ChatGPT

Guia de Uso das APIs Helius: Webhooks, RPC e Transações

Helius fornece uma plataforma poderosa para interagir com a blockchain Solana por meio de Webhooks em tempo real, nós RPC de alto desempenho e APIs de **Transações Enriquecidas**. Este guia explica em detalhes cada funcionalidade – **Webhooks**, **RPC** e **Transações** – incluindo como funcionam, exemplos práticos de chamadas de API (com parâmetros e respostas), além de informações sobre custos, limites e melhores práticas de integração.

Webhooks Helius

Visão Geral e Tipos de Webhook

Os **Webhooks Helius** permitem monitorar eventos on-chain do Solana em tempo real, recebendo notificações HTTP sempre que ocorrerem ações específicas associadas a endereços que você acompanha [1]. Isso é útil para receber alertas instantâneos de operações como vendas de NFT, transferências de tokens, swaps DeFi, stakes, entre outros. A Helius oferece diferentes tipos de webhooks para atender a casos de uso variados [2]:

- **Enhanced Transaction Webhooks**: fornecem dados *parseados* (interpretados) e legíveis para humanos, filtrando apenas tipos específicos de transação (ex: `NFT_SALE`, `SWAP`, etc.) relacionados aos endereços monitorados [3]. Esse tipo é ideal quando você deseja receber eventos **enriquecidos** e já categorizados, focando apenas em informações relevantes e acionáveis (por exemplo, somente vendas de NFT de certos wallets).
- **Raw Transaction Webhooks**: entregam os dados brutos de qualquer transação envolvendo os endereços monitorados, **sem filtragem por tipo de evento** [4]. Você recebe o conteúdo da transação exatamente como ocorreu na blockchain (similar ao retorno de um `getTransaction` do RPC). Por não realizarem o parsing de alto nível, os webhooks *raw* oferecem latência menor – ou seja, notificam mais rapidamente – em comparação aos webhooks enriquecidos [5].
- **Discord Webhooks**: enviam atualizações sobre eventos específicos diretamente para um canal Discord via mensagens formatadas [6]. Nesse caso, em vez de um endpoint HTTP próprio, você fornece a URL de Webhook do Discord, e a Helius publicará as notificações lá (útil para notificações comunitárias em tempo real).

Dica: Se a prioridade for a rapidez na notificação (latência muito baixa), use webhooks do tipo *raw*, pois eles não esperam o parsing completo do evento [5]. Se precisar de eventos já interpretados (por exemplo, saber imediatamente quem foi o comprador e vendedor em uma venda de NFT), use os webhooks *enhanced*, ciente de que eles podem ter alguns milissegundos adicionais de atraso devido ao processamento.

Criação, Configuração e Gestão de Webhooks

Você pode criar e gerenciar webhooks de três formas: pela interface web (Dashboard Helius), via API REST ou usando os SDKs oficiais [7] [8]. A seguir explicamos o método recomendado para integrações

programáticas – via API – mas lembre que a dashboard oferece uma solução no-code (até 25 endereços por webhook no painel) [9] e os SDKs (TypeScript e Rust) fornecem abstrações convenientes [8].

1. Via API REST: A Helius expõe endpoints RESTful para criar, consultar, atualizar e deletar webhooks programaticamente [10]. Todas as chamadas exigem incluir sua API key da Helius (por exemplo, como query param `?api-key=SUACHAVE` ou cabeçalho, conforme autenticado). A criação de um webhook é feita com uma requisição HTTP POST para o endpoint `/v0/webhooks`. No corpo JSON, você especifica os parâmetros de configuração do webhook. Exemplo de criação de um webhook *enhanced*:

```
POST https://api.helius.xyz/v0/webhooks?api-key=SUACHAVE
Content-Type: application/json

{
  "webhookURL": "https://seu-endereco.com/meu-webhook",    // URL do seu
endpoint que receberá os eventos
  "accountAddresses": [
    "EndereçoOuContaAlvoAqui"
  ],
  "transactionTypes": [
    "NFT_SALE"
  ],
  "webhookType": "enhanced",
  "authHeader": "Bearer 123abc",
  "encoding": "json",
  "txnStatus": "finalized"
}
```

No exemplo acima: - `webhookURL` é o endpoint público do seu servidor que receberá as requisições POST do webhook.
- `accountAddresses` é a lista de endereços Solana que você deseja monitorar. Aqui pode ser um endereço de carteira (wallet), um token account, ou até a *public key* de um programa Solana. Qualquer transação que **mencione** um desses endereços (seja como remetente, destinatário ou mesmo envolvido nos accounts de uma instrução) poderá acionar o webhook. Você pode listar vários endereços (lembre-se que pela Dashboard há limite de 25, mas via API/SDK é possível adicionar mais endereços conforme necessidade) [9]. Para monitorar um programa inteiro, informe o endereço do programa; para monitorar atividades de um NFT específico, poderia usar o address da mint NFT ou conta associada, etc.
- `transactionTypes` define os filtros de tipos de transação que irão disparar o webhook. No exemplo, usamos `"NFT_SALE"` para receber somente eventos de venda de NFT. A Helius suporta **mais de 100 tipos** predefinidos de transação (NFT_SALE, NFT_LISTING, TRANSFER, SWAP, STAKE_SOL, etc.) [11] [12]. Você pode usar `"ANY"` para indicar "todos os tipos" (padrão utilizado geralmente para webhooks raw) [13] [14]. **Importante:** o filtro por tipo **só é válido** para webhooks do tipo *enhanced* – se você configurar `webhookType":"raw"`, a propriedade `transactionTypes` é ignorada e você receberá todos os eventos envolvendo os endereços monitorados [4].
- `webhookType` indica se o webhook será do tipo `"enhanced"` (eventos parseados) ou `"raw"` (eventos brutos). Use `"discord"` caso esteja configurando um webhook para um canal Discord.
- `authHeader` (opcional) permite definir um header de autorização que a Helius incluirá nas requisições enviadas ao seu endpoint. Por exemplo, você pode definir um token secreto aqui e validar, do lado do seu servidor, que cada chamada recebida contém esse header – assim você garante que a chamada veio da Helius.

- `encoding` (opcional) especifica o formato de envio. Tipicamente `"json"` (o padrão), mas a Helius poderia suportar outras codificações (ex: `base64`) para dados raw, se especificado. Para webhooks enhanced, mantenha JSON.
- `txnStatus` (opcional) define o nível de confirmação da transação para disparar o webhook. Pode ser `"confirmed"` para receber notificações assim que a transação for confirmada (mais rápido porém existe chance mínima de rollback se o fork não finaliza) ou `"finalized"` para receber apenas após finalização no bloco (um pouco mais lento, mas garante imutabilidade). Se não especificado, o padrão é provavelmente `"finalized"` (a documentação sugere que *commitment* padrão é finalized para consultas similares) – ajuste conforme sua necessidade de latência vs. certeza.

Resposta: Ao criar o webhook com sucesso (HTTP 200), a API retornará um objeto JSON com os dados do webhook criado, incluindo um `webhookID` único que identifica o recurso, os endereços e filtros configurados, etc. Por exemplo:

```
{
  "webhookID": "abc123...",
  "wallet": "Seu ID de conta Helius",
  "webhookURL": "https://seu-endereco.com/meu-webhook",
  "transactionTypes": ["NFT_SALE"],
  "accountAddresses": ["EndereçoOuContaAlvoAqui"],
  "webhookType": "enhanced",
  "authHeader": "Bearer 123abc"
}
```

Guarde o `webhookID`, pois ele será usado para editar ou remover o webhook via API.

- **Atualizar Webhook:** Para modificar um webhook existente (por exemplo, adicionar mais endereços ou mudar o filtro), use o endpoint PUT `/v0/webhooks/{webhookID}` com a estrutura JSON similar à de criação (incluindo somente os campos que quer alterar, mantendo os demais iguais). O `webhookID` vai na URL.
- **Deletar Webhook:** Para remover um webhook e parar as notificações, utilize DELETE `/v0/webhooks/{webhookID}`. Não é necessário corpo na requisição de delete – apenas certifique-se de passar a API key.
- **Listar Webhooks:** Você pode obter todos os webhooks configurados na sua conta usando GET `/v0/webhooks` (isso retorna um array com cada webhook e suas configurações). E para detalhes de um em específico, GET `/v0/webhooks/{webhookID}`.

Observação: Criar, editar ou deletar webhooks via API incorre em custo de 100 créditos por requisição [15] (veja seção de custos abaixo). Portanto, convém planejar suas configurações com cuidado para evitar mudanças excessivas frequentes.

Estrutura dos Eventos Recebidos

Uma vez configurado, o webhook ficará **escutando** as transações na rede Solana envolvendo os endereços alvo. Quando um evento é disparado, a Helius fará uma requisição HTTP POST para o seu

`webhookURL` contendo um **payload JSON** com os detalhes do evento. A estrutura desse JSON varia conforme o tipo do webhook:

- **Webhook Enhanced (Parseado):** O payload traz **um array JSON contendo um objeto por transação** (geralmente será um array com um único objeto, correspondente à transação recém-ocorrida que satisfaz os filtros). Cada objeto inclui diversos campos de alto nível que descrevem a transação de forma legível:
- `type`: o tipo categorizado do evento, por exemplo `"NFT_SALE"`, `"SWAP"`, `"TRANSFER"`, etc., conforme a lista de mais de 100 tipos suportados [11] [12].
- `source`: a fonte ou protocolo associado, se conhecido – por exemplo, `"MAGIC_EDEN"`, `"JUPITER"`, `"SYSTEM_PROGRAM"`, etc., indicando qual programa ou marketplace originou aquela transação [16] [17].
- `description`: uma frase descritiva em inglês resumindo o que aconteceu, incluindo contas e valores (por ex: *"Alice sold Fox #7637 to Bob for 72 SOL on MAGIC_EDEN."*) [18].
- `slot` e `timestamp`: o slot do bloco e timestamp UNIX da inclusão da transação [16].
- `signature`: a assinatura (transaction hash) da transação [19].
- `fee` e `feePayer`: taxa paga (em lamports) e o endereço pagador da taxa [20] [21].
- `nativeTransfers`: lista de transferências de SOL ocorridas na transação, mostrando `amount` (em lamports), `fromUserAccount` e `toUserAccount` para cada transferência nativa [22] [23].
- `tokenTransfers`: lista de transferências de tokens SPL, com campos análogos (contas de origem/destino, quantidade e mint do token, e padrão/tokenStandard se disponível) [24] [25].
- `accountData`: um array listando cada conta tocada na transação e as mudanças de saldo resultantes [26] [27]. Para cada conta envolvida, você tem o endereço (`account`), mudança de saldo nativo (`nativeBalanceChange` em lamports) e possivelmente `tokenBalanceChanges` (que detalha mudanças em tokens SPL, incluindo mint do token, quantidade antes/depois) [28] [29]. Isso permite ver facilmente quais contas ganharam ou perderam saldo devido à transação.
- `instructions`: um array das instruções de baixo nível executadas na transação, cada qual com os `programId` chamados, accounts usados e dados brutos (`data`) em base58 [30] [31]. Também inclui `innerInstructions` aninhadas quando aplicável (instruções disparadas dentro de invocações de programa) [31]. *Este campo é mais útil caso você precise depurar ou analisar detalhes técnicos específicos não cobertos pelos campos de alto nível.*
- `events`: talvez o mais importante, este campo consolida **detalhes específicos do tipo de evento**, facilitando sua manipulação. O conteúdo de `events` é um objeto que pode conter subobjetos como `nft`, `swap`, `stake`, etc., dependendo do tipo da transação. Por exemplo, em uma venda de NFT (`type: "NFT_SALE"`), haverá um objeto `events.nft` com campos especializados: `seller`, `buyer`, `nfts` (detalhes do NFT vendido, incluindo mint), `amount` (valor pago, em lamports), `saleType` (ex: `"INSTANT_SALE"` ou leilão), marketplace `source`, etc. [32] [33]. Em um swap DeFi (`type: "SWAP"`), pode haver um `events.swap` contendo `tokenInputs` e `tokenOutputs` com as quantias e contas trocadas, além de informações de pool. Em geral, esse campo `events` é onde a Helius fornece a interpretação pronta da transação – tornando desnecessário que você mesmo decodifique instruções comuns de NFT ou token. Se uma transação englobar múltiplas categorias (ex: uma operação que envolva um swap e uma NFT, algo raro), o payload poderia teoricamente ter múltiplos subobjetos em `events` (ex: `events: { nft: {...}, swap: {...} }`). Na maioria dos casos, será uma categoria principal só.

Exemplo: Abaixo um trecho (simplificado) de um payload de webhook **enhanced** para uma venda de NFT, ilustrando alguns campos descritos:

```
[
  {
    "type": "NFT_SALE",
    "source": "MAGIC_EDEN",
    "description": "AliceWallet sold NFT #7637 to BobWallet for 72 SOL on
MAGIC_EDEN.",
    "fee": 10000,
    "feePayer": "BobWalletPublicKey...",
    "signature": "5nNtjezQMYBHv...787HAmL",
    "slot": 171942732,
    "timestamp": 1673445241,
    "nativeTransfers": [ ... ],
    "tokenTransfers": [
      {
        "mint": "FdsNQ...mzYqu",
        "fromTokenAccount": "25DTU...fy9UiM",
        "toTokenAccount": "DTYuh7...GCs6Z",
        "fromUserAccount": "SellerPublicKey...",
        "toUserAccount": "BuyerPublicKey...",
        "tokenAmount": 1,
        "tokenStandard": "NonFungible"
      }
    ],
    "accountData": [ ... ],
    "events": {
      "nft": {
        "type": "NFT_SALE",
        "amount": 72000000000,
        "buyer": "BuyerPublicKey...",
        "seller": "SellerPublicKey...",
        "nfts": [{ "mint": "FdsNQ...mzYqu", "tokenStandard":
"NonFungible" }],
        "saleType": "INSTANT_SALE",
        "source": "MAGIC_EDEN",
        "fee": 10000,
        "feePayer": "BobWalletPublicKey..."
      }
    }
  }
]
```


Note como neste exemplo o objeto `events.nft` resume quem vendeu/comprou, o NFT e valor transacionado, enquanto campos como `tokenTransfers` e `nativeTransfers` detalham as transferências subjacentes. Para um desenvolvedor, isso facilita extrair informações de interesse (como notificar "*Bob comprou NFT X de Alice por Y SOL*").

- **Webhook Raw (Bruto):** O payload de um webhook tipo *raw* também é um array JSON com objetos de transação, **porém sem os campos interpretados de alto nível**. Em vez disso, você recebe a estrutura crua conforme fornecida pelo RPC do Solana. Essa estrutura contém:
- `blockTime`: timestamp UNIX do bloco [34].

- `indexWithinBlock`: posição da transação no bloco.
- `transaction`: objeto aninhado com os dados da transação (contendo as contas, instruções base64, assinaturas, etc. – equivalente ao campo `transaction` retornado pelo método RPC `getTransaction` com `encoding=json`).
- `meta`: objeto com metadados da execução da transação [35] – incluindo `err` (erro, se ocorreu, ou null se sucesso) [35], `fee` (taxa paga) [36], `preBalances` / `postBalances` (saldos antes/depois de cada account) [37], `innerInstructions` (instruções internas executadas, se houver) [38] [39], `logMessages` (todos os logs e eventos de programa gerados durante a execução) [40] [41], entre outros campos padrão do runtime Solana. Essencialmente, é a resposta completa que você obteria chamando RPC `getTransaction` (com opção de incluir logs, etc.) para aquela assinatura de transação [42] [35].

O payload raw é verboso e exige que você mesmo interprete as instruções e logs caso queira derivar ações de alto nível. Por outro lado, ele garante que **nenhuma informação é perdida** e chega alguns instantes antes do enhanced. A Helius envia o raw assim que a transação atinge o compromisso definido (confirmed/finalized), sem aguardar parseá-la.

Importante: independentemente do tipo, **cada evento enviado a você consome 1 crédito** da sua conta Helius [15] [43]. Ou seja, se seu webhook monitorar muitos endereços ativos ou filtros amplos, prepare-se para um volume potencialmente grande de requisições (cada uma descontando créditos). Filtrar pelos tipos relevantes (quando possível) é uma boa prática para economizar créditos e tráfego.

Custos e Limites dos Webhooks

- **Custos por Evento:** Conforme mencionado, cada notificação de evento enviada (cada POST para seu endpoint) custa **1 crédito** [15] [43]. Isso independe de seu sistema retornar sucesso ou erro – a Helius cobra pelo processamento/envio do evento, não pela recepção. Portanto, mesmo que seu servidor esteja indisponível ou retorne um erro, o evento foi contabilizado. *Dica:* garanta alta disponibilidade do seu endpoint e responda rapidamente (idealmente com HTTP 200 em poucos segundos) para não perder eventos. A Helius não especifica tentativas de reenvio; presumivelmente, cada transação gera uma única tentativa de webhook. Logo, se seu endpoint falhar no momento, aquele evento pode ser perdido (mas você já pagou o crédito). Por isso, **tenha uma infraestrutura confiável** ou alguma forma de redundância no processamento de webhooks (ex: enfileirar internamente para processamento posterior assim que receber).

- **Custos de Gerenciamento:** As operações de manutenção via API – criar, editar ou deletar webhooks – custam **100 créditos por requisição** [15] [44]. Esse valor relativamente alto significa que você deve evitar recriar webhooks desnecessariamente em produção. Procure configurar tudo de uma vez ou agrupar alterações. Se precisar monitorar novos endereços com frequência, considere adicionar vários endereços em um mesmo webhook (em vez de criar dezenas de webhooks separados), pois adicionar endereços via atualização custará 100 créditos mas pode incluir múltiplos endereços de uma vez.

- **Limites de Quantidade:** Não há um número explícito de webhooks simultâneos documentado para cada plano nas novas políticas – o sistema de créditos substituiu em grande parte limites rígidos. Em planos antigos (V3), o plano gratuito permitia 1 webhook ativo, desenvolvedor 3, business 10 e profissional 20 [45]. Embora os planos atuais não listem isso, é provável que a Helius agora permita múltiplos webhooks até onde seus créditos suportem. Ainda assim, **prefira consolidar monitoramento em menos webhooks** quando possível: cada webhook adicional significa outra chamada externa e mais coordenação. Por exemplo, se você quer monitorar 100

endereços para *NFT_SALE*, pode usar um único webhook enhanced com os 100 endereços na lista em vez de 100 webhooks separados – isso será muito mais eficiente.

- **Latência e Finalidade:** Considere o compromisso (`txnStatus`) adequado ao seu uso. Usar `"confirmed"` reduz latência (você recebe o evento assim que a transação entra em um bloco confirmado, geralmente 1-2s após submissão) mas há raríssimo risco de a transação ser revertida se não finalizou. `"finalized"` espera confirmação irreversível (poucos segundos extra). Para notificações críticas (ex: movimentação financeira final), finalized é mais seguro; para bots de trading ou reações muito rápidas, confirmed pode ser preferível.

- **Melhores Práticas:**

- **Desempenho do Endpoint:** Seu servidor deve responder rapidamente aos POSTs (ideal < 1s). Realize processamento pesado de forma assíncrona (por exemplo, enfileire a mensagem para processamento posterior) e retorne HTTP 200 o quanto antes. Assim você evita sobrecarga e garante que a Helius não fique aguardando (o que pode causar timeouts).
- **Segurança:** Utilize o `authHeader` para validar a origem das requisições. Outra abordagem é verificar o User-Agent e IPs, mas a forma mais robusta é incluir um token secreto via `authHeader` e checar em cada chamada recebida. Isso evita processar posts falsos enviados por terceiros à sua URL.
- **Filtro de Eventos:** Configure os `transactionTypes` pertinentes para reduzir ruído. Por exemplo, se você só quer saber de depósitos e saques de um token específico, talvez seja melhor monitorar transferências daquele token via filtro de programa ou mesmo usar a API de transações enriquecidas para pós-processar, ao invés de receber *tudo*. Webhooks raw para endereços muito movimentados podem gerar carga altíssima (e consumir rapidamente seus créditos). Sempre que possível, use *enhanced* com filtros ou restrinja a lista de `accountAddresses` ao mínimo necessário.
- **Testes:** A Dashboard Helius permite enviar eventos de teste para seu webhook, facilitando verificar se seu endpoint está consumindo corretamente os dados. Utilize essa função antes de ir a produção, simulando, por exemplo, um evento de NFT sale, para ajustar seu parser.
- **Logs e Monitoramento:** Os webhooks configurados via Dashboard exibem logs das entregas (sucesso/erro). Acompanhe esses logs durante a operação, pois podem alertar para falhas no seu endpoint (ex: muitos erros 500 podem indicar problemas). Além disso, monitore seu consumo de créditos no dashboard para evitar surpresas (1 crédito por evento pode acumular rápido se um endereço gerar milhares de transações em pouco tempo).

RPC (Remote Procedure Call) – Nodes Helius

Endpoints RPC da Helius e Autenticação

A Helius opera nós RPC de Solana de alto desempenho, oferecendo URLs públicos para acesso. Esses endpoints são compatíveis com a API JSON-RPC do Solana – ou seja, você pode usar os mesmos métodos e parâmetros disponíveis em um nó padrão, mas com melhorias de confiabilidade, velocidade e alguns recursos extras de infraestrutura.

Endpoints Principais: Para a rede principal (Mainnet-beta) e Devnet, utilize:

- **Mainnet RPC URL:** `https://mainnet.helius-rpc.com/?api-key=SUA_API_KEY` [46]
- **Devnet RPC URL:** `https://devnet.helius-rpc.com/?api-key=SUA_API_KEY` [46]

Substitua `SUA_API_KEY` pela sua chave de API fornecida no dashboard da Helius. A API key **é obrigatória** em todas as requisições; se omitida ou inválida, você receberá erros 401 Unauthorized. Você pode fornecer a API key na query string (como acima) ou via header `api-key` em alguns casos – porém, note que bibliotecas Solana comuns (como `@solana/web3.js`) geralmente esperam apenas a URL, então incluir na query string é o método mais simples.

A Helius anuncia que seus endpoints RPC são de alta performance e **totalmente compatíveis** com todos os métodos JSON-RPC do Solana, incluindo chamadas de leitura de dados (getBalance, getAccountInfo, etc.), envio de transações (sendTransaction), subscrição via WebSocket, entre outros [47].

Exemplo de Chamada RPC: Para consultar o saldo de 1 endereço, você pode fazer uma requisição JSON-RPC padrão. Por exemplo, usando `curl` para o método `getBalance`:

```
curl -X POST 'https://mainnet.helius-rpc.com/?api-key=SUA_API_KEY' \
  -H 'Content-Type: application/json' \
  -d '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "getBalance",
    "params": ["Gh9ZwEmdLJ8DscKNTkTqPbNwLUNA8Z5DiZpxU1x8Zt1 (exemplo)",
{"commitment": "confirmed"}]
  }'
```

Esta chamada retornaria um JSON contendo o saldo em lamports do endereço especificado, por exemplo:

```
{
  "jsonrpc": "2.0",
  "result": {
    "context": {"slot": 171950000},
    "value": 1234567890
  },
  "id": 1
}
```

Você pode usar qualquer biblioteca ou ferramenta compatível com Solana RPC apontando-a para o endpoint Helius. Por exemplo, em JavaScript com web3.js:

```
const connection = new Connection("https://mainnet.helius-rpc.com/?api-
key=SUA_API_KEY");
const balanceLamports = await connection.getBalance(new PublicKey(pubkey));
```

A chamada acima internamente aciona o mesmo método RPC, só que através da biblioteca.

Segurança em Front-ends: Se você estiver chamando a Helius diretamente de um aplicativo cliente (por exemplo, um front-end web), é arriscado expor sua API key publicamente. Para isso, a Helius

oferece **Secure RPC Endpoints**, que são URLs mascaradas e com rate-limit mais baixo, para uso seguro no front-end [48]. Essas URLs têm formato como `https://<subdomínio>.helius-rpc.com` (geradas no dashboard) e **não exigem passar a API key explicitamente**, pois ela já está vinculada ao subdomínio. Elas impõem um limite de ~5 requisições por segundo [48] e protegem seus créditos caso alguém descubra o endpoint (pois não consegue extrair sua key nem fazer flood além do limite baixo). Use essas URLs dedicadas no front-end ou, alternativamente, implemente um proxy/backend próprio para segurar sua chave.

Principais Métodos RPC e Diferenças de Comportamento

A Helius suporta **todos os métodos RPC oficiais do Solana**. Isso inclui: - *Métodos de conta e estado:* `getAccountInfo`, `getTokenAccountsByOwner`, `getProgramAccounts`, `getBalance`, `getBlockheight`, `getSupply`, etc. Qualquer consulta de dados on-chain funcionará da mesma forma que em um nó normal.
- *Métodos de transações:* `getTransaction` (para obter detalhes de uma transação via signature), `getConfirmedSignaturesForAddress2` / `getSignaturesForAddress` (para listar assinaturas de transações históricas envolvendo um endereço), `sendTransaction` (para enviar transações assinadas para a rede), `simulateTransaction` etc.
- *Métodos de slot e bloco:* `getSlot`, `getBlock`, `getBlockTime`, `getLatestBlockhash`, entre outros, para acompanhar o progresso da rede.
- *Subscrições via WebSocket:* `accountSubscribe`, `programSubscribe`, `logsSubscribe`, `signatureSubscribe`, `slotSubscribe`, etc., para receber updates em tempo real (detalhado adiante).

Em termos de **resultados retornados**, a Helius não altera o formato nem adiciona nada customizado – as respostas seguem exatamente o padrão do Solana. Isso significa que se você migrar de um endpoint RPC público ou próprio para a Helius, sua aplicação deve continuar funcionando, exceto mais rápido e com menos erros de rate-limit possivelmente.

Diferenças e Benefícios do RPC Helius: A principal diferença está na infraestrutura por trás. A Helius investe em otimizações que oferecem: - **Maior disponibilidade e throughput**: Os limites de requisições por segundo são mais generosos que nós públicos gratuitos. Por exemplo, no plano gratuito Helius você tem até 10 RPCs/segundo, e planos pagos vão de 50 até 500 req/s [49] [50] – bem acima do que um RPC público suportaria sem falhar. Isso é essencial para apps em escala.
- **Menor latência**: A arquitetura de nós Helius promete ~95% de redução de latência em respostas comparado à média [51]. Ou seja, consultas retornam muito rápido. Em prática, isso significa experiência de usuário mais fluida e menor tempo de espera para confirmar transações ou carregar dados.
- **Novos recursos**: integração nativa com serviços como **Priority Fee** e **Jito** para otimização de envio de transações (Helius Sender), e serviços de dados complementares (APIs de ativos, transações enriquecidas, etc.). Esses são extras além do RPC core.
- **Stake-Weighted QoS**: *Quality of Service por Peso de Stake.* Esse é um diferencial importante: O Solana implementa prioridade de rede baseada no stake delegado ao validador RPC. A Helius, sendo também uma operadora de validadores, utiliza conexões *staked* para enviar suas transações, o que aumenta significativamente a chance delas serem incluídas rapidamente nos próximos blocos mesmo em períodos de congestão [52] [53]. Em outras palavras, clientes Helius (especialmente planos pagos) têm suas transações roteadas via validadores de alto stake, obtendo **taxa de sucesso e velocidade de confirmação maiores** do que enviando através de um RPC não staked. (Explicamos mais sobre *Staked RPC* abaixo.)

RPC Padrão vs RPC com Conexão Staked

Originalmente, a Helius oferecia um endpoint separado chamado "staked RPC" (`staked.helius-rpc.com`), para que usuários enviassem transações através de uma conexão com validador com stake, garantindo prioridade. No entanto, **esse endpoint foi depreciado**, pois a Helius integrou a funcionalidade *stake-weighted* diretamente no endpoint padrão para todos os clientes pagos [54].

O que é um RPC Staked? Em Solana, validadores que têm muito SOL delegado possuem prioridade na propagação de transações durante congestionamentos (Stake-Weighted QoS). Assim, transações submetidas a nós RPC associados a validadores de alto stake tendem a "atropelar" transações enviadas a nós sem stake significativo, resultando em *landing rates* (probabilidade de entrar em um bloco) maiores e confirmações mais rápidas [55]. Helius, ao rodar um validador e serviço RPC, utiliza esse mecanismo para benefício dos usuários.

Na prática: Hoje, se você está em um plano pago (Developer, Business, Professional), **todas as suas transações enviadas via** `mainnet.helius-rpc.com` **já são automaticamente roteadas por conexões staked** [52] [56]. Você **não precisa fazer nada diferente no código** – o boost é transparente. Isso garante que durante picos de uso da rede, suas transações tenham prioridade e não sejam descartadas por limites de throughput. Para o desenvolvedor, o resultado é menos situações de `Transaction dropped` ou longos atrasos em confirmações, especialmente relevantes em aplicações financeiras e de trading.

Para usuários no plano gratuito, presume-se que as transações ainda utilizem conexões padrão (não staked). Ainda assim, elas são enviadas pelos robustos nodes da Helius – porém podem ficar atrás na fila se a rede estiver congestionada pelos validadores de stake alto. Considere fazer upgrade de plano se sua aplicação exige garantias de performance sob carga da rede.

Créditos e Custos: Anteriormente, usar o RPC staked custava 10 créditos por transação enviada, devido aos recursos extras consumidos. A Helius recentemente **reduziu esse custo para 1 crédito por envio** e unificou com o padrão [57] [58]. Ou seja, enviar uma transação (método `sendTransaction`) agora custa **1 crédito** em qualquer plano, igual a outras chamadas RPC comuns. Essa é uma melhoria significativa de custo, facilitando uso do serviço otimizado sem penalizar seus créditos.

Subscrições WebSocket Padrão

Além de chamadas HTTP, os endpoints Helius RPC suportam **WebSockets**, seguindo o protocolo padrão do Solana. Isso permite ao seu aplicativo **assinar eventos em tempo real** – por exemplo, acompanhar mudanças de saldo de uma conta, receber notificação quando uma determinada transação for confirmada, ou escutar logs de programas conforme são emitidos.

Os endpoints WebSocket correspondentes são: - **Mainnet WS:** `wss://mainnet.helius-rpc.com/?api-key=SUA_API_KEY` [59]
- **Devnet WS:** `wss://devnet.helius-rpc.com/?api-key=SUA_API_KEY` [59]

Qualquer cliente WebSocket pode conectar a essas URLs. Após conectado, use o formato JSON-RPC de subscrição do Solana: envie, por exemplo, um `{"jsonrpc":"2.0","id":1,"method":"accountSubscribe","params":["<address>",{"commitment":"confirmed"}]}` para começar a receber notificações de alterações naquela conta. A Helius suporta *todos os métodos de subscrição padrão*:
- `accountSubscribe` – notifica sempre que os dados de uma conta on-chain mudam (por qualquer

transação).
- `programSubscribe` – notifica quando há mudanças em contas pertencentes a um determinado programa (útil para ouvir eventos de um contrato específico).
- `signatureSubscribe` – notifica quando uma transação específica (por signature) é confirmada ou finalizada, servindo para saber em tempo real quando determinada tx concluiu.
- `logsSubscribe` – transmite logs de programas em execução (por exemplo, você pode filtrar logs de um programa específico por sua public key).
- `slotsSubscribe`, `slotsUpdatesSubscribe` – para acompanhar o progresso de slots e raiz de confiança.

Esses WebSockets **funcionam igual aos de qualquer RPC Solana**, mas com a robustez da infraestrutura Helius. Lembre-se de implementar *pings periódicos* para manter a conexão viva: a recomendação é enviar um ping a cada 30–60 segundos [60], pois conexões WebSocket inativas podem ser fechadas pelo servidor ou por intermediários após alguns minutos. Os websockets Helius requerem isso para garantir que o canal permaneça aberto e não seja derrubado por timeout.

Melhores Práticas para WebSockets padrão:
- Utilize bibliotecas oficiais quando possível. Por exemplo, usando web3.js: `connection.onAccountChange(publicKey, callback)` internamente lida com subscrição via WS. Essas libs geralmente já gerenciam reconexões e ping automáticos.
- Se implementar manualmente, monitore o evento de fechamento de conexão e re-conecte quando necessário. Reinscreva-se às assinaturas após reconectar (a Helius não mantém estado de assinaturas perdidas após um disconnect).
- Não abuse de subscrições desnecessárias. Cada assinatura consome recursos tanto do servidor quanto do seu cliente. Por exemplo, se precisa acompanhar 1000 contas, avalie se realmente necessita WebSocket em todas ou se consultas periódicas são suficientes em alguns casos. Os planos Helius não especificam um limite exato de assinaturas simultâneas, mas há limites de tráfego (50 req/s para subs criar? – não explicitado). Em geral, seja eficiente: feche subscriptions que não precisa mais, agregue monitoramento se possível (talvez monitorar um programa em vez de N contas individualmente, etc.).

Enhanced WebSockets (Geyser Feed)

Para casos que demandam **atualizações em tempo real ainda mais rápidas** e diretamente do mecanismo do validador, a Helius oferece os **Geyser Enhanced WebSockets**. Esses endpoints (codinome *Atlas*) conectam seu app a uma fonte de dados em tempo quase-real, com latências potencialmente menores que as do RPC padrão. As URLs são:
- **Mainnet Enhanced WS:** `wss://atlas-mainnet.helius-rpc.com/?api-key=SUA_API_KEY` [61]
- **Devnet Enhanced WS:** `wss://atlas-devnet.helius-rpc.com/?api-key=SUA_API_KEY` [61]

Características:
- Disponível apenas para planos **Business e Professional** (não acessível no plano Developer ou Free) [62]. Isso reflete o fato de ser um serviço premium, possivelmente limitado para garantir qualidade.
- Suporta atualmente **apenas dois tipos de subscrição**: `accountSubscribe` e `transactionSubscribe` [63]. - O `accountSubscribe` aqui deve funcionar similar ao padrão, mas com dados chegando diretamente do plugin Geyser do validador, potencialmente antes mesmo da transação ser finalizada (talvez já no momento em que é processada). Isso significa notificações de mudança quase instantâneas após inclusão no bloco.
- Já o `transactionSubscribe` é algo especial – ele provavelmente entrega notificações de novas transações em tempo real (talvez todas as transações ou filtradas por algum critério?). A documentação não detalha, mas possivelmente te inscreve a fluxos de transações do validador. Pode ser útil para casos

de monitoramento global de mempool/blocos. *(Nota: Solana não tinha um método de subscrição de transação no RPC padrão – parece ser exclusivo desse websocket enhanced, fornecendo eventos de transações em vez de esperar confirmações.)*
- Há um timeout de inatividade de **10 minutos** na conexão [63]. Ou seja, se você ficar 10 min sem enviar nada, a conexão pode ser fechada. Portanto, **é obrigatório enviar heartbeats frequentes (pings a cada ~60s)** para evitar timeouts (ainda mais agressivo que no WS padrão).
- O serviço está em **beta**. A própria Helius menciona que ocasionalmente pode haver pequenos delays ou comportamentos incipientes [64]. Ou seja, use com essa expectativa e reporte problemas se encontrar. Em geral, deve operar bem, mas talvez não tenha as mesmas garantias de estabilidade dos websockets padrão.
- **Not a Replacement for Dedicated Geyser (Yellowstone):** A Helius diferencia o Enhanced WS (Atlas) de outro produto chamado Yellowstone (gRPC de dados via node dedicado). O Atlas não requer você ter um node dedicado, mas entrega parte do benefício. Já o Yellowstone Geyser seria um stream full de dados da blockchain, disponível apenas via nodes dedicados. Mencionamos isso apenas para esclarecer que o Enhanced WS é um meio-termo: feed rápido sem necessidade de infra própria.

Quando usar Enhanced WS? Se sua aplicação exige detectar eventos on-chain **o mais rápido possível** – por exemplo, algoritmos de arbitragem, bots de trading que reagem a swaps ou mudanças de preço, sistemas de alerta de segurança que precisam notar movimentações suspeitas instantaneamente – então o Enhanced WS pode ser muito valioso. Ele reduzirá a latência entre a transação acontecer e você ser notificado. Para apps comuns (carteiras, jogos, etc.), o WebSocket padrão já é suficiente, pois a diferença de alguns segundos não é crítica.

Custo: O Enhanced WebSocket em si não consome créditos por mensagem, ele está incluso no serviço (limitado a planos maiores). Entretanto, há um custo indireto: ao usar esse feed intensivamente, você pode consumir mais banda (que, no caso de gRPC LaserStream, seria cobrado por MB – mas no WS não há tarifação explícita, diferente do LaserStream). O que existe é o limite de requisições por segundo do plano, mas subscrições WS não contam para o limite HTTP. Fique atento ao volume de dados – se você se inscrever em *transactionSubscribe* de Mainnet, será inundado por todas as transações (centenas por segundo), o que pode ser inviável processar do seu lado. Prefira filtros específicos ou use *accountSubscribe* focado nos pontos de interesse para não sobrecarregar.

Custos, Rate Limits e Boas Práticas RPC

Sistema de Créditos: Cada requisição RPC HTTP consome créditos da sua conta. A tabela geral de custos destaca que **a maioria das chamadas RPC custa 1 crédito cada** [65]. Isso inclui todos os métodos padrão não listados separadamente. Chamadas mais pesadas ou serviços especiais têm custo diferenciado, por exemplo: - `getProgramAccounts`: 10 créditos (pois pode retornar muitos dados) [44]. - Chamadas das APIs especiais (DAS, ZK, Enhanced Transactions) custam 10 ou 100 créditos (ver próxima seção para Enhanced TX) [44]. - Gerenciamento de webhooks: 100 cr (já citado) [44]. - Envio de transação com prioridade staked: 1 crédito (reduzido de 10) [66]. - Uso do **Helius Sender** (serviço de envio de tx ultra-rápido via block engine Jito): 0 crédito [67] – ou seja, gratuito além do custo normal da tx na rede. *(Sender é um recurso avançado: para traders que enviam bundles para maximizar chances, não abordaremos profundamente aqui.)*

Você recebe mensalmente uma cota de créditos conforme seu plano (Free: 1 milhão, Developer: 10M, Business: 100M, Professional: 200M) [49]. Monitorar o consumo é crucial.

Rate Limits: A Helius impõe limite de **requisições por segundo** para evitar abusos. Isso se aplica a chamadas HTTP RPC e chamadas das APIs Enhanced/DAS separadamente: - Plano Free: 10 RPC/segundo; 2 req/segundo para APIs aprimoradas (DAS/Enhanced) [49] [68]. - Developer: 50 RPC/s; 10 API-

spl/seg. - Business: 200 RPC/s; 50 API/s. - Professional: 500 RPC/s; 100 API/s. Esses limites são elevados e dificilmente atingidos por apps normais. Se ultrapassados, o servidor retornará HTTP 429 Too Many Requests [69]. Se isso ocorrer, implemente **retentativas exponenciais** ou espere um pouco antes de continuar, para voltar abaixo do limite. *Boas práticas:* distribua suas chamadas ao longo do tempo, evite picos súbitos (por exemplo, não dispare 1000 calls simultâneas na mesma fração de segundo; em vez disso, faça batches menores). Utilize caches locais quando possível para reduzir chamadas repetitivas (ex: se precisar checar o saldo de um mesmo conjunto de contas com frequência, considere armazenar o resultado por alguns segundos em vez de consultar a cada milissegundo).

Melhores Práticas Gerais para RPC:
- **Commitment apropriado:** Quase todas as chamadas de leitura aceitam parâmetro opcional de *commitment* ("processed", "confirmed", "finalized"). Use *finalized* para dados que devem ser definitivos (porém com potencial leve atraso). Use *confirmed* para feedback rápido em UX, sabendo que ainda pode mudar. *Processed* não é suportado em algumas APIs Helius (por ex., no histórico de txs enriquecidas é ignorado) [70]. Regra geral, *confirmed* é um bom padrão para equilíbrio entre velocidade e confiabilidade na maioria dos casos. - **Envio de Transações:** Ao usar `sendTransaction`, certifique-se de já ter obtido um `recentBlockhash` atualizado (use `getLatestBlockhash`) e assinado a transação corretamente antes de enviar. A Helius não altera o comportamento de validação – se a transação estiver mal formada ou com blockhash expirado, você receberá erros da mesma forma. Após enviar, você pode usar `signatureSubscribe` via WebSocket ou fazer polling de `getTransaction` para saber quando for confirmada. Lembre-se que, graças à conexão staked, suas transações enviadas via Helius têm grandes chances de serem rapidamente incluidas – mas ainda assim implemente um timeout/retry se necessário (ex: se não for confirmada em X segundos, reenvie ou avise o usuário).
- **Uso do Secure Frontend Endpoint:** Reforçando, nunca exponha diretamente sua API key em código front-end público. Use o endpoint protegido ou um proxy. A Helius facilita isso fornecendo aquele subdomínio com limite de 5TPS [48]. 5TPS costuma ser suficiente para interações de usuário, mas não para workloads intensos – então para operações pesadas faça-as via backend seguro.
- **Monitoramento e Resiliência:** Utilize o Status Page da Helius [71] para se informar de manutenções ou incidentes. Em caso de indisponibilidade, tenha um plano de contingência (por exemplo, fallback a outro RPC público temporariamente). Embora raro, nenhum serviço está 100% imune a falhas. Registrar logs das respostas RPC e tratar erros retornados (campos `error` nas respostas JSON-RPC) é importante para diagnosticar problemas.
- **Chamada Pesadas:** Evite chamar métodos muito custosos em alta frequência. Por exemplo, `getProgramAccounts` em um programa grande retorna milhares de contas – isso custa 10 créditos e pode demorar. Se precisar monitorar um programa, talvez seja melhor usar `programSubscribe` ou indexar as contas incrementalmemte ao invés de chamar repetidamente *getProgramAccounts*. Da mesma forma, se você precisar pegar um histórico de transações de um endereço, em vez de usar `getSignaturesForAddress` + `getTransaction` em loop (o que pode consumir muitos créditos), considere usar a **API de Transações Enriquecidas** descrita abaixo, que retorna histórico parseado de forma otimizada.

Enhanced Transactions – Transações Enriquecidas

A API de **Transações Enriquecidas da Helius** foi criada para simplificar o entendimento de atividades on-chain, fornecendo já o *parse* de transações complexas em um formato fácil de usar. Essencialmente, ela aplica o mesmo mecanismo de parsing dos webhooks enhanced, porém via consultas sob demanda. Isso é extremamente útil para **análise histórica** ou consulta pontual de transações sem precisar montar todo um indexador Solana manualmente.

Visão Geral

Sem a Enhanced Transactions API, desenvolvedores que quisessem interpretar transações Solana precisavam decodificar manualmente as instruções (geralmente em formato base58/base64) combinando com os ABIs de vários programas, o que é trabalhoso e propenso a erros. A Helius automatiza isso: você faz uma chamada passando uma ou várias transações (ou um endereço de usuário), e recebe de volta JSONs estruturados indicando **o que aconteceu, com quem, e quanto** em cada transação. Em resumo, a API extrai: **que tipo de ação foi realizada (transferência, swap, mint NFT, stake, etc.), quais contas estiveram envolvidas (quem mandou pra quem), valores de SOL/tokens movimentados, timestamps e outros metadados** [72]. Tudo isso sem você precisar lidar com bytes de instrução ou identificadores de programas.

Atualmente, a Enhanced Transactions API suporta parsing principalmente para **transações relacionadas a NFTs, tokens SPL e swaps via Jupiter** [73]. Essas cobrem boa parte das use-cases populares: negociação de NFTs, transferências de tokens fungíveis e operações DeFi comuns via agregador Jupiter. Transações de DeFi mais complexas (Serum, Orca direto, etc.) ou muito específicas podem **não** ser totalmente interpretadas – nesses casos, a API retorna resultados genéricos (`type: "UNKNOWN"` ou `type: "UNLABELED"`) com as instruções brutas, cabendo a você interpretar se necessário. A Helius declara que está trabalhando numa **V2 da Enhanced Transactions**, e que a versão atual (V1) não receberá atualizações de novos protocolos até lá [73]. Ou seja, tenha em mente as limitações e não confie cegamente para 100% dos programas.

Ainda assim, para **NFTs, SPL e Jupiter**, a Enhanced Transactions é extremamente útil. Por exemplo, ela reconhece transações de marketplaces (Magic Eden, Solanart, OpenSea etc.) identificando listagens, vendas, lances; reconhece operações SPL (transferências, burns, mints); e swaps de DEXes integrados no Jupiter (Orca, Raydium, Serum via Jupiter) identificando pares trocados e quantidades.

Consultando Transação(s) por Assinatura (Parse Transaction)

Você pode pedir o parse de **uma ou várias transações específicas** através do endpoint POST `/v0/transactions`. Essa chamada espera que você forneça no corpo da requisição as transações de interesse, seja na forma de **assinaturas (transaction signatures)** ou do **conteúdo raw** delas.

- **Por assinatura:** método mais comum – você tem o txid (signature base58) e quer os detalhes legíveis.
- **Por conteúdo bruto:** você pode fornecer a transação codificada (base64) se tiver capturado antes, e obtê-la parseada. Útil se quiser parsear uma transação que ainda não foi publicada on-chain, por exemplo, ou vinda de algum mempool.

Exemplo de requisição (por assinatura):

```
curl -X POST 'https://api.helius.xyz/v0/transactions/?api-key=SUA_API_KEY' \
  -H 'Content-Type: application/json' \
  -d '{
        "transactions": [

"5rfFLBUp5YPr6rC2g1KBBW8LGZBcZ8Lvs7gKAdgrBjmQvFf6EKkgc5cpAQUTwGxDJbNqtLYkjV5vS5zVK4tb6JtP"
        ]
      }'
```

No JSON, passamos um array `"transactions"` com uma ou mais strings. No exemplo é uma assinatura de transação. A API retornará um array de resultados, um para cada transação pedida, seguindo o mesmo formato descrito em "Estrutura de Evento Webhook Enhanced" anteriormente (campos `type`, `source`, `description`, `nativeTransfers`, `tokenTransfers`, `accountData`, `instructions`, `events`, etc.) [74] [75]. Basicamente, é como se estivéssemos recebendo aquele evento parseado, só que sob demanda e não em tempo real.

Exemplo de resposta (genérico):

```
[
  {
    "description": "Human readable interpretation of the transaction",
    "type": "TRANSFER",
    "source": "SYSTEM_PROGRAM",
    "fee": 5000,
    "feePayer": "8cRr...cs2Y",
    "signature": "yy5BT9be...to8DK",
    "slot": 148277128,
    "timestamp": 1656442333,
    "nativeTransfers": [
      {
        "fromUserAccount": "SenderPubKey...",
        "toUserAccount": "ReceiverPubKey...",
        "amount": 1234567
      }
    ],
    "tokenTransfers": [],
    "accountData": [
      {
        "account": "SenderPubKey...",
        "nativeBalanceChange": -1234567,
        "tokenBalanceChanges": []
      },
      {
        "account": "ReceiverPubKey...",
        "nativeBalanceChange": 1234567,
        "tokenBalanceChanges": []
      }
    ],
    "transactionError": null,
    "instructions": [
      {
        "accounts": ["SenderPubKey...", "ReceiverPubKey..."],
        "data": "3Bxs3zs...4XWo",
        "programId": "11111111111111111111111111111111"
      }
    ],
    "events": {}
```

```
  }
]
```

Acima simulamos a parse de uma transação simples de transferência de SOL (type `TRANSFER`). Repare que `events` veio vazio, pois não é uma transação de NFT, swap ou outro evento especial – apenas uma transferência normal, então não há subevento específico. Ainda assim, `type` categorizou corretamente como TRANSFER e temos os transfers nativos e mudanças de saldo listados claramente.

Agora, se a transação for algo como uma **venda de NFT ou swap**, o objeto de resultado incluirá aqueles campos específicos dentro de `events`. Por exemplo, se `type` for `NFT_BID` (um lance em NFT), poderíamos ter `events.nft` com buyer, seller, price, etc. Ou `type: "SWAP"` com `events.swap` detalhando tokens trocados. Esse formato é idêntico ao recebido via webhook.

Batch requests: Você pode passar múltiplas signatures no array `"transactions"` para obter resultados em batch em uma única chamada (economizando overhead e créditos). O tamanho máximo do batch não é explicitamente documentado, mas é comum usar até 50 ou 100 de uma vez. Use essa capacidade quando precisar parsear histórico extenso – por exemplo, se você tem 1000 txs, em vez de 1000 chamadas individuais, faça 10 chamadas com 100 cada. Isso será muito mais rápido e eficiente em créditos (cada chamada custará 100 créditos, total 1000, em vez de 1000 chamadas * 100 cr = 100k créditos; detalhes de custo a seguir).

Custo: Cada requisição ao endpoint de parse (**não** cada transação parseada, mas a chamada em si) custa **100 créditos** [76]. Portanto, aproveite para incluir o máximo de transações útil por chamada (respeitando limites razoáveis) em vez de fazer várias chamadas separadas de um em um. Por exemplo, parsear 1 transação custa 100 créditos, e parsear 50 transações de uma vez também custará 100 créditos (bem mais vantajoso). *Nota:* não há indicação de que o custo seja linear por transação; a documentação sugere um custo fixo por request. Portanto, use batches dentro do possível para obter melhor custo-benefício.

Consultando Histórico de Transações por Endereço

Além do parse por assinatura específica, a Helius oferece um endpoint **já pronto para histórico de um endereço**: o GET `/v0/addresses/{address}/transactions`. Esta chamada retorna uma lista das transações (já parseadas) envolvendo aquele endereço, até um limite e abrangendo tipos suportados.

Por exemplo, para obter as transações de um wallet público:

```
curl 'https://api.helius.xyz/v0/addresses/EnderecoPublico/transactions?api-key=SUA_API_KEY&limit=20&type=NFT_SALE'
```

Acima, buscamos as últimas 20 transações do endereço que sejam do tipo **NFT_SALE** (vendas de NFT) [77]. Você pode ajustar diversos **parâmetros de query** para filtrar e paginar os resultados:

- `limit`: número máximo de transações a retornar (1 a 100) [78]. Se não especificado, pode ter um default (geralmente 100). Limite grande retornará mais dados e usará mais créditos (custo aqui também é 100 créditos por requisição, independente do limite).
- `before`: uma signature (txid). Se fornecido, a busca *começa antes* **dessa transação na ordem cronológica. Em outras palavras, retorna txs** mais antigas que** a signature dada [79]. Use

para paginação: por exemplo, após pegar as últimas 100, pegue a signature da última do array e passe `before=<essa_signature>` para buscar as 100 anteriores.

- `until`: oposto de before – indica para parar a busca quando chegar nessa signature (inclusive ou exclusiva, conforme docs do Solana). Útil se você deseja resultados em um intervalo específico e já sabe a tx fim. Frequentemente `before` é suficiente para paginação simples [80].
- `commitment`: igual nas outras chamadas, pode ser `"confirmed"` ou `"finalized"` [81]. Default é `"finalized"`. "processed" não é suportado. Se você quer incluir txs muito recentes que ainda não finalizadas, use confirmed, mas sabendo que os resultados podem mudar caso role back (raro).
- `type`: filtra para retornar apenas transações de um certo tipo (mesmo conjunto de tipos que discutimos) [82]. Por exemplo, `type=SWAP` retorna só swaps envolvendo o endereço (se ele fez trades DeFi via Jupiter). Você pode chamar múltiplas vezes com diferentes tipos para segmentar atividades. Se omitir `type`, retorna todos os tipos suportados.
- `source`: filtra por fonte (protocolo) [17]. Por exemplo, `source=MAGIC_EDEN` retornaria apenas transações envolvendo Magic Eden (listagens, vendas, bids daquele marketplace) relacionadas ao endereço. A lista de fontes suportadas é grande – inclui nomes de muitos protocolos e programas (FORM_FUNCTION, EXCHANGE_ART, todas versões de Candy Machine, marketplaces NFT, protocolos DeFi populares, etc.) [83] [84]. Útil se você quer focar em atividades de um dApp específico.

A resposta desse GET será um array JSON, ordenado do mais recente para o mais antigo (padrão). Cada elemento do array é um objeto de transação parseada, no exato formato já descrito (campos type, source, etc.) [85] [86]. Ou seja, usar esse endpoint é semelhante a combinar `getSignaturesForAddress` + várias chamadas de parseTransaction, mas a Helius faz tudo de uma vez e devolve já os objetos interpretados.

Exemplo de uso prático: Suponha que você queira mostrar em uma interface todas as atividades de um certo wallet: envios de SOL, recebimentos, compras/vendas de NFT, swaps no DeFi, stakes, etc. Você pode simplesmente chamar `GET /addresses/wallet/transactions?limit=100` (e possivelmente paginações adicionais) e obter uma lista pronta para usar. Cada item diz o que foi a ação (`type`) e você pode renderizar textos amigáveis ou categorizá-los no frontend. Você pode também filtrar previamente: digamos que queira compor abas "DeFi Trades" e "NFT Activity" – pode chamar uma vez com `source=JUPITER` e outra com `type=NFT_*` para separar.

Paginação: Combine `limit` e `before` para paginação infinita. Ex: primeira chamada sem `before` pega últimas 100; próxima chamada use `before=<signature_da_última_recebida>` para pegar as 100 anteriores, e assim por diante, até esgotar (ou use `until` se tiver um ponto de parada). Lembre de que a API é *finalized by default*, então novas transações chegando no address após sua primeira chamada não aparecerão a menos que você refaça sem o before para pegar atualizações.

Custo: Assim como parse individual, cada chamada ao endpoint de endereço custa **100 créditos** (fixo) [76]. Portanto, pedir 10 transações ou 100 tem o mesmo custo. Aproveite o máximo (até 100) por chamada para eficiência. Se você precisar varrer um histórico muito grande, ainda serão múltiplas chamadas – por exemplo, para 1000 txs, 10 chamadas * 100 créditos = 1000 créditos. Avalie esse custo no seu orçamento de créditos.

Observações e Melhores Práticas com Enhanced Transactions:
- **Confiabilidade vs. Cobertura:** Lembre que transações que não se enquadram nos tipos suportados sairão como `UNKNOWN/UNLABELED`. Isso ocorre para algumas instruções DeFi ou contratos customizados. Nesses casos, os campos genéricos (transfers, accountData, instructions) estarão presentes, mas você não terá um `events.swap` bonitinho, por exemplo. Esteja preparado para tratar

esses casos ou ignorá-los. Por exemplo, se você está construindo um explorador de histórico, pode optar por exibir transações desconhecidas de forma mais técnica (mostrar o programId chamado e dizer "Interação com programa X", etc.).
- **Tipos e Fontes:** A lista de tipos e fontes suportadas é extensa. A Helius documenta mais de 100 tipos e dezenas de fontes (protocolos) mapeadas [11] [83]. Isso significa que quando a API reconhece a assinatura de um programa, ela já classifica adequadamente. Ex: uma interação com Stake Program vira `STAKE_SOL` ou `UNSTAKE_SOL` e assim por diante. Use essas categorizações ao seu favor para filtragem e apresentação.
- **Análise de Atividade:** Para tracking de atividade de usuários, esta API é uma mina de ouro. Você pode, por exemplo, rapidamente obter "quanto em tokens X o usuário movimentou no último mês" filtrando por aquele token mint nos resultados; ou "quantas NFTs ele comprou/vendeu" filtrando `type=NFT_SALE` vs `NFT_BUY` (existe `BUY` e `SELL` também, dependendo do contexto). Note que alguns tipos podem ser complementares – ex: `BUY` e `SELL` são provavelmente relativos a trades fungíveis; já NFT_SALE cobre ambos comprador e vendedor no mesmo evento (com buyer e seller distintos no objeto).
- **Performance:** Apesar de custar 100 créditos, as chamadas tendem a ser relativamente rápidas dado o volume retornado, mas não abuse solicitando histórico completo de contas enormes em um único momento repetidas vezes. Para endereços com milhares de transações, considere implementar um cache ou armazenar os resultados parseados em sua base de dados após a primeira chamada, atualizando incrementalmente conforme novas transações chegam (via webhook ou via consultas periódicas). Assim você não precisa chamar o mesmo histórico antigo várias vezes.
- **Integração com Webhooks:** Uma estratégia poderosa é usar **Webhooks + Enhanced Transactions em conjunto**. Por exemplo, seu sistema pode receber em tempo real via webhook eventos importantes (NFTs, swaps) e agir instantaneamente; em paralelo, regularmente usar o endpoint de endereço para ver histórico completo e reconciliar qualquer evento perdido ou para construir relatórios completos. Os identificadores (signature) permitem correlacionar: um evento recebido via webhook já está parseado, então você pode salvar; se alguma transação passou despercebida, a consulta de histórico a trará.
- **Limits e Erros:** Se você solicitar um endereço que nunca teve transações suportadas, provavelmente obterá um array vazio. Se pedir um endereço inválido, erro 400. Se sua API key não tiver créditos suficientes, erro 402/429 possivelmente. Trate esses cenários. Além disso, a API Enhanced Transactions V1 tem *limitações conhecidas* e possivelmente não lida bem com algumas coisas (por exemplo, não parseia transações *inner* de forma isolada se não se refletem em eventos). Sempre valide os resultados antes de usá-los cegamente.

Em resumo, a API de Transações Enriquecidas da Helius é uma ferramenta valiosa para desenvolvedores Solana, facilitando integrações complexas com muito menos esforço manual. Com webhooks para notificações em tempo real e RPC de alta performance para consultas e envio, a Helius oferece um **ecossistema completo** para construir aplicações robustas sobre Solana. Ao usar este guia, você deve conseguir criar e gerenciar webhooks filtrados, consumir a API RPC com eficiência (aproveitando staked nodes e websockets), e extrair insights de transações on-chain de forma rápida através dos endpoints de transações enriquecidas – tudo isso seguindo as melhores práticas de custos e confiabilidade.

Referências: Documentação oficial da Helius (Webhooks, RPC Nodes, Enhanced Transactions) [2] [52] [77], Políticas de créditos e limites [87] [65], e exemplos extraídos dos guias Helius para implementar essas funcionalidades [19] [61]. Boa integração e bom desenvolvimento!

---

1 2 3 4 5 6 7 8 9 15 16 18 19 20 21 22 23 24 25 26 27 28 29 32 33 34 35 36 37 38 39 40 41 42 87 Solana Webhooks: Real-Time Blockchain Event Notifications - Helius Docs
https://www.helius.dev/docs/webhooks

10 Webhooks API - Helius Docs
https://www.helius.dev/docs/api-reference/webhooks

11 12 Solana Webhook Transaction Types - Helius Docs
https://www.helius.dev/docs/webhooks/transaction-types

13 14 Create Webhook - Helius Docs
https://www.helius.dev/docs/api-reference/webhooks/create-webhook

17 70 78 79 80 81 82 83 84 85 86 Get Enhanced Transactions By Address - Helius Docs
https://www.helius.dev/docs/api-reference/enhanced-transactions/gettransactionsbyaddress

30 31 74 75 Get Enhanced Transactions - Helius Docs
https://www.helius.dev/docs/api-reference/enhanced-transactions/gettransactions

43 44 45 49 50 51 56 58 65 66 67 68 69 71 76 Helius Pricing Plans, Rate Limits & Credit System - Helius Docs
https://www.helius.dev/docs/billing/plans-and-rate-limits

46 47 48 52 53 54 57 59 60 61 62 63 64 Solana RPC URLs and Endpoints - Helius Docs
https://www.helius.dev/docs/api-reference/endpoints

55 What are the implications of Solana's stake-weighted mechanisms ...
https://solana.stackexchange.com/questions/13235/what-are-the-implications-of-solanas-stake-weighted-mechanisms-when-choosing-an

72 73 77 Solana Enhanced Transactions API - Helius Docs
https://www.helius.dev/docs/enhanced-transactions