Aula 17 – Conectando com a Blockchain: Ethers.js

Imagine que você tem um aplicativo incrível rodando no seu computador, mas ele precisa conversar com um banco de dados que está em outro continente, protegido por criptografia e acessível apenas por um protocolo muito específico. Como você faria essa comunicação de forma segura e eficiente? No mundo da Web3, nossos "aplicativos" são os DApps (Aplicativos Descentralizados) e o "banco de dados" é a blockchain. Para que seu DApp possa interagir com essa rede complexa, ler dados, enviar transações e chamar funções de contratos inteligentes, precisamos de uma ferramenta poderosa que atue como um tradutor e um mensageiro confiável.

É exatamente aqui que entra o Ethers.js. Esta biblioteca JavaScript é a ponte essencial entre o seu frontend (a interface que o usuário vê e interage) e a blockchain Ethereum. Sem ela, seu DApp seria como um carro sem motor, incapaz de se mover e realizar suas funções. Aprender Ethers.js não é apenas uma habilidade técnica; é a chave para desbloquear a capacidade de construir aplicações descentralizadas verdadeiramente interativas e funcionais.

Ao final desta aula, você será capaz de compreender a arquitetura do Ethers.js, configurar uma conexão básica com a blockchain, e, crucialmente, instanciar e interagir com contratos inteligentes diretamente do código do seu frontend. Vamos explorar como ler informações da blockchain e como preparar seu DApp para enviar transações que modificam o estado da rede, sempre com um olhar atento às melhores práticas de segurança e às ferramentas modernas que o mercado adota. Prepare-se para dar vida aos seus DApps!

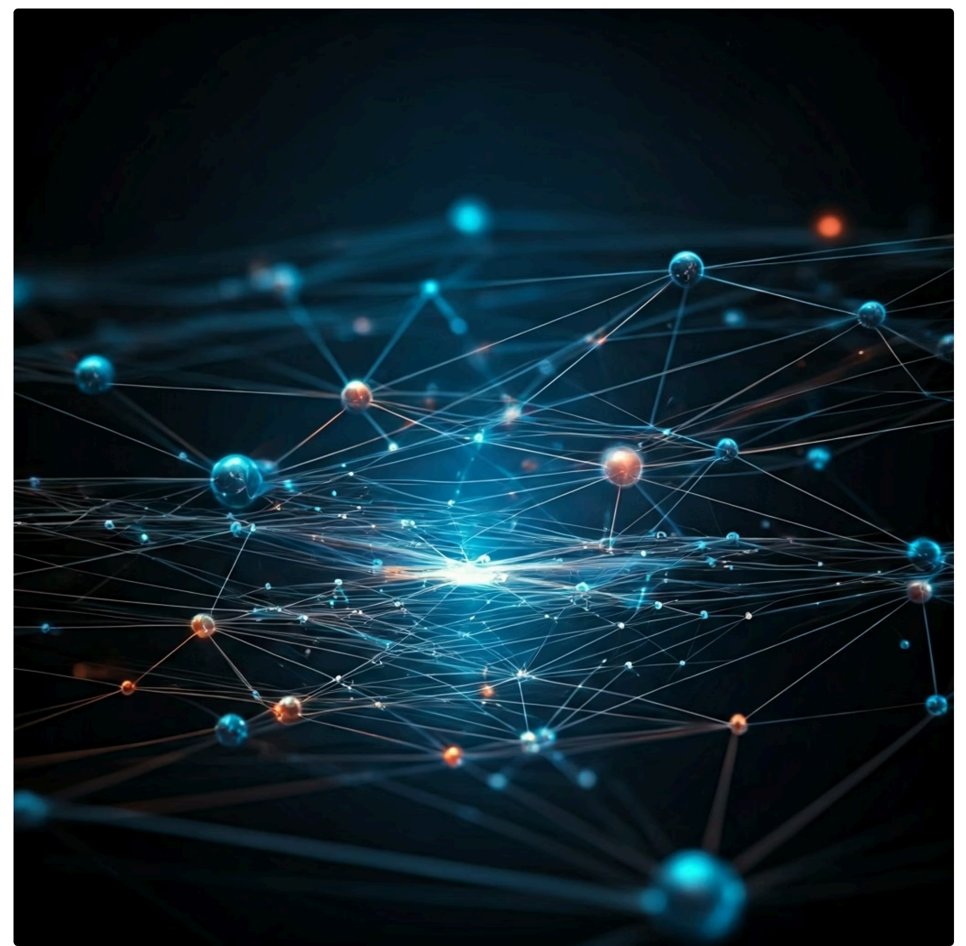
O Desafio da Comunicação Descentralizada

Web2: Comunicação Centralizada

No universo da Web2, quando seu aplicativo precisa de dados ou quer salvar alguma informação, ele simplesmente faz uma requisição para um servidor centralizado. É como pedir uma pizza: você liga para a pizzaria, eles recebem seu pedido e entregam. Simples, direto.

Web3: Comunicação Descentralizada

Na Web3, a blockchain é um sistema distribuído e descentralizado. Não há um servidor único para conversar. Em vez disso, seu DApp precisa interagir com uma rede de milhares de computadores (nós) que mantêm uma cópia do estado da blockchain.



O Desafio: Como seu código JavaScript, que roda no navegador do usuário, pode "falar" a linguagem da blockchain e garantir que suas interações sejam válidas e seguras?

É aqui que bibliotecas como o Ethers.js se tornam indispensáveis. Elas abstraem a complexidade de interagir diretamente com os nós da blockchain, fornecendo uma interface amigável e padronizada para que os desenvolvedores possam se concentrar na lógica do DApp, e não nos detalhes de baixo nível da comunicação de rede. Pense no Ethers.js como um intérprete universal que traduz suas intenções em comandos que a blockchain entende, e vice-versa.

Ethers.js: O Intérprete da Blockchain

Imagine que você está em um país estrangeiro e não fala o idioma local. Para se comunicar, você precisaria de um intérprete que traduzisse suas perguntas e as respostas que você recebe. No contexto da blockchain Ethereum, o Ethers.js desempenha exatamente esse papel. Ele é uma biblioteca JavaScript leve e robusta, projetada especificamente para facilitar a interação entre o seu frontend e a rede Ethereum.

Simplifica Operações

Criação de transações, assinatura de mensagens, interação com contratos

Foco em Segurança

Arquitetura modular e práticas seguras de desenvolvimento

API Moderna

Interface intuitiva e fácil de usar para desenvolvedores

Historicamente, o Web3.js foi uma das primeiras e mais populares bibliotecas para essa finalidade. No entanto, o Ethers.js ganhou muita tração nos últimos anos devido à sua arquitetura mais modular, foco em segurança e uma API mais moderna e fácil de usar. Muitos desenvolvedores e projetos, incluindo ferramentas como o Hardhat, o adotaram como a biblioteca padrão para interagir com a Ethereum, garantindo que o conhecimento em Ethers.js seja uma habilidade altamente relevante e valorizada no mercado atual.

Os Pilares do Ethers.js: Providers, Signers e Contracts

Para entender como o Ethers.js funciona, é fundamental conhecer seus três componentes principais, que atuam como os alicerces de qualquer interação com a blockchain. Pense neles como os membros de uma equipe bem coordenada, cada um com uma função específica, mas trabalhando juntos para alcançar um objetivo comum.

1

Providers

Função: Janela de leitura para a blockchain

Um Provider é responsável por se conectar a um nó Ethereum (seja ele local, como o Hardhat, ou remoto, como Infura ou Alchemy) e permitir que seu DApp consulte informações da rede. Quer saber o saldo de uma conta? Ou o estado atual de uma variável em um contrato inteligente? O Provider é quem faz essa pergunta e traz a resposta.

- Conexão com nós Ethereum
- Consultas de leitura apenas
- Não modifica o estado da blockchain

2

Signers

Função: Caneta para assinar transações

Um Signer representa uma conta Ethereum (geralmente uma carteira como o MetaMask) e é responsável por assinar transações e mensagens. Quando você quer enviar Ether, interagir com um contrato inteligente que modifica seu estado, ou aprovar uma operação, é o Signer que autoriza essa ação com sua chave privada.

- Representa uma conta Ethereum
- Assina transações e mensagens
- Autoriza modificações na blockchain

3

Contracts

Função: Interface para contratos inteligentes

Este componente permite que você interaja com contratos inteligentes já implantados na blockchain. Para isso, o Ethers.js precisa de duas informações cruciais: o endereço do contrato na rede e sua ABI (Application Binary Interface). Com essas duas peças, o Ethers.js cria uma representação JavaScript do seu contrato.

- Requer endereço do contrato
- Requer ABI do contrato
- Permite chamar funções do contrato

Providers: A Janela para a Blockchain

Metáfora

Imagine que a blockchain é uma vasta biblioteca cheia de livros de registros. Para ler esses livros, você precisa de um bibliotecário que saiba onde cada livro está, como acessá-lo e como interpretar suas informações. No Ethers.js, o **Provider** é esse bibliotecário.

Tipos de Providers

- **JsonRpcProvider**

  Conecta a um nó remoto (Infura, Alchemy) ou local (Hardhat)

- **BrowserProvider**

  Interage com o provedor injetado pela carteira (MetaMask)

- **InfuraProvider / AlchemyProvider**

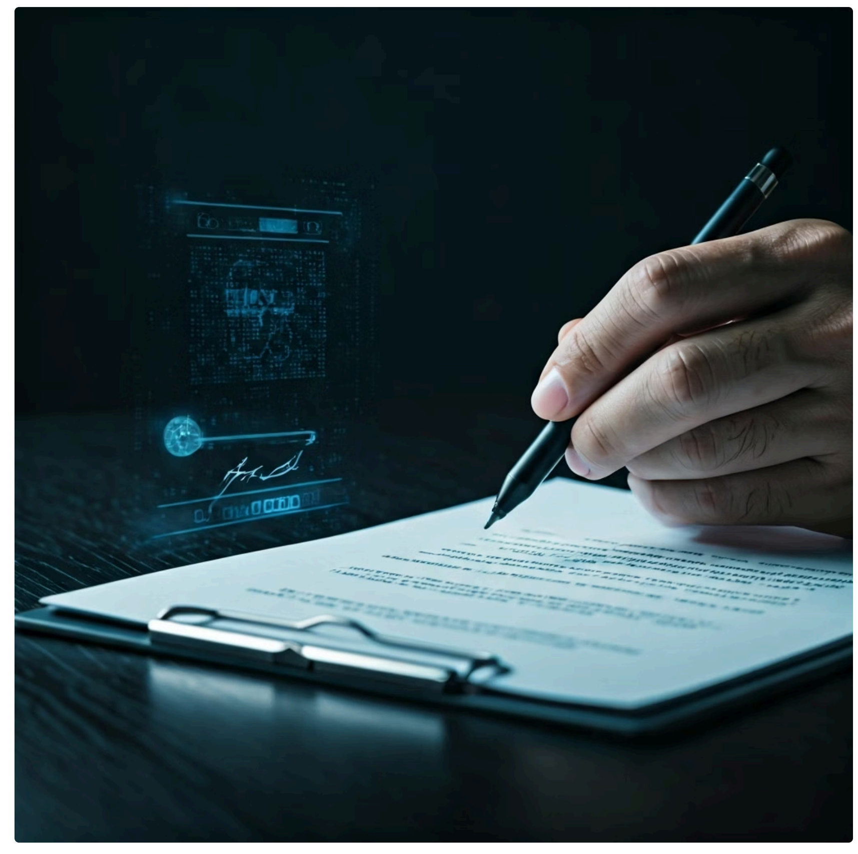
  Provedores especializados para serviços específicos

> **Responsabilidade Principal:** Fornecer métodos para consultar o estado da blockchain: obter o saldo de uma conta, o número do bloco atual, o código de um contrato, ou o resultado de uma chamada de função view ou pure de um contrato inteligente.

```
// Exemplo de como instanciar um Provider
// Conectando a um nó público (Infura, Alchemy, etc.)
const ethers = require('ethers'); // Ou import { ethers } from 'ethers';

// Usando um provedor padrão para a rede Goerli (exemplo)
// Você precisaria de sua própria chave API para um ambiente de produção
const provider = new ethers.JsonRpcProvider('https://goerli.infura.io/v3/SUA_CHAVE_API');

// Ou conectando a um nó local (como o Hardhat Network)
// const provider = new ethers.JsonRpcProvider('http://127.0.0.1:8545/');

async function getBlockNumber() {
 const blockNumber = await provider.getBlockNumber();
 console.log(`Número do bloco atual: ${blockNumber}`);
}

getBlockNumber();
```

Signers: A Chave para Ações na Blockchain

Se o Provider nos permite ler a blockchain, o **Signer** é quem nos dá o poder de agir, de escrever, de modificar o estado da rede. Pense no Signer como a sua assinatura digital, a prova de que você autoriza uma determinada transação. No contexto da Ethereum, isso significa que o Signer detém a chave privada de uma conta e a utiliza para assinar transações, tornando-as válidas e prontas para serem enviadas à rede.



01

Associação com Provider

Signer sempre conectado a um Provider para enviar transações

02

Integração com Carteiras

Interação via MetaMask ou outras carteiras de navegador

03

Assinatura Segura

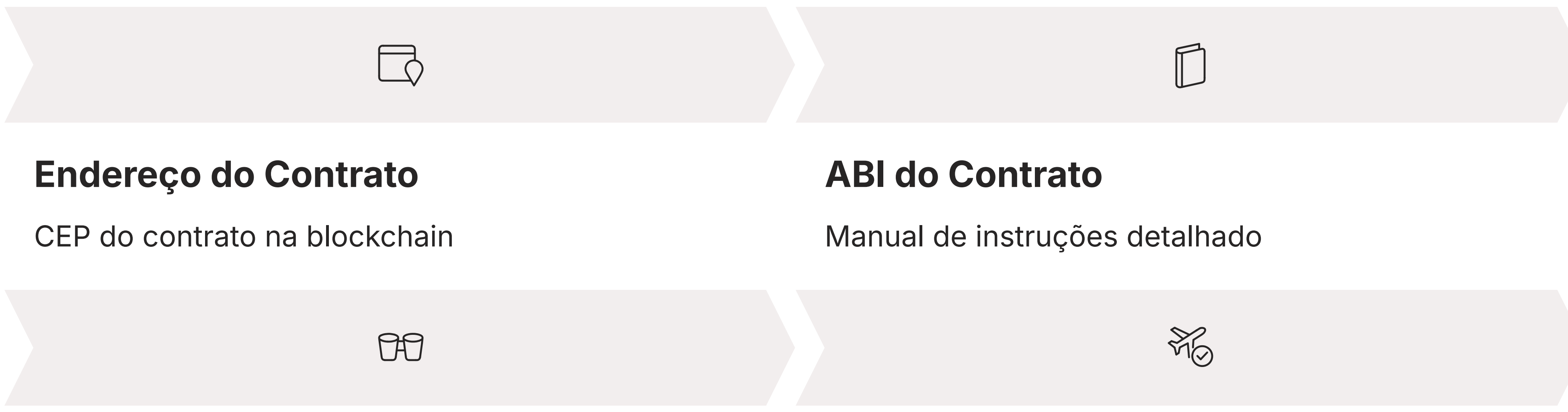
DApp solicita assinatura sem acessar chave privada

Responsabilidade: Assinar transações e mensagens, incluindo enviar Ether, chamar funções de contratos que modificam estado, ou assinar mensagens para provar posse de conta.

```
// Exemplo de como obter um Signer (geralmente via MetaMask)
// Este código seria executado no navegador
async function getSignerFromMetamask() {
 if (window.ethereum) {
 const provider = new ethers.BrowserProvider(window.ethereum); // Novo em ethers v6
 await provider.send("eth_requestAccounts", []); // Solicita conexão com a carteira
 const signer = await provider.getSigner();
 console.log(`Endereço do Signer: ${await signer.getAddress()}`);
 return signer;
 } else {
 console.error("MetaMask ou outra carteira Ethereum não detectada.");
 return null;
 }
}

// getSignerFromMetamask(); // Chamaria esta função para obter o signer
```

Contracts: Interagindo com a Lógica da Blockchain

Até agora, vimos como o Provider nos permite ler a blockchain e como o Signer nos capacita a assinar transações. Mas como fazemos para interagir com a lógica complexa de um contrato inteligente implantado? É aí que o objeto **Contract** do Ethers.js entra em cena. Ele é a representação JavaScript de um contrato inteligente na blockchain, permitindo que você chame suas funções e ouça seus eventos como se estivesse interagindo com um objeto JavaScript comum.



Endereço do Contrato

CEP do contrato na blockchain

ABI do Contrato

Manual de instruções detalhado

Provider ou Signer

Para leitura ou escrita

Objeto Contract

Pronto para interação

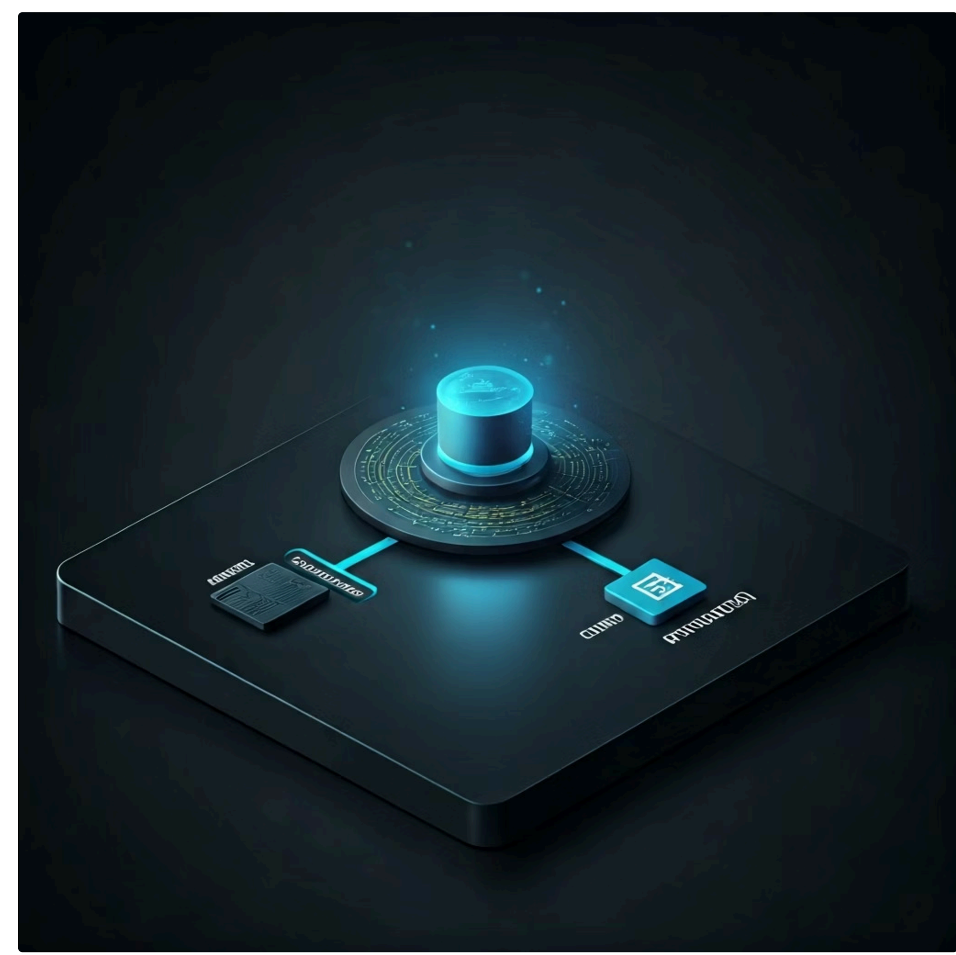
O que é ABI?

A **ABI (Application Binary Interface)** é um arquivo JSON que descreve todas as funções públicas e eventos do seu contrato Solidity. É como um dicionário que mapeia os nomes das funções que você conhece no Solidity para a forma como elas são representadas em bytecode na blockchain.

- Descreve funções e eventos
- Inclui nomes e tipos de parâmetros
- Define tipos de retorno
- Gerada automaticamente na compilação

Instanciando um Contract

Uma vez instanciado, o objeto Contract permite que você chame funções view ou pure (que apenas leem dados) usando o Provider associado. Para funções que modificam o estado, você precisará de um Signer associado ao objeto Contract.



```
// Exemplo de como instanciar um objeto Contract
const contractAddress = "0x..."; // Endereço do seu contrato implantado
const contractABI = [/* Sua ABI aqui, como um array de objetos JSON */];

// Primeiro, precisamos de um Provider (para leitura)
const provider = new ethers.JsonRpcProvider('https://goerli.infura.io/v3/SUA_CHAVE_API');

// Instanciando o contrato para operações de leitura
const myContract = new ethers.Contract(contractAddress, contractABI, provider);

async function readContractData() {
 // Exemplo: chamar uma função 'name()' de um contrato ERC-20
 const name = await myContract.name();
 console.log(`Nome do contrato: ${name}`);
}

// readContractData(); // Chamaria esta função para ler dados
```

Preparando o Ambiente de Desenvolvimento

Antes de mergulharmos nos exemplos práticos de conexão e interação, precisamos garantir que nosso ambiente de desenvolvimento esteja configurado corretamente. Pense nisso como preparar sua bancada de trabalho antes de começar um projeto complexo: você precisa das ferramentas certas e do espaço organizado.

Instalar Node.js

Verifique se o Node.js está instalado digitando `node -v` e `npm -v` no terminal. Se não estiver, visite nodejs.org e baixe o instalador.

Criar Projeto

Crie um novo diretório para o seu projeto e inicialize um projeto Node.js com `npm init -y`

Instalar Ethers.js

Instale a biblioteca Ethers.js usando o comando `npm install ethers`

Comandos de Instalação

```
# 1. Crie um novo diretório para o seu projeto
mkdir meu-dapp-ethers
cd meu-dapp-ethers

# 2. Inicialize um novo projeto Node.js
npm init -y

# 3. Instale a biblioteca Ethers.js
npm install ethers
```

Após esses passos, você terá o Ethers.js pronto para ser importado em seus arquivos JavaScript, seja para construir um frontend interativo ou para escrever scripts de automação que interagem com a blockchain. Este é o ponto de partida para qualquer desenvolvimento com Ethers.js.

Conexão Básica: Instanciando um Provider

Com o ambiente configurado, o primeiro passo prático para qualquer DApp é estabelecer uma conexão com a blockchain. Como discutimos, o **Provider** é a sua porta de entrada para essa conexão. Instanciar um Provider é como ligar o rádio para sintonizar uma estação: você escolhe a frequência (a rede Ethereum) e o Ethers.js faz o resto, permitindo que você ouça as informações que vêm da rede.

Desenvolvimento Local

Conecte-se a um nó local como o Hardhat Network para testes rápidos e desenvolvimento sem custos.

Serviços Remotos

Use Infura ou Alchemy para acessar redes públicas (Mainnet, Sepolia, Goerli) sem rodar seu próprio nó.

Carteira do Usuário

No frontend, use o provedor injetado pelo MetaMask para interagir através da conta do usuário.

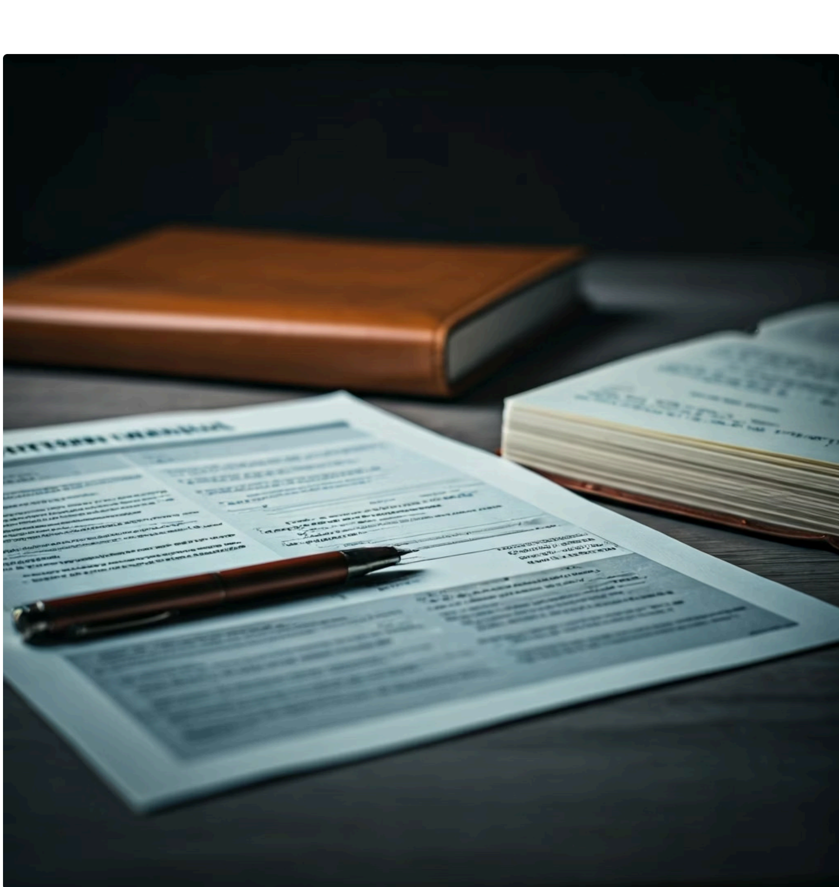
Exemplos de Diferentes Tipos de Providers

```
// Exemplo de diferentes tipos de Providers
const { ethers } = require("ethers"); // Para Node.js

// 1. Conectando a um nó local (ex: Hardhat Network)
const localProvider = new ethers.JsonRpcProvider("http://127.0.0.1:8545/");
console.log("Conectado ao nó local.");

// 2. Conectando a um serviço de nó remoto (ex: Infura para Sepolia)
// Substitua 'SUA_CHAVE_API_INFURA' pela sua chave real
const infuraProvider = new ethers.InfuraProvider("sepolia", "SUA_CHAVE_API_INFURA");
console.log("Conectado ao Infura (Sepolia).");

// 3. Conectando a um serviço de nó remoto (ex: Alchemy para Goerli)
// Substitua 'SUA_CHAVE_API_ALCHEMY' pela sua chave real
const alchemyProvider = new ethers.AlchemyProvider("goerli", "SUA_CHAVE_API_ALCHEMY");
console.log("Conectado ao Alchemy (Goerli).");

// Para frontend com MetaMask (exemplo conceitual, precisa de ambiente de navegador)
/*
async function connectToMetamaskProvider() {
  if (window.ethereum) {
    const browserProvider = new ethers.BrowserProvider(window.ethereum);
    const network = await browserProvider.getNetwork();
    console.log(`Conectado ao MetaMask na rede: ${network.name}`);
  } else {
    console.warn("MetaMask não detectado.");
  }
}
connectToMetamaskProvider();
*/
```

Escolher o Provider correto é o primeiro passo crucial para qualquer DApp, definindo como ele se comunicará com a rede Ethereum.

Lendo Dados de um Contrato Inteligente: ABI e Endereço

Com um Provider instanciado, já podemos ler informações gerais da blockchain, como o número do bloco ou o saldo de uma conta. Mas o verdadeiro poder dos DApps reside na interação com contratos inteligentes. Para que seu DApp possa "entender" e "falar" com um contrato específico, ele precisa de duas peças de informação que são como o "manual de instruções" e o "endereço" do contrato.



ABI

Arquivo JSON que descreve funções e eventos públicos do contrato

Endereço

Local único onde o contrato foi implantado na blockchain

Importante: A ABI é como um dicionário que mapeia os nomes das funções que você conhece no Solidity para a forma como elas são representadas em bytecode na blockchain. Sem a ABI, o Ethers.js não saberia quais funções estão disponíveis no contrato, quais parâmetros elas esperam e quais tipos de dados elas retornam. Ferramentas como o Hardhat geram a ABI automaticamente quando você compila seus contratos.

Exemplo: Lendo o símbolo de um token ERC-20

```
// Exemplo: Lendo o símbolo de um token ERC-20
const { ethers } = require("ethers");

// 1. Endereço do contrato (ex: DAI na rede Goerli)
const daiAddress = "0x11fE4B6AEce7085aeeE79DfEABc2aE61B5eE65Ee"; // Exemplo de endereço DAI em Goerli

// 2. ABI simplificada do contrato (apenas as funções que queremos usar)
// Em um projeto real, você importaria a ABI completa de um arquivo JSON
const daiABI = [
 "function name() view returns (string)",
 "function symbol() view returns (string)",
 "function totalSupply() view returns (uint256)",
 "function balanceOf(address owner) view returns (uint256)"
];

// 3. Instancie um Provider
const provider = new ethers.JsonRpcProvider('https://goerli.infura.io/v3/SUA_CHAVE_API');

// 4. Instancie o objeto Contract
const daiContract = new ethers.Contract(daiAddress, daiABI, provider);

async function getDaiInfo() {
 try {
 const name = await daiContract.name();
 const symbol = await daiContract.symbol();
 const totalSupply = await daiContract.totalSupply();

 console.log(`Nome do Token: ${name}`);
 console.log(`Símbolo do Token: ${symbol}`);
 console.log(`Total Supply: ${ethers.formatUnits(totalSupply, 18)}`); // DAI tem 18 decimais
 } catch (error) {
 console.error("Erro ao ler informações do contrato DAI:", error);
 }
}

getDaiInfo();
```

Instanciando um Contrato no Frontend (Operações de Leitura)

Agora que entendemos os componentes fundamentais e como obter a ABI e o endereço de um contrato, vamos ver como instanciar um contrato no código do frontend especificamente para operações de leitura. Isso é crucial para DApps que precisam exibir informações em tempo real da blockchain, como o saldo de um token, o status de um pedido ou qualquer dado armazenado em um contrato inteligente.

Apenas Provider Necessário

Para operações de leitura, você só precisa de um Provider. O Signer não é necessário porque você não está modificando o estado da blockchain.

Carregar ABI de Arquivo

É uma prática comum carregar a ABI do contrato a partir de um arquivo JSON gerado pelo seu ambiente de desenvolvimento.

Objeto Somente Leitura

Ao instanciar o contrato com o Provider, você está criando um objeto que é uma representação "somente leitura" do seu contrato inteligente.

Escrevendo Dados para um Contrato Inteligente: O Papel do Signer

Ler dados da blockchain é apenas metade da história. Para que seu DApp seja verdadeiramente interativo, ele precisa ser capaz de modificar o estado da rede, seja enviando tokens, registrando informações ou executando lógica de negócios em um contrato inteligente. Para essas operações de escrita, o **Signer** se torna indispensável. Ele é a sua permissão, a sua autorização para que uma transação seja enviada e processada pela blockchain.

Como Funciona

1. Você instancia o objeto Contract com um **Signer** em vez de apenas um Provider
2. Ao chamar uma função de escrita, o Ethers.js constrói a transação
3. O Signer (geralmente MetaMask) solicita aprovação do usuário
4. Usuário revisa e aprova a transação na carteira
5. Transação assinada é enviada para a rede através do Provider

Segurança

É um processo seguro, pois o DApp nunca tem acesso direto à chave privada do usuário; ele apenas solicita a assinatura.

```
// Este código seria executado em um arquivo JavaScript do seu frontend (ex: app.js)
import { ethers } from "ethers";
import contractABI from './MyContract.json'; // Assumindo que MyContract.json contém a ABI

// Endereço do contrato implantado (substitua pelo seu)
const myContractAddress = "0x...";

let provider;
let signer;
let myContractWithSigner;

async function connectWalletAndInstantiateContract() {
 if (window.ethereum) {
 provider = new ethers.BrowserProvider(window.ethereum);

 // Solicita ao usuário para conectar sua carteira
 await provider.send("eth_requestAccounts", []);
 signer = await provider.getSigner();
 console.log(`Carteira conectada: ${await signer.getAddress()}`);

 // Instancia o contrato com o Signer para operações de escrita
 myContractWithSigner = new ethers.Contract(myContractAddress, contractABI.abi, signer);
 console.log("Contrato instanciado com Signer para escrita.");
 } else {
 console.error("MetaMask ou outra carteira Ethereum não detectada.");
 alert("Por favor, instale o MetaMask para interagir com este DApp.");
 }
}

async function updateContractMessage(newMessage) {
 if (!myContractWithSigner) {
 alert("Por favor, conecte sua carteira primeiro.");
 return;
 }

 try {
 // Chama uma função de escrita do contrato (ex: setMessage)
 const tx = await myContractWithSigner.setMessage(newMessage);
 document.getElementById('status').innerText = `Transação enviada! Hash: ${tx.hash}`;
 console.log("Aguardando confirmação da transação...");

 await tx.wait(); // Espera a transação ser minerada
 document.getElementById('status').innerText = `Mensagem atualizada com sucesso!`;
 console.log("Transação confirmada!");

 // Opcional: recarregar a mensagem para mostrar a atualização
 // await initializeReadContract();
 } catch (error) {
 console.error("Erro ao enviar transação:", error);
 document.getElementById('status').innerText = `Erro: ${error.message}`;
 }
}
```

Instanciando um Contrato no Frontend (Operações de Escrita)

Continuando nossa jornada, a capacidade de instanciar um contrato para operações de escrita é o que realmente diferencia um DApp estático de um DApp dinâmico e interativo. Como vimos, isso envolve o uso de um **Signer**. A transição de um objeto Contract somente leitura para um que pode enviar transações é suave no Ethers.js, exigindo apenas que você forneça um Signer em vez de um Provider durante a instanciação.

01

Conectar Carteira

Solicitar ao usuário que conecte sua carteira (MetaMask)

02

Obter Signer

Obter o Signer a partir do provedor injetado pela carteira

03

Instanciar Contract

Criar instância do contrato com Signer como terceiro argumento

04

Chamar Funções

Métodos de escrita retornam TransactionResponse com hash e wait()

Exemplo: DApp de Contador Simples

Contrato Solidity

```
contract Counter {
  uint public count;

  constructor() {
    count = 0;
  }

  function increment() public {
    count++;
  }

  function decrement() public {
    count--;
  }

  function getCount() public view returns (uint) {
    return count;
  }
}
```

Interface do Usuário

O usuário pode ver o contador atual e clicar em botões para incrementar ou decrementar, enviando transações para a blockchain.

Lidando com Eventos e Confirmações de Transação

No mundo da blockchain, as transações não são instantâneas. Elas precisam ser processadas pelos mineradores (ou validadores em PoS) e incluídas em um bloco. Isso leva tempo, e seu DApp precisa ser capaz de lidar com essa assincronicidade, informando o usuário sobre o status da transação e reagindo a eventos que ocorrem na blockchain. O Ethers.js oferece ferramentas robustas para gerenciar isso.

1. Transação Enviada
DApp envia transação e recebe hash

2. Pending
Transação aguarda ser incluída em bloco

3. Confirmada
Transação incluída em bloco (tx.wait())

4. Evento Emitido
Contrato emite evento capturado pelo DApp

Aguardando Confirmações

O método `await tx.wait()` é seu melhor amigo. Ele pausa a execução do seu código até que a transação seja incluída em um bloco e tenha um número mínimo de confirmações (geralmente 1, mas pode ser configurado para mais para maior segurança).

> Isso é crucial para garantir que seu DApp só atualize sua interface ou execute lógica subsequente após a transação ter sido realmente processada pela rede.

Escutando Eventos

Contratos inteligentes podem emitir **eventos** para sinalizar que algo importante aconteceu. Pense nos eventos como "notificações" que o contrato envia. O Ethers.js permite que seu DApp "escute" esses eventos.

- Atualizar interface em tempo real
- Acionar outras lógicas no frontend
- Fins de auditoria e monitoramento

Melhores Práticas e Segurança com Ethers.js

Desenvolver DApps com Ethers.js não é apenas sobre fazer as coisas funcionarem; é sobre fazê-las de forma segura e eficiente. A segurança é uma prioridade máxima no ecossistema blockchain, e a forma como seu frontend interage com os contratos inteligentes pode ter implicações significativas. Adotar melhores práticas desde o início é fundamental para construir DApps robustos e confiáveis.

Validação de Entradas

Sempre valide as entradas do usuário no frontend antes de enviá-las para o contrato inteligente. Embora o contrato deva ter suas próprias validações, uma camada extra no frontend pode melhorar a experiência do usuário e evitar transações desnecessárias.

Bibliotecas Auditadas

Use bibliotecas auditadas, como a OpenZeppelin para seus contratos inteligentes, e certifique-se de que suas ABIs sejam as corretas para os contratos implantados.

Gerenciamento de Chaves

Nunca exponha chaves privadas no frontend. Sempre confie nas carteiras do usuário (como MetaMask) para assinar transações.

Estimativa de Gás

Esteja ciente das taxas de gás. O Ethers.js permite estimar o gás necessário para uma transação, o que pode ajudar a informar o usuário e evitar falhas por falta de gás.

Tabela de Conceitos de Segurança

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
Validação	Frontend e Smart Contract	Prevenção de erros e ataques	Verificar se um valor é positivo antes de enviar para uma função deposit
Chaves Privadas	Nunca no Frontend	Segurança criptográfica	MetaMask gerencia a assinatura, DApp apenas solicita
Gás	Custo de transação na Blockchain	Mecanismo de incentivo e anti-spam	contract.someFunction.estimateGas() para prever o custo
ABIs	Interface entre DApp e Contrato	Compilação do Smart Contract	Usar ABI gerada por Hardhat para garantir correspondência

```
// Exemplo de estimativa de gás
async function estimateGasForTransaction(signer, contract, functionName, ...args) {
 try {
 const gasEstimate = await contract.getFunction(functionName).estimateGas(...args);
 console.log(`Estimativa de gás para ${functionName}: ${gasEstimate.toString()}`);
 return gasEstimate;
 } catch (error) {
 console.error(`Erro ao estimar gás para ${functionName}:`, error);
 return null;
 }
}

// Uso:
// const estimatedGas = await estimateGasForTransaction(signer, myContractWithSigner, "setMessage", "Nova mensagem");
// if (estimatedGas) {
// const tx = await myContractWithSigner.setMessage("Nova mensagem", { gasLimit: estimatedGas });
// // ...
// }
```

Consolidação e Próximos Passos

Chegamos ao fim de nossa exploração sobre como conectar seu DApp à blockchain usando Ethers.js. Vimos que esta poderosa biblioteca atua como o intérprete essencial, permitindo que seu frontend leia dados (com Providers), assine transações (com Signers) e interaja com a lógica de contratos inteligentes (com objetos Contract). Desde a configuração básica do ambiente até a manipulação de eventos e a aplicação de melhores práticas de segurança, você agora tem uma base sólida para construir DApps interativos.

Em prática

Comece a experimentar! Crie um contrato Solidity simples (um contador, um registro de mensagens). Compile-o com Hardhat para obter a ABI e o endereço. Em seguida, crie um pequeno frontend HTML/JavaScript. Use Ethers.js para conectar-se ao MetaMask, ler o estado inicial do seu contrato e, finalmente, enviar uma transação para modificar esse estado. Observe as confirmações de transação e os eventos. Essa prática é a chave para solidificar seu aprendizado.

Autoavaliação

Questão 1

Qual dos componentes do Ethers.js é responsável por se conectar a um nó Ethereum e realizar *apenas* consultas de leitura?

1. Signer
2. Contract
3. Provider
4. Wallet

Questão 2

Para que um DApp possa chamar uma função de um contrato inteligente que *modifica* o estado da blockchain, qual componente do Ethers.js é indispensável?

1. Provider
2. ABI
3. Signer
4. Address

Questão 3

O que é a ABI (Application Binary Interface) de um contrato inteligente e por que ela é necessária para o Ethers.js?

1. É o endereço único do contrato na blockchain
2. É um arquivo JSON que descreve as funções e eventos do contrato
3. É a chave privada do contrato
4. É o protocolo de comunicação entre o DApp e o nó

Questão 4

Ao enviar uma transação de escrita com Ethers.js, qual método é comumente usado para aguardar a inclusão da transação em um bloco?

1. tx.send()
2. tx.confirm()
3. tx.wait()
4. tx.listen()

> **Questão 5:** Explique a importância de validar as entradas do usuário no frontend antes de enviá-las para um contrato inteligente, mesmo que o contrato já possua suas próprias validações.

Gabarito e Próximos Passos

Gabarito

1	2
c) Provider	c) Signer
3	**4**
b) É um arquivo JSON que descreve as funções e eventos do contrato, permitindo ao Ethers.js interagir com ele.	c) tx.wait()

Próxima Aula

Aula 18 – Interagindo com Wallets: O Papel do MetaMask

Na próxima aula, aprofundaremos ainda mais na experiência do usuário, explorando como as carteiras como o MetaMask se integram com os DApps, como gerenciar a conexão, as permissões e as assinaturas de forma segura e intuitiva.

Recursos Adicionais

- **Documentação Oficial do Ethers.js:** A fonte mais completa e atualizada para a biblioteca.
- **Documentação do Hardhat:** Para entender como compilar contratos e obter ABIs de forma eficiente.
- **OpenZeppelin Contracts:** Exemplos de contratos inteligentes seguros e auditados para referência.

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