ChatGPT

Integração de Mint/Burn de Tokens na Rede Polygon no Projeto PrevisionMarket

1 Visão geral do problema

O repositório **PrevisionMarket** já implementa um fluxo de depósitos e saques em reais (BRL) via PIX e em USDC. Quando um depósito é aprovado, o servidor chama funções do módulo `brl3-client.ts` (como `notifyMintToBRL3` ou `notifyDualMintToBRL3`) para acionar um serviço externo que emite tokens BRL3 na rede Polygon. De forma análoga, saques aprovados chamam funções de burn.

O novo requisito é emitir e queimar tokens diretamente no contrato ERC-20 já implantado na Polygon (você informou que o contrato e o address já existem). Além disso, os usuários não devem pagar taxas de gas para queimar seus tokens durante um saque; o administrador da plataforma deverá arcar com essas taxas. A solução recomendada é utilizar **meta-transações**: o usuário assina uma autorização off-chain e o administrador (relayer) envia a transação e paga o gas.

O padrão 1">EIP-2612 introduz a função `permit`, que permite ao titular de um token aprovar um gasto com uma assinatura fora da cadeia. Isso simplifica a experiência do usuário (apenas uma assinatura) e reduz custos de gas [1]. A documentação da OpenZeppelin mostra que **meta-transações gasless** permitem ao usuário "assinar uma transação gratuitamente e ter essa transação executada por um terceiro que paga o gas" [2]. É exatamente o nosso caso: o usuário autoriza a transferência de seus tokens (por meio de `permit`) e o administrador executa as chamadas on-chain (`permit` → `transferFrom` → `burn`), arcando com as taxas de gas.

> **Importante**: a solução abaixo pressupõe que seu token na Polygon implementa as extensões `ERC20Permit` (para `permit`) e `ERC20Burnable` (para `burn`). Se o contrato atual não suportar essas funções, será necessário atualizar ou substituir o contrato para adicionar suporte a EIP-2612/permit.

2 Dependências necessárias

1. **ethers.js** – biblioteca para interagir com a Polygon.
2. **dotenv** – para ler chaves privadas e URLs do arquivo `.env`.
3. **ABI do contrato** – você precisa salvar o ABI do seu token (por exemplo exportado do Etherscan) em um arquivo JSON para que a aplicação possa chamar as funções `mint`, `burn`, `permit` e `transferFrom`.

Atualize o `package.json` adicionando `ethers` e `dotenv` às dependências:

```
// package.json (parcial)
"dependencies": {
```

```
  ...
  "ethers": "^6.8.0",
  "dotenv": "^16.3.1"
}
```

Crie ou atualize um arquivo `.env` na raiz do projeto com as seguintes variáveis (substitua pelos valores reais):

```
# URL RPC da Polygon (pode usar um endpoint da Alchemy, Infura, QuickNode, etc.)
POLYGON_RPC_URL=https://polygon-rpc.example.com

# Chave privada da carteira do administrador (esta carteira realiza as chamadas de mint e burn)
ADMIN_PRIVATE_KEY=0xSEUHEXAPRIVADO

# Endereço do contrato de token já implantado
TOKEN_CONTRACT_ADDRESS=0xSEU_CONTRACT_ADDRESS

# Número de decimais do token (ex: 18)
TOKEN_DECIMALS=18
```

Salve o ABI do contrato em `server/tokenABI.json`. Você pode obtê-lo do Etherscan (tab “Contract” → “Code” → “Contract ABI”).

3 Cliente Polygon (`server/polygonClient.ts`)

Crie um novo módulo `server/polygonClient.ts` para encapsular toda a interação com a blockchain. O código abaixo lê as variáveis de ambiente, instancia o provedor da Polygon, a carteira do administrador e o contrato. Ele fornece funções para mintar e queimar tokens, inclusive com permissão EIP-2612.

```
// server/polygonClient.ts
import { ethers } from "ethers";
import * as fs from 'fs';
import * as path from 'path';
import dotenv from 'dotenv';

// Carregar variáveis do .env
dotenv.config();

const {
  POLYGON_RPC_URL,
  ADMIN_PRIVATE_KEY,
  TOKEN_CONTRACT_ADDRESS,
  TOKEN_DECIMALS
```

```
} = process.env;

if (!POLYGON_RPC_URL || !ADMIN_PRIVATE_KEY || !TOKEN_CONTRACT_ADDRESS || !
TOKEN_DECIMALS) {
  throw new Error("Variáveis de ambiente Polygon incompletas. Verifique .env");
}

// Ler ABI do token
const abiPath = path.join(__dirname, 'tokenABI.json');
const tokenAbi = JSON.parse(fs.readFileSync(abiPath, 'utf8'));

const provider = new ethers.JsonRpcProvider(POLYGON_RPC_URL);
const adminWallet = new ethers.Wallet(ADMIN_PRIVATE_KEY, provider);
const tokenContract = new ethers.Contract(TOKEN_CONTRACT_ADDRESS, tokenAbi,
adminWallet);

// Converter valor humano (ex: 10.5) para unidades do token
function toUnits(amount: number): bigint {
  return ethers.parseUnits(amount.toString(), Number(TOKEN_DECIMALS));
}

/**
 * Mintar tokens para um endereço. Apenas a carteira do admin deve executar.
 * Retorna o hash da transação.
 */
export async function mintTo(toAddress: string, amount: number):
Promise<string> {
  const units = toUnits(amount);
  const tx = await tokenContract.mint(toAddress, units);
  await tx.wait();
  return tx.hash;
}

/**
 * Mintar a mesma quantidade de tokens para o usuário e para o admin (DUAL
MINT).
 */
export async function mintDual(userAddress: string, amount: number):
Promise<{userTx: string; adminTx: string}> {
  const units = toUnits(amount);
  const txUser = await tokenContract.mint(userAddress, units);
  const txAdmin = await tokenContract.mint(adminWallet.address, units);
  await txUser.wait();
  await txAdmin.wait();
  return { userTx: txUser.hash, adminTx: txAdmin.hash };
}

/**
```

```
 * Queimar tokens da carteira do admin (simples burn).  Use quando o admin já
 * possui tokens a queimar (por exemplo, após transferências).
 */
export async function burnFromAdmin(amount: number): Promise<string> {
  const units = toUnits(amount);
  const tx = await tokenContract.burn(units);
  await tx.wait();
  return tx.hash;
}

/**
 * Queima tokens de um usuário sem que ele pague gas.
 * O usuário assina offline uma autorização EIP-2612 (`permit`) e envia a
assinatura
 * para a API de saque.  O admin então chama `permit`, `transferFrom` e `burn`.
 *
 * @param owner endereço do usuário
 * @param amount quantidade de tokens (valor humano)
 * @param deadline timestamp (em segundos) até quando a autorização é válida
 * @param v,r,s componentes da assinatura EIP-712
 */
export async function burnWithPermit(owner: string, amount: number, deadline:
bigint, v: number, r: string, s: string): Promise<{permitTx: string;
transferTx: string; burnTx: string}> {
  const units = toUnits(amount);
  // 1. Chamar permit para aprovar o admin como spender
  const permitTx = await tokenContract.permit(owner, adminWallet.address,
units, deadline, v, r, s);
  await permitTx.wait();
  // 2. Transferir tokens do usuário para o admin
  const transferTx = await tokenContract.transferFrom(owner,
adminWallet.address, units);
  await transferTx.wait();
  // 3. Queimar os tokens da carteira do admin
  const burnTx = await tokenContract.burn(units);
  await burnTx.wait();
  return { permitTx: permitTx.hash, transferTx: transferTx.hash, burnTx:
burnTx.hash };
}

/**
 * Dupla queima: queima tokens do usuário (com permit) e a mesma quantidade do
admin.
 */
export async function burnDual(owner: string, amount: number, deadline: bigint,
v: number, r: string, s: string): Promise<{userTxs: {permitTx: string;
transferTx: string; burnTx: string}; adminBurnTx: string}> {
```

```
  const userTxs = await burnWithPermit(owner, amount, deadline, v, r, s);
  const adminBurnTx = await burnFromAdmin(amount);
  return { userTxs, adminBurnTx };
}
```

A função `burnWithPermit` exige a assinatura do usuário (`v`, `r`, `s` e `deadline`). Essas informações devem vir do frontend quando o usuário solicitar saque, conforme explicado na seção **5**.

4 Atualizar o cliente BRL3 (`server/brl3-client.ts`)

Atualmente esse módulo envia requisições HTTP para um serviço externo. Você pode reutilizar os nomes das funções (`notifyMintToBRL3`, `notifyBurnToBRL3`, `notifyDualMintToBRL3` e `notifyDualBurnToBRL3`) para manter o restante do código inalterado, mas implementá-las chamando o `polygonClient`. Substitua o conteúdo de `server/brl3-client.ts` pelo seguinte:

```
// server/brl3-client.ts
// Integração on-chain com o token da Polygon

import { mintTo, burnWithPermit, mintDual, burnDual } from './polygonClient';
import { db } from './db';
import { users } from '@shared/schema';

/**
 * Recupera o endereço da carteira Polygon de um usuário a partir de seu ID.
 * Esta função pressupõe que você adicionou um campo `walletAddress` na tabela
 * `users`.  Ajuste conforme a sua implementação.
 */
async function getUserWalletAddress(userId: string): Promise<string> {
  const user = await db.select({ address: (users as
any).walletAddress }).from(users).where(users.id.eq(userId)).limit(1);
  const addr = user?.[0]?.address;
  if (!addr) throw new Error(`Usuário ${userId} não possui carteira
configurada.`);
  return addr;
}

/**
 * Mintar tokens somente para o usuário (versão simples).  A quantidade recebida
 * deve ser um número em BRL; a função converte internamente para unidades do
token.
 */
export async function notifyMintToBRL3(userId: string, amount: number,
depositId: string): Promise<void> {
  const wallet = await getUserWalletAddress(userId);
  const txHash = await mintTo(wallet, amount);
  console.log(`[MINT] Depósito ${depositId}: mintado ${amount} tokens para $
```

```
{wallet}. Tx: ${txHash}`);
}

/**
 * Burn com permit (usuário paga zero gas).  É necessário que, ao solicitar o
saque,
 * o frontend envie a assinatura (v,r,s) e o deadline obtidos do usuário.  Para
 * simplificação, esta função recebe esses dados através do objeto
`permitData`.
 */
export async function notifyBurnToBRL3(userId: string, amount: number,
withdrawalId: string, permitData: {deadline: bigint; v: number; r: string; s:
string}): Promise<void> {
  const wallet = await getUserWalletAddress(userId);
  const { permitTx, transferTx, burnTx } = await burnWithPermit(wallet, amount,
permitData.deadline, permitData.v, permitData.r, permitData.s);
  console.log(`[BURN] Saque ${withdrawalId}: burned ${amount} tokens do usuário
${wallet}. PermitTx: ${permitTx}, TransferTx: ${transferTx}, BurnTx: ${burnTx}
`);
}

/**
 * DUAL MINT: mintar quantidade `amount` para o usuário e a mesma quantidade
para
 * a carteira do admin.  O front-end envia apenas o valor depositado, e esta
 * função lida com as transações.
 */
export async function notifyDualMintToBRL3(userId: string, amount: number,
depositId: string): Promise<void> {
  const wallet = await getUserWalletAddress(userId);
  const { userTx, adminTx } = await mintDual(wallet, amount);
  console.log(`[DUAL MINT] Depósito ${depositId}: ${amount} tokens para usuário
(tx ${userTx}) e ${amount} tokens para admin (tx ${adminTx})`);
}

/**
 * DUAL BURN: queima a mesma quantidade de tokens do usuário e do admin.  O
 * usuário assina permit e o admin paga gas.  Recebe os dados de assinatura via
 * `permitData`.  Queima primeiro os tokens do usuário e depois os do admin.
 */
export async function notifyDualBurnToBRL3(userId: string, amount: number,
withdrawalId: string, permitData: {deadline: bigint; v: number; r: string; s:
string}): Promise<void> {
  const wallet = await getUserWalletAddress(userId);
  const { userTxs, adminBurnTx } = await burnDual(wallet, amount,
permitData.deadline, permitData.v, permitData.r, permitData.s);
  console.log(`[DUAL BURN] Saque ${withdrawalId}: queima ${amount} tokens do
```

```
usuário (permit ${userTxs.permitTx}, transfer ${userTxs.transferTx}, burn $
{userTxs.burnTx}) e ${amount} tokens do admin (burn ${adminBurnTx})`);
}
```

> **Observação:** Para que `getUserWalletAddress` funcione é necessário incluir um campo `walletAddress` na tabela `users` e garantir que os usuários informem sua carteira Polygon ao se cadastrarem. Se preferir, é possível armazenar o endereço da carteira em outra tabela ou receber como parâmetro na requisição.

5 Fluxo de saque com `permit` (front-end)

No saque, o usuário deve assinar uma autorização EIP-712 fora da cadeia. O **front-end** solicita a assinatura e envia a assinatura (`v`, `r`, `s`), o `deadline` e o valor para o servidor junto com o pedido de saque. Exemplo usando ethers.js no navegador:

```
// Supõe que window.ethereum esteja disponível (MetaMask/Phantom) e que
// tokenContract seja uma instância de ethers.Contract conectada à carteira do
usuário.
import { ethers } from 'ethers';

async function signPermit(amount: number, tokenDecimals: number, tokenContract:
any, ownerAddress: string, spenderAddress: string) {
  const value = ethers.parseUnits(amount.toString(), tokenDecimals);
  const nonce = await tokenContract.nonces(ownerAddress);
  const deadline = BigInt(Math.floor(Date.now() / 1000) + 60 * 60); // 1 hora

  const domain = {
    name: await tokenContract.name(),
    version: '1',
    chainId: (await tokenContract.provider.getNetwork()).chainId,
    verifyingContract: tokenContract.target,
  };

  const types = {
    Permit: [
      { name: 'owner', type: 'address' },
      { name: 'spender', type: 'address' },
      { name: 'value', type: 'uint256' },
      { name: 'nonce', type: 'uint256' },
      { name: 'deadline', type: 'uint256' },
    ],
  };

  const message = {
    owner: ownerAddress,
    spender: spenderAddress, // endereço da carteira do admin (relayer)
```

```
    value,
    nonce,
    deadline,
  };

  const signer = tokenContract.runner;
  const signature = await signer.signTypedData(domain, types, message);
  const { v, r, s } = ethers.Signature.from(signature);
  return { deadline, v, r: r as string, s: s as string };
}

// Depois de assinar, envie { amount, currency: 'BRL', permitData } no corpo da
// requisição POST /api/wallet/withdraw/request para armazenar esses dados no
// banco.  Quando o admin aprovar o saque, esses dados serão repassados para
// notifyBurnToBRL3/notifyDualBurnToBRL3.
```

Salvando a assinatura

1. **Durante o pedido de saque** (`/api/wallet/withdraw/request`), o front-end deve incluir um campo `permitData` com `deadline`, `v`, `r` e `s`. É necessário atualizar a tabela de `pendingWithdrawals` para armazenar essas informações (ex.: colunas `permit_deadline`, `permit_v`, `permit_r`, `permit_s`).
2. **Na aprovação do saque**, o servidor deve recuperar o `permitData` da retirada pendente e repassá-lo para `notifyDualBurnToBRL3` ou `notifyBurnToBRL3` para realizar a queima.

6 Ajustes nas rotas de depósito e saque

- **Depósitos** (`/api/deposits/:id/approve`): após atualizar o saldo do usuário e criar o registro de transação, substitua a chamada a `notifyDualMintToBRL3` pela versão que recebe apenas o valor e o userId. Exemplo:

```
if (deposit.currency === 'BRL') {
  await notifyDualMintToBRL3(deposit.userId, depositAmount, depositId);
}
```

- **Saques** (`/api/withdrawals/:id/approve`): ao aprovar um saque em BRL, recupere o `permitData` da retirada pendente e passe-o para `notifyDualBurnToBRL3`:

```
if (withdrawal.currency === 'BRL') {
  const permitData = {
    deadline: BigInt(withdrawal.permit_deadline),
    v: withdrawal.permit_v,
    r: withdrawal.permit_r,
    s: withdrawal.permit_s,
  };
```

```
  await notifyDualBurnToBRL3(withdrawal.userId, withdrawAmount, withdrawalId,
permitData);
}
```

Certifique-se de adicionar as novas colunas `permit_deadline`, `permit_v`, `permit_r` e `permit_s` à tabela `pendingWithdrawals` no banco de dados.

7 Resumo

- O **EIP-2612** introduz o método `permit`, que permite aos usuários aprovarem gasto de tokens com uma assinatura off-chain, simplificando a experiência e reduzindo custos de gas [1].
- A OpenZeppelin descreve meta-transações gasless, nas quais o usuário assina uma mensagem off-chain e um relayer (administrador) envia a transação pagando o gas [2]. Esse modelo é usado na função `burnWithPermit` para que o usuário não pague taxas ao sacar.
- Um novo módulo `polygonClient.ts` encapsula a interação com o contrato ERC-20 na Polygon e oferece funções para mint e burn (incluindo `permit`).
- O módulo `brl3-client.ts` foi reescrito para chamar essas funções, mantendo a mesma interface pública.
- O front-end deve coletar a assinatura `permit` do usuário ao solicitar o saque e enviá-la ao servidor. O servidor executa `permit`, `transferFrom` e `burn`, pagando as taxas de gas.

Esta integração permite que depósitos via PIX emitam tokens diretamente na rede Polygon e que saques queimem os tokens de forma gasless para o usuário. Adapte os exemplos conforme a estrutura do seu banco de dados e garanta que o token implantado suporte `ERC20Permit` e `ERC20Burnable`.

---

[1] How To Approve Token Transfers Without Spending Gas Using EIP-2612 | Quicknode Guides
https://www.quicknode.com/guides/ethereum-development/transactions/how-to-use-erc20-permit-approval

[2] Relaying gasless meta-transactions with a web app | OpenZeppelin Docs
https://docs.openzeppelin.com/defender/guide/meta-tx