usuários podem acioná-los com um simples clique em uma interface web.

Este é o ponto onde a teoria encontra a prática, onde o código Solidity que você aprendeu a escrever ganha vida nas mãos de um usuário final. Sem uma forma eficaz de interagir com esses contratos a partir de um navegador, nossas DApps seriam apenas peças de engenharia brilhantes, mas inacessíveis. É aqui que entra o frontend, a camada visível e interativa que traduz a complexidade da blockchain em uma experiência de usuário intuitiva.

Ao final desta aula, você não apenas entenderá os mecanismos por trás da comunicação entre o frontend e a blockchain, mas também será capaz de configurar seu ambiente de desenvolvimento, chamar funções de smart contracts que alteram o estado da rede e lidar com os resultados dessas transações. Nosso objetivo é capacitá-lo a construir interfaces que permitam aos usuários interagir plenamente com seus contratos inteligentes, transformando ideias em aplicações descentralizadas funcionais e seguras.

A Ponte entre o Usuário e a Blockchain: O Papel do Frontend

No dia a dia, quando usamos um aplicativo de banco ou uma rede social, estamos interagindo com uma interface gráfica que, por sua vez, se comunica com servidores complexos nos bastidores. No mundo das DApps, a dinâmica é semelhante, mas com uma diferença crucial: em vez de servidores centralizados, a interface do usuário (o frontend) se conecta diretamente à rede blockchain. Essa conexão é a espinha dorsal de qualquer aplicação descentralizada, permitindo que as ações do usuário sejam traduzidas em transações imutáveis na rede.

> **Ponto-chave:** O frontend atua como intérprete entre o usuário e a blockchain, traduzindo cliques e comandos em transações verificáveis.

O desafio reside em como essa ponte é construída. Um usuário comum não vai escrever código Solidity ou interagir diretamente com nós de blockchain. Ele precisa de um ambiente familiar, como um navegador web, e de elementos visuais, como botões e campos de texto. Para que um clique em um botão "Enviar" no seu navegador se transforme em uma transação na blockchain, precisamos de ferramentas e bibliotecas que atuem como intérpretes e mensageiros, facilitando essa comunicação bidirecional.

Pense no frontend como o painel de controle de um carro autônomo. Você não precisa entender cada circuito eletrônico ou algoritmo complexo que faz o carro andar; você apenas interage com o volante, os pedais e a tela de navegação. Da mesma forma, o frontend de uma DApp é o painel que permite ao usuário "dirigir" seu smart contract, enviando comandos que alteram o estado da blockchain sem precisar mergulhar nos detalhes técnicos de como isso acontece. É uma abstração poderosa que torna a tecnologia acessível.

Configurando o Ambiente Frontend para Interação

Ferramentas Essenciais

- **Biblioteca JavaScript** (ethers.js ou web3.js)
- **Provedor de Blockchain** (MetaMask)
- **Conexão Segura** via objeto ethereum

Antes de podermos enviar qualquer transação, precisamos estabelecer uma conexão robusta e segura entre nosso frontend e a rede blockchain. Isso geralmente envolve algumas etapas essenciais, começando pela escolha de uma biblioteca JavaScript que nos permita interagir com a blockchain, como ethers.js ou web3.js. Essas bibliotecas são como os adaptadores universais que permitem que seu navegador "fale" a linguagem da blockchain.

Uma vez que temos a biblioteca, o próximo passo é conectar o frontend a um provedor de blockchain. Na maioria dos casos, isso significa conectar-se a uma carteira digital como o MetaMask, que atua como uma interface para o usuário gerenciar suas chaves privadas e assinar transações. O MetaMask, por exemplo, injeta um objeto ethereum no navegador, que as bibliotecas ethers.js ou web3.js podem usar para se comunicar com a rede. É um processo de "login" descentralizado, onde o usuário concede permissão para sua DApp interagir com sua carteira.

> **Analogia:** Imagine que você está em um aeroporto e precisa se comunicar com o controle de tráfego aéreo. Você não vai gritar para o céu; você usa um rádio específico (a biblioteca ethers.js), que se conecta à torre de controle (o provedor, como o MetaMask). A torre, por sua vez, tem a autoridade para enviar suas mensagens para os aviões (os smart contracts) na pista. Sem essa conexão e permissão, suas mensagens nunca sairiam do chão.

```
// Exemplo básico de conexão com ethers.js e MetaMask
import { ethers } from "ethers";

async function connectWallet() {
 if (window.ethereum) {
 try {
 const provider = new ethers.BrowserProvider(window.ethereum);
 const signer = await provider.getSigner();
 console.log("Carteira conectada:", await signer.getAddress());
 return { provider, signer };
 } catch (error) {
 console.error("Erro ao conectar a carteira:", error);
 // Tratar erros como usuário recusando a conexão
 }
 } else {
 console.warn("MetaMask não detectado. Por favor, instale-o.");
 // Sugerir instalação do MetaMask
 }
}
```

Interagindo com Contratos: O Objeto Contract

Com a conexão estabelecida e um "signer" (representando a conta do usuário que assinará as transações) em mãos, o próximo passo é instruir nosso frontend sobre qual smart contract queremos interagir e como. Para isso, precisamos de duas informações cruciais: o **endereço do contrato** na blockchain e a **ABI (Application Binary Interface)** do contrato.

Endereço do Contrato

O número de telefone exclusivo do seu smart contract na rede

ABI

O manual de instruções que descreve todas as funções e eventos públicos

O endereço do contrato é como o número de telefone exclusivo do seu smart contract na rede. Ele nos diz exatamente onde encontrar o contrato. A ABI, por outro lado, é um arquivo JSON que descreve todas as funções e eventos públicos do contrato, incluindo seus nomes, tipos de argumentos e tipos de retorno. Ela funciona como um manual de instruções, dizendo ao frontend quais comandos o contrato entende e como eles devem ser formatados. Sem a ABI, o frontend não saberia como "chamar" as funções do contrato de forma correta.

Pense na ABI como o cardápio de um restaurante e o endereço do contrato como a localização física do restaurante. Você precisa saber onde o restaurante está para ir até ele, e precisa do cardápio para saber quais pratos ele serve e como pedi-los. O objeto Contract que criamos com ethers.js ou web3.js é a representação programática desse restaurante, permitindo que você "faça pedidos" (chame funções) de forma estruturada.

```
// Exemplo de criação de um objeto Contract com ethers.js
import { ethers } from "ethers";

// Supondo que você já tenha o provider e signer de connectWallet()
async function getContractInstance(signer) {
 const contractAddress = "0x..."; // Substitua pelo endereço do seu contrato
 const contractABI = [
 // Array JSON da ABI do seu contrato
 "function setValue(uint _newValue) public",
 "function getValue() public view returns (uint)",
 "event ValueChanged(uint oldValue, uint newValue)"
 ];

 const contract = new ethers.Contract(contractAddress, contractABI, signer);
 return contract;
}
```

Chamando Funções de Escrita (State-Changing Functions)

01

Preparação da Transação

A biblioteca ethers.js prepara os dados da transação

02

Envio para Carteira

Os dados são enviados para o MetaMask do usuário

03

Confirmação do Usuário

O usuário revisa e aprova a transação, incluindo o custo de gás

04

Assinatura e Envio

A transação é assinada e enviada para a blockchain

Agora que temos nosso objeto contract configurado, podemos finalmente chamar as funções que alteram o estado da blockchain. Diferentemente das funções de leitura (view ou pure), que apenas consultam dados e não custam gás, as funções de escrita (aquelas que modificam variáveis de estado) exigem uma transação. Isso significa que elas precisam ser assinadas pela carteira do usuário e pagam uma taxa de gás para serem executadas pelos mineradores ou validadores da rede.

Quando você chama uma função de escrita a partir do frontend, a biblioteca ethers.js (ou web3.js) prepara os dados da transação e os envia para a carteira do usuário (como o MetaMask). A carteira então exibe uma janela de confirmação, mostrando os detalhes da transação, incluindo o custo estimado do gás. Somente após o usuário aprovar e assinar a transação, ela é enviada para a rede blockchain para ser processada. Este é um passo crucial de segurança, garantindo que o usuário tenha controle total sobre suas ações e fundos.

> **Analogia:** Imagine que você está enviando uma carta registrada importante. Você escreve a carta (chama a função), mas antes de enviá-la, precisa ir ao correio, preencher um formulário, pagar a taxa de envio (gás) e assinar para confirmar que você é o remetente. Só então a carta é despachada. No mundo blockchain, a carteira é o correio, e sua assinatura digital é a prova de que você autorizou o envio da "carta" (transação) para o "endereço" (smart contract).

```
// Exemplo de chamada de função de escrita
async function callSetValue(contract, newValue) {
 try {
 const tx = await contract.setValue(newValue);
 console.log("Transação enviada:", tx.hash);
 // tx.hash é o identificador único da transação na blockchain
 return tx;
 } catch (error) {
 console.error("Erro ao enviar transação:", error);
 // Tratar erros como usuário recusando a transação ou gás insuficiente
 }
}
```

Lidando com o Retorno da Transação e Eventos

O que é o Transaction Hash?

O hash da transação é como um **número de rastreamento** de um pacote: ele identifica sua transação na rede, mas não significa que ela já foi confirmada.

A transação precisa ser incluída em um bloco e validada pela rede, o que pode levar alguns segundos ou minutos, dependendo da congestão da rede.

Função tx.wait()

Para garantir que a transação foi processada com sucesso e obter informações detalhadas sobre sua execução, precisamos "esperar" pela sua confirmação. A função `tx.wait()` é essencial para isso.

Após enviar uma transação, o trabalho do frontend não termina. A chamada await contract.setValue(newValue) retorna imediatamente um objeto de transação (tx) que contém o hash da transação. Este hash é como um número de rastreamento de um pacote: ele identifica sua transação na rede, mas não significa que ela já foi confirmada. A transação precisa ser incluída em um bloco e validada pela rede, o que pode levar alguns segundos ou minutos, dependendo da congestão da rede.

Para garantir que a transação foi processada com sucesso e obter informações detalhadas sobre sua execução, precisamos "esperar" pela sua confirmação. A função tx.wait() é essencial para isso. Ela pausa a execução do código até que a transação seja minerada e confirmada, retornando um receipt (recibo) que contém informações valiosas, como o status da transação (sucesso ou falha), o gás consumido e, crucialmente, os **eventos** emitidos pelo smart contract.

Status da Transação

Sucesso ou falha da execução

Gás Consumido

Quantidade exata de gás utilizado

Eventos Emitidos

Notificações do smart contract

Os eventos são uma forma poderosa de o smart contract "notificar" o mundo exterior sobre o que aconteceu. Eles são armazenados na blockchain junto com o receipt da transação e podem ser facilmente lidos pelo frontend. Se o seu contrato emite um evento ValueChanged quando setValue é chamado, o frontend pode escutar por esse evento no receipt para atualizar a interface do usuário, confirmando visualmente que a alteração de estado ocorreu. É como receber uma notificação de entrega do seu pacote, com detalhes sobre o que foi entregue e por quem.

```
// Exemplo de espera por confirmação e leitura de eventos
async function waitForTransactionAndEvents(tx) {
 try {
 const receipt = await tx.wait(); // Espera pela confirmação
 console.log("Transação confirmada:", receipt.status === 1 ? "Sucesso" : "Falha");
 console.log("Gás consumido:", receipt.gasUsed.toString());

 // Exemplo de leitura de eventos
 for (const log of receipt.logs) {
 const parsedLog = contract.interface.parseLog(log);
 if (parsedLog && parsedLog.name === "ValueChanged") {
 console.log("Evento ValueChanged detectado:");
 console.log(" Old Value:", parsedLog.args.oldValue.toString());
 console.log(" New Value:", parsedLog.args.newValue.toString());
 }
 }
 return receipt;
 } catch (error) {
 console.error("Erro ao esperar pela transação:", error);
 }
}
```

Gerenciando Erros e Segurança no Frontend

Usuário Recusa

O usuário pode recusar a transação na carteira

Gás Insuficiente

Pode não haver gás suficiente para executar

Revert do Contrato

O smart contract pode reverter devido a condições internas

Rede Congestionada

A rede pode estar sobrecarregada

Interagir com a blockchain a partir do frontend não é isento de desafios, e um dos mais importantes é o tratamento robusto de erros. Transações podem falhar por diversos motivos: o usuário pode recusar a transação na carteira, pode não ter gás suficiente, o smart contract pode reverter a transação devido a uma condição interna (como um require falhando), ou a rede pode estar congestionada. Um frontend bem construído deve antecipar e gerenciar esses cenários de forma elegante, fornecendo feedback claro ao usuário.

Melhores Práticas de Segurança:

- Utilize blocos try-catch para capturar exceções
- Valide entrada de dados no frontend antes de enviar
- Adicione uma camada de defesa extra além das validações do contrato
- Use bibliotecas auditadas e padrões seguros

A utilização de blocos try-catch é fundamental para capturar e lidar com exceções. Além disso, a segurança deve ser uma prioridade máxima. Isso inclui a validação de entrada de dados no frontend antes de enviá-los para o smart contract, para evitar que dados maliciosos ou inesperados causem problemas. Embora o smart contract deva ter suas próprias validações robustas (como as que a OpenZeppelin oferece), uma camada de validação no frontend adiciona uma defesa extra e melhora a experiência do usuário, evitando transações desnecessárias que falhariam.

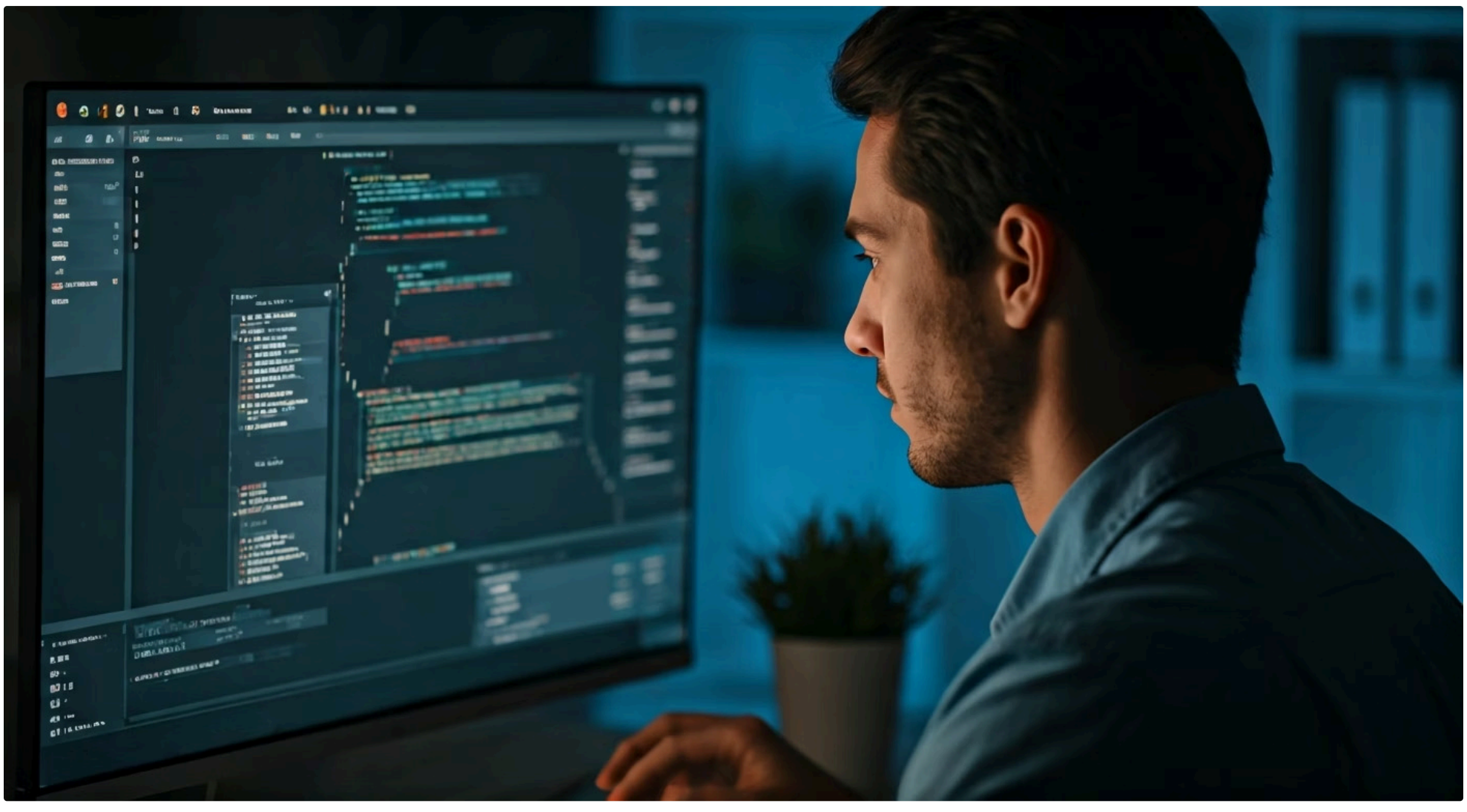
Conectando com as tendências de 2025, a segurança é um pilar inegociável. Ataques como a reentrância, embora sejam falhas no smart contract, podem ser mitigados também por um frontend que não permite interações maliciosas ou repetidas de forma inadequada. Utilizar bibliotecas auditadas e padrões de desenvolvimento seguros, tanto no backend (Solidity) quanto no frontend, é a melhor prática para proteger os usuários e a integridade da DApp.

```
// Exemplo de tratamento de erro com try-catch
async function safeCallSetValue(contract, newValue) {
 try {
 // Validação básica no frontend
 if (newValue < 0 || newValue > 1000) {
 throw new Error("Valor deve estar entre 0 e 1000.");
 }

 const tx = await contract.setValue(newValue);
 await tx.wait();
 alert("Valor atualizado com sucesso!");
 } catch (error) {
 console.error("Falha na transação:", error);

 if (error.code === 4001) { // Código comum para usuário recusando transação
 alert("Transação recusada pelo usuário.");
 } else if (error.message.includes("insufficient funds")) {
 alert("Gás insuficiente para a transação.");
 } else {
 alert("Ocorreu um erro: " + error.message);
 }
 }
}
```

O Papel do Hardhat no Desenvolvimento Frontend



Hardhat Network

Uma blockchain local em execução no seu computador, simulando a rede principal com todas as suas funcionalidades.

Vantagens

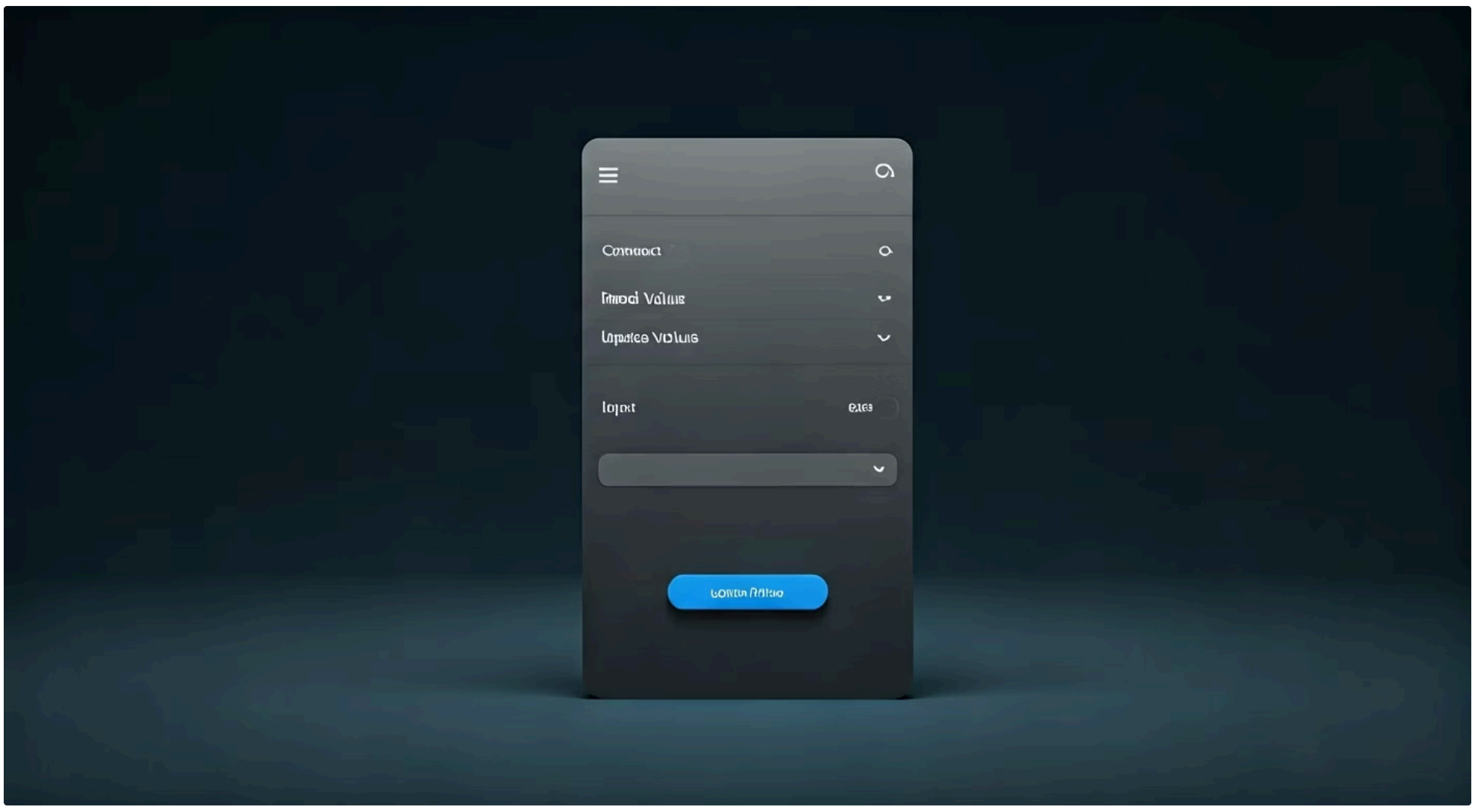
- Sem custos de gás
- Transações instantâneas
- Ambiente controlado e isolado
- Testes sem riscos

Desenvolver DApps exige um ciclo de feedback rápido: escrever o contrato, implantá-lo, testá-lo com o frontend e repetir. O Hardhat, um framework de desenvolvimento para Ethereum, otimiza drasticamente esse processo. Ele não apenas facilita a escrita, compilação e implantação de smart contracts, mas também oferece um ambiente de desenvolvimento local que é inestimável para o frontend.

Com o Hardhat Network, você pode ter uma blockchain local em execução no seu computador, simulando a rede principal com todas as suas funcionalidades, mas sem custos de gás e com transações instantâneas. Isso permite que os desenvolvedores de frontend testem suas interações com contratos em um ambiente controlado e isolado, sem a necessidade de implantar em redes de teste públicas ou na rede principal. É como ter um laboratório particular onde você pode experimentar à vontade sem medo de quebrar algo ou gastar recursos reais.

> **Analogia:** Imagine que você está construindo um carro de corrida. Você não vai testar cada pequena modificação diretamente na pista de corrida oficial. Em vez disso, você usa um simulador avançado que replica as condições da pista, permitindo que você ajuste e teste cada componente do carro em tempo real, sem riscos ou custos. O Hardhat Network é esse simulador para suas DApps, acelerando o desenvolvimento e garantindo que o frontend e o smart contract funcionem em perfeita harmonia antes de serem lançados para o público.

Construindo uma Interface Simples: Um Caso Prático



Para consolidar todo o conhecimento, vamos esboçar como seria a construção de uma interface simples para interagir com um contrato que armazena um único valor numérico, permitindo que o usuário o visualize e o altere. Este é um exemplo clássico, mas que demonstra a jornada completa do frontend à blockchain.

Conectar Carteira

Chama connectWallet() e obtém provider e signer

Instanciar Contrato

Cria instância com endereço e ABI

Ler Valor

Chama getValue() sem transação

Atualizar Valor

Envia transação com novo valor

Primeiro, teríamos um botão "Conectar Carteira" que, ao ser clicado, chamaria a função connectWallet que vimos anteriormente, obtendo o provider e o signer. Uma vez conectado, o endereço da carteira do usuário poderia ser exibido na tela. Em seguida, usaríamos o signer para criar uma instância do nosso contrato, fornecendo seu endereço e ABI.

Com o contrato instanciado, poderíamos ter um botão "Ler Valor" que chamaria a função getValue() do contrato (uma função view, sem transação) e exibiria o resultado. Mais importante, teríamos um campo de entrada de texto e um botão "Atualizar Valor". Quando o usuário digitasse um número e clicasse em "Atualizar", a função safeCallSetValue seria acionada, enviando uma transação para a blockchain. A interface ficaria "carregando" enquanto a transação é processada, e então exibiria uma mensagem de sucesso ou erro, atualizando o valor exibido após a confirmação.

Estado: Desconectado

Exibir botão "Conectar Carteira"

Estado: Conectado

Mostrar endereço e habilitar interações

Estado: Processando

Exibir indicador de carregamento

Estado: Confirmado

Atualizar UI com novo valor e feedback

Este fluxo de trabalho, desde a conexão da carteira até a manipulação de transações e o tratamento de seus resultados, é o cerne da construção de DApps interativas. Ele exige uma compreensão clara de como as bibliotecas Web3 funcionam e como os smart contracts se comunicam com o mundo exterior, sempre com foco na experiência do usuário e na segurança.

Consolidação e Próximos Passos

Chegamos ao fim de nossa jornada sobre como dar vida aos smart contracts através de interfaces de frontend. Vimos que a interação entre o usuário e a blockchain é mediada por bibliotecas Web3, que se conectam a carteiras digitais como o MetaMask. Aprendemos a instanciar contratos usando seus endereços e ABIs, e a chamar funções de escrita que alteram o estado da rede, sempre com a necessidade de uma transação assinada e o pagamento de gás. A importância de esperar pela confirmação da transação e de lidar com eventos para feedback ao usuário foi destacada, assim como a necessidade de um tratamento robusto de erros e a priorização da segurança em todas as camadas. O Hardhat se mostrou um aliado poderoso para testar essas interações em um ambiente local.

Em prática

Para aplicar o que foi aprendido, comece configurando um projeto frontend simples (com React, Vue ou Vanilla JS) e instale ethers.js. Implante um contrato simples (como o Storage.sol do Hardhat) em uma rede de teste ou no Hardhat Network. Crie botões para conectar a carteira, ler um valor do contrato e, crucialmente, um campo de input e um botão para enviar uma transação que altere esse valor. Observe o fluxo de confirmação do MetaMask e como você pode usar tx.wait() para atualizar sua interface.

Autoavaliação

1. Qual é a principal função da ABI (Application Binary Interface) ao interagir com um smart contract a partir do frontend?
   - a) Definir o custo de gás de uma transação.
   - b) Descrever as funções e eventos públicos do contrato para o frontend.
   - c) Armazenar o código-fonte do contrato na blockchain.
   - d) Gerenciar as chaves privadas do usuário.
2. Ao chamar uma função de escrita (que altera o estado) de um smart contract a partir do frontend, qual das seguintes etapas é **obrigatória**?
   - a) A função deve ser marcada como pure no Solidity.
   - b) O usuário deve assinar a transação em sua carteira digital.
   - c) A transação não pode consumir gás.
   - d) O resultado da função é retornado imediatamente sem espera.
3. Qual biblioteca JavaScript é comumente utilizada para conectar um frontend à blockchain Ethereum e interagir com smart contracts?
   - a) jQuery
   - b) React.js
   - c) ethers.js
   - d) Node.js
4. O que a função tx.wait() de um objeto de transação (tx) faz em ethers.js?
   - a) Envia a transação para a rede blockchain.
   - b) Cancela a transação se ela demorar muito.
   - c) Espera que a transação seja confirmada na blockchain e retorna seu receipt.
   - d) Calcula o custo de gás da transação.
5. Explique a importância de tratar erros e implementar validações de segurança no frontend ao interagir com smart contracts, citando pelo menos dois cenários de falha comuns e como eles podem ser mitigados.

Gabarito

1. b) | 2. b) | 3. c) | 4. c)

Próxima Aula

Na Aula 21 – Tokens Fungíveis: O Padrão ERC-20, exploraremos um dos tipos de smart contracts mais difundidos e importantes: os tokens fungíveis. Veremos como o padrão ERC-20 define a criação e manipulação desses ativos digitais, e como a interação frontend que aprendemos hoje é fundamental para que os usuários possam transferir, aprovar e gerenciar seus tokens.

Recursos Adicionais

- **Documentação ethers.js**: Para aprofundar nas funcionalidades da biblioteca.
- **Documentação Hardhat**: Para explorar o ambiente de desenvolvimento local e testes.
- **OpenZeppelin Contracts**: Para entender os padrões de segurança em smart contracts.

> **NOTA IMPORTANTE**: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.