Detecção Automática de Ambiente na Integração Foundry com Next.js

Este documento explica como a aplicação de tokenização detecta automaticamente em qual ambiente está sendo executada (desenvolvimento, testnet, produção) e como os contratos inteligentes são conectados de forma diferente dependendo do ambiente.

1. Arquivos de Configuração e Hooks React

A integração entre o Foundry (para desenvolvimento de smart contracts) e a aplicação Next.js é gerenciada principalmente pelos seguintes arquivos:

- `/app/lib/contracts/config.ts`: Define as configurações específicas para cada ambiente
- `/app/lib/contracts/hooks.ts`: Implementa os hooks React para interagir com os contratos
- `/app/lib/contracts/index.ts`: Exporta as configurações, hooks e ABIs dos contratos

Estrutura de Arquivos

```
/app
  /lib
    /contracts
      - AssetToken.abi.json
      - Marketplace.abi.json
      - Waitlist.abi.json
      - config.ts
      - hooks.ts
      - index.ts
```

2. Detecção Automática de Ambiente

A aplicação detecta automaticamente o ambiente em que está sendo executada através da variável de ambiente `NEXT_PUBLIC_NETWORK_ENV`. Esta variável é definida no arquivo `config.ts`:

```
// Ambiente atual
export const CURRENT_ENV = process.env.NEXT_PUBLIC_NETWORK_ENV || 'development';
```

Quando a variável `NEXT_PUBLIC_NETWORK_ENV` não está definida, o sistema assume automaticamente que está no ambiente de desenvolvimento.

Os possíveis valores para esta variável são: - `development`: Ambiente de desenvolvimento local - `testnet`: Ambiente de teste (Sepolia) - `production`: Ambiente de produção (Ethereum Mainnet)

3. Configurações Específicas para Cada Ambiente

Cada ambiente possui configurações específicas definidas no arquivo `config.ts`:

Endereços dos Contratos

```
export const CONTRACT_ADDRESSES = {
  // Endereços de desenvolvimento local (Anvil)
  development: {
    assetToken: '0x5FbDB2315678afecb367f032d93F642f64180aa3',
    marketplace: '0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512',
    waitlist: '0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0',
  },
  // Endereços de teste (Sepolia)
  testnet: {
    assetToken: '',
    marketplace: '',
    waitlist: '',
  },
  // Endereços de produção (Mainnet)
  production: {
    assetToken: '',
    marketplace: '',
    waitlist: '',
  },
};
```

URLs dos Provedores RPC

```
export const RPC_URLS = {
  development: 'http://localhost:8545',
  testnet: 'https://sepolia.infura.io/v3/YOUR_INFURA_KEY',
  production: 'https://mainnet.infura.io/v3/YOUR_INFURA_KEY',
};
```

Configuração da Rede

```
export const NETWORK_CONFIG = {
  development: {
    chainId: 31337,
    name: 'Anvil Local',
  },
  testnet: {
    chainId: 11155111,
    name: 'Sepolia',
  },
  production: {
```

```
    chainId: 1,
    name: 'Ethereum Mainnet',
  },
};
```

4. Conexão com os Contratos

A aplicação utiliza hooks React personalizados para conectar-se aos contratos inteligentes. Estes hooks são implementados no arquivo `hooks.ts` e utilizam as configurações específicas do ambiente atual.

Obtenção das Configurações do Ambiente Atual

O arquivo `config.ts` fornece funções auxiliares para obter as configurações do ambiente atual:

```
// Obter configuração com base no ambiente
export const getContractAddresses = () => CONTRACT_ADDRESSES[CURRENT_ENV as keyof typeof CON
export const getRpcUrl = () => RPC_URLS[CURRENT_ENV as keyof typeof RPC_URLS];
export const getNetworkConfig = () => NETWORK_CONFIG[CURRENT_ENV as keyof typeof NETWORK_CON
```

Conexão com os Contratos

Os hooks React utilizam estas funções para obter as configurações corretas para o ambiente atual:

```
// Hook para o contrato AssetToken
export function useAssetToken() {
  const { provider, signer, account } = useEthers();
  const [contract, setContract] = useState<ethers.Contract | null>(null);
  const addresses = getContractAddresses();

  useEffect(() => {
    if (provider) {
      const contractInstance = new ethers.Contract(
        addresses.assetToken,
        AssetTokenABI,
        signer || provider
      );
      setContract(contractInstance);
    }
  }, [provider, signer, addresses.assetToken]);

  // ...
}
```

O mesmo padrão é seguido para os outros contratos (Marketplace e Waitlist).

Fallback para Provedor Somente Leitura

Se o usuário não tiver o MetaMask instalado ou não estiver conectado, a aplicação utiliza um provedor somente leitura baseado na URL RPC do ambiente atual:

```
// Usar provedor somente leitura para fallback
const fallbackProvider = new ethers.providers.JsonRpcProvider(getRpcUrl());
setProvider(fallbackProvider);
```

5. Configuração Manual ao Mudar de Ambiente

Para mudar o ambiente da aplicação, é necessário definir a variável de ambiente `NEXT_PUBLIC_NETWORK_ENV` com um dos seguintes valores:

- `development`: Para desenvolvimento local com Anvil
- `testnet`: Para testes na rede Sepolia
- `production`: Para produção na rede Ethereum Mainnet

Esta variável pode ser definida:

1. No arquivo `.env.local` da aplicação Next.js:

   ```
   NEXT_PUBLIC_NETWORK_ENV=testnet
   ```

2. Durante a execução da aplicação:

   ```
   NEXT_PUBLIC_NETWORK_ENV=testnet npm run dev
   ```

3. No ambiente de hospedagem (Vercel, Netlify, etc.)

Além disso, para ambientes de testnet e produção, é necessário:

1. Preencher os endereços dos contratos implantados nas respectivas redes no arquivo `config.ts`
2. Substituir `YOUR_INFURA_KEY` pelas chaves reais da Infura nas URLs RPC

Conclusão

A aplicação de tokenização utiliza um sistema simples e eficaz para detectar automaticamente o ambiente em que está sendo executada. Através da variável de ambiente `NEXT_PUBLIC_NETWORK_ENV`, a aplicação carrega as configurações apropriadas para cada ambiente, permitindo uma transição suave entre desenvolvimento local, testnet e produção.

Os hooks React personalizados abstraem a complexidade da conexão com os contratos inteligentes, fornecendo uma interface consistente para interagir com eles, independentemente do ambiente em que a aplicação está sendo executada.