ChatGPT

Jupiter – APIs de Dados (Token, Preço, Cotação, Lend, Trigger, Recorrente)

Jupiter oferece diversas APIs focadas em dados de mercado (em vez de execuções diretas de swaps), permitindo acessar informações de tokens, preços em USD, rotas de swap, dados de pools de lending, ordens de limite (trigger) e ordens recorrentes. Todas as APIs públicas possuem uma versão **Lite** (gratuita, sem necessidade de API key) e uma versão **Pro** (necessária API key, com limites de uso maiores) [1] [2]. Em geral, use os endpoints `https://lite-api.jup.ag/...` durante o desenvolvimento ou para baixo volume, e troque para `https://api.jup.ag/...` com uma API Key para produção ou alto volume (evitando atingir o limite de ~60 requisições/minuto no plano Lite) [2]. A seguir, detalhamos cada API de dados, seus endpoints, parâmetros, respostas esperadas, erros comuns e boas práticas de uso.

Token API v2 (Catálogo de Tokens)

A **Token API v2** fornece informações completas sobre tokens (mints) no ecossistema Solana, incluindo metadados, verificações e métricas de mercado. Ela disponibiliza **vários endpoints** para consultar tokens específicos ou listas de tokens por critérios como nome, símbolo, tags ou categorias [3]. O retorno consiste em uma lista de objetos de token com diversos campos úteis (nome, símbolo, ícone, decimais, suprimento, indicadores de auditoria, score orgânico, verificação, métricas de liquidez/atividade etc.) [4] [5]. **Nota:** A resposta da API pode evoluir conforme melhorias, portanto integradores devem lidar com possíveis novos campos [6].

GET `/tokens/v2/search`

Descrição: Busca tokens pelo **símbolo**, **nome** ou **endereço mint**. Este endpoint realiza a busca internamente e retorna tokens correspondentes ao termo de consulta fornecido [3] [7].

- **Parâmetros de entrada:**
- `query` (string, obrigatório): Termo de busca – pode ser o símbolo do token (ex: `"SOL"`), parte do nome, ou o endereço completo do mint. É possível passar **múltiplos valores separados por vírgula** (até 100 mints) para obter vários tokens de uma vez [8].

- `limit` (inteiro, opcional): Limita o número de resultados retornados. Por padrão, ao buscar por símbolo/nome, retorna até 20 resultados se houver muitas correspondências [8]. (Ao buscar por endereços específicos, o limite é o número de mints fornecidos, máx. 100.)

- **Resposta:** JSON contendo um **array de tokens** que correspondem à busca. Cada objeto de token inclui vários campos, por exemplo:

- `id`: endereço do mint do token.
- `name`, `symbol`: nome e símbolo do token.
- `icon`: URL do ícone.
- `decimals`: casas decimais do token.
- `circSupply` e `totalSupply`: suprimento circulante e total.

- `holderCount`: número de holders (detentores) do token [4].
- `isVerified`: booleano indicando se o token é verificado pela Jupiter [4].
- `firstPool.createdAt`: timestamp ISO da criação do primeiro pool de liquidez desse token (usado como referência de token recente).
- `audit`: objeto com auditoria do mint (ex.: flags se autoridade de mint/freeze foram revogadas) [4].
- `organicScore` e `organicScoreLabel`: pontuação orgânica calculada pela Jupiter indicando quão "saudável" é o token (baseada em distribuição de holders, volume orgânico etc.) [4].
- `tags`: lista de tags atribuídas (e.g. `"verified"`, `"lst"` etc.).
- `cexes`: lista de exchanges centralizadas que listam o token (se aplicável).
- **Métricas de mercado:** campos como `fdv` (Fully Diluted Value), `mcap` (Market Cap estimado), `usdPrice` (preço estimado em USD) e `liquidity` (valor total em liquidez nos AMMs) [5].
- **Estatísticas de atividade:** objetos `stats5m`, `stats1h`, `stats6h`, `stats24h` contendo métricas de variação nos últimos 5 minutos, 1h, 6h e 24h, respectivamente – incluem `priceChange`, `liquidityChange`, `volumeChange` (variação de preço, liquidez e volume), além de volumes de compra/venda orgânicos, número de trades, traders, compradores líquidos etc. nesses intervalos [9] [10].
- `updatedAt`: timestamp da última atualização destes dados.

Exemplo de resposta (parcial):


```
[
  {
    "id": "So11111111111111111111111111111111111111112",
    "name": "Wrapped SOL",
    "symbol": "SOL",
    "decimals": 9,
    "holderCount": 2342610,
    "organicScore": 98.9239,
    "isVerified": true,
    "fdv": 8.782449942922047e10,
    "mcap": 7.727535203779674e10,
    "usdPrice": 145.47114211747515,
    "liquidity": 89970631.83880953,
    "stats24h": {
      "priceChange": 1.5076,
      "volumeChange": -2.1516,
      "numBuys": 15125444,
      "numSells": 17582713,
      "numTraders": 754618
    },
    "tags": ["community","strict","verified"],
    "...": "..."
  }
]
```


- **Possíveis erros:** Consulta inválida ou muito genérica pode retornar erro **400** (por exemplo, nenhum termo fornecido). Se nenhum token corresponder, a resposta será um array vazio (não é

considerado erro). Tokens não suportados ou endereços inválidos também retornam array vazio ou erro 400 se a sintaxe da query estiver incorreta.

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/tokens/v2/search?query=SOL"
```

Este exemplo busca tokens relacionados a "SOL" (que retornaria, por exemplo, o SOL e tokens com "Sol" no nome). Para buscar por endereço específico, use o mint completo, e.g.:

```
curl "https://lite-api.jup.ag/tokens/v2/search?
query=So11111111111111111111111111111111111111112"
```

(No exemplo acima, foi usado o endereço do Wrapped SOL.)

GET `/tokens/v2/tag`

Descrição: Retorna todos os tokens que possuem uma determinada **tag** padrão. Útil para filtrar tokens por categoria especial, como tokens verificados ou Liquid Staking Tokens (LSTs) [11] .

- **Parâmetros de entrada:**
- `query` (string, obrigatório): Tag desejada. Somente dois valores são aceitos atualmente: `"verified"` (tokens verificados) ou `"lst"` (tokens de liquid staking) [12] .
- **Resposta:** Array de objetos de token (mesmo formato descrito anteriormente) contendo **todos** os tokens que possuem a tag informada [12] . *Ex:* `query=verified` retorna todos os tokens verificados no catálogo Jupiter.
- **Possíveis erros:** Fornecer uma tag não suportada retorna erro **400** ou uma lista vazia. Tags devem ser exatamente `"verified"` ou `"lst"` – valores diferentes resultarão em erro.
- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/tokens/v2/tag?query=verified"
```

Retorna a lista completa de mints marcados como verificados.

GET `/tokens/v2/{category}/{interval}`

Descrição: Retorna tokens destacados em **uma das categorias predefinidas** de mercado da Jupiter, que ajudam a identificar tendências. As categorias disponíveis são:
- `toporganicscore` – tokens com maior *organic score* (qualidade orgânica de volume/holders) [13]
- `toptraded` – tokens mais negociados (volume)
- `toptrending` – tokens em tendência de interesse/negociação crescente

Pode-se especificar também um **intervalo de tempo** para considerar nas métricas de ranking: `5m`, `1h`, `6h` ou `24h` (5 minutos, 1 hora, 6 horas ou 24 horas) [13]. Por exemplo, `/toptraded/24h` listará os tokens mais negociados nas últimas 24h.

- **Parâmetros de entrada:**
- `{category}` (string, obrigatório no caminho da URL): Uma das três categorias citadas acima.
- `{interval}` (string, obrigatório no caminho): Um dos intervalos suportados: `5m`, `1h`, `6h` ou `24h` [13].
- `limit` (inteiro, opcional): número máximo de tokens a retornar. Por padrão retorna 50 tokens, mas é possível ajustar para mais ou menos conforme necessário [14].

- **Resposta:** Array de objetos de token ordenados conforme a categoria e intervalo escolhidos. Os campos dos objetos são os mesmos já descritos (incluindo estatísticas correspondentes ao intervalo). Observação: tokens muito comuns como SOL, USDC etc. podem ser **omitidos** nessas listas para destacar outros ativos (já que esses blue chips geralmente estariam sempre no topo) [13].

- **Possíveis erros:** Se a categoria ou intervalo forem inválidos, retorna erro **404/400**. Por exemplo, usar um intervalo não suportado ou escrever errado a categoria resultará em erro de recurso não encontrado.

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/tokens/v2/toporganicscore/5m?limit=100"
```

Retorna até 100 tokens com maior score orgânico nos últimos 5 minutos [15].

GET `/tokens/v2/recent`

Descrição: Lista os **tokens adicionados recentemente** ao ecossistema (especificamente, tokens cujo primeiro pool de liquidez foi criado há pouco tempo) [16]. Essa lista permite exibir aos usuários ativos novos que acabaram de se tornar negociáveis.

- **Parâmetros de entrada:** Não há parâmetros requeridos além da própria rota. (Opcionalmente pode haver `limit` semelhante aos outros endpoints, não documentado explicitamente, mas por padrão retorna ~30 tokens recentes [17].)

- **Resposta:** Array de objetos de token dos tokens mais "recentes" em termos de criação de pool. Inclui os mesmos campos detalhados anteriormente. *Nota:* O critério de "RECENTE" refere-se **à data de criação do primeiro pool** de liquidez do token, **não** à data de cunhagem do token em si [18]. Assim, se um token existia há meses mas só teve pool agora, ele aparecerá nessa lista.

- **Possíveis erros:** Nenhum em particular além de problemas de rede – a chamada sempre retorna uma lista (vazia se não houver tokens novos, o que é improvável).

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/tokens/v2/recent"
```

Price API v3 (Preço USD Confiável)

A **Price API v3** fornece um **preço em USD "fonte de verdade"** para tokens, consolidado pela Jupiter. É uma API simples que, dada uma lista de mints, retorna seus preços atuais em dólares, calculados com base em transações reais (últimos swaps) e ajustados por heurísticas para evitar manipulações [19] [20]. A versão v3 substitui a v2 (agora deprecada) e retorna apenas o preço único e estável por token, eliminando campos ambíguos das versões anteriores. Deve ser usada como referência consistente de preço em todas as integrações Jupiter.

- **Endpoint:** GET `/price/v3?ids={mint1},{mint2},...` (Lite ou Pro).
- **Parâmetros de entrada:**
- `ids` (string, obrigatório): Lista de um ou mais endereços de mint **separados por vírgula** dos tokens que deseja o preço. Suporta até **50 tokens por requisição** [21]. Ex.: `ids=So111...112, JUPy...CN` para consultar SOL e JUP.
- **Resposta:** JSON com um objeto onde cada chave é o mint solicitado e o valor é um sub-objeto com informações de preço:
- `usdPrice`: preço estimado em dólar para o token.
- `decimals`: número de decimais do token (útil para apresentar preço formatado corretamente) [22].
- `blockId`: slot do blockchain Solana em que esse preço foi calculado (serve para verificar a recência) [23].
- `priceChange24h`: variação percentual de preço nas últimas 24h (se disponível).

Exemplo de resposta:

```
{
  "JUPyiwrYJFskUPiHa7hkeR8VUtAeFoSYbKedZNsDvCN": {
    "usdPrice": 0.4056,
    "blockId": 348004026,
    "decimals": 6,
    "priceChange24h": 0.5293
  },
  "So11111111111111111111111111111111111111112": {
    "usdPrice": 147.4789,
    "blockId": 348004023,
    "decimals": 9,
    "priceChange24h": 1.2907
  }
}
```

No exemplo, são retornados preços para JUP e SOL (WSOL).

- **Possíveis erros:** Se nenhum dos `ids` for fornecido ou estiver malformado, retorna erro **400**. Caso algum token não tenha preço disponível, ele **não aparecerá** no resultado (ou virá com valor `null`). É **comum** a API não retornar preço para tokens muito novos ou com pouca liquidez/atividade de trade recente (últimos ~7 dias) [24]. Nesses casos, significa que não há dados confiáveis de swap recentes ou que o token foi considerado **suspeito** pelos heurísticos de integridade (ex.: detecção de wash trading); tokens sinalizados como suspeitos no campo `audit.isSus` do Token API possivelmente não terão preço disponibilizado aqui [25]. A resposta bem-sucedida vem com código HTTP 200 mesmo se alguns tokens não tiverem preço (basta verificar se a chave do mint existe no JSON). Em caso de uso acima do limite (sem API Key no Lite ou excedendo quota), pode retornar **429 Too Many Requests**.

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/price/v3?
ids=JUPyiwrYJFskUPiHa7hkeR8VUtAeFoSYbKedZNsDvCN,So11111111111111111111111111111111111111112"
```

Consulta o preço do token JUP e do WSOL (Wrapped SOL) [26]. Recomenda-se limitar requisições em lote a no máximo 50 tokens por chamada.

Boas práticas: A Price API v3 é ideal para exibir preços atualizados de tokens em dashboards, carteiras e interfaces. Lembre-se que ela dá **apenas o preço** – para detalhes adicionais (volume, liquidez etc.) use a Token API ou a Swap Quote API. Em caso de ausência de preço para um token, evite usar preços de fontes desconhecidas; sinalize ao usuário que o preço não está disponível, pois isso indica falta de liquidez ou dados confiáveis para aquele asset [27]. Para alta frequência, utilize a versão Pro com API key, pois a Price API tem um bucket de rate limit separado nos planos Pro (para garantir throughput mesmo com muitos tokens) [28].

Swap API – Endpoint de Cotação (`/swap/v1/quote`)

O **Swap API** da Jupiter permite obter a **melhor cotação de swap** entre dois tokens, incluindo a rota ótima e estimativas de preço/quantidade, sem realizar a troca imediatamente. Aqui focamos no endpoint de **Quote** (cotação), que fornece dados de mercado e rota; a construção e envio de transações (endpoints `build` e `send`) não são abordadas, pois o foco são dados. O **endpoint de cotação** consulta o mecanismo de roteamento Jupiter (Metis v1) agregando liquidez de todos AMMs/DEXes em Solana para encontrar a melhor rota [29]. Ele retorna qual será o output estimado para uma quantia de input especificada, quais mercados serão usados (pode dividir entre AMMs se vantajoso) e informações como impacto no preço.

GET `/swap/v1/quote`

Descrição: Calcula a rota de swap ótima de um token de entrada para um token de saída, para uma quantidade desejada, sob uma tolerância de slippage especificada. Retorna o plano de rota com um ou múltiplos hops, incluindo quantidades estimadas de saída, taxas e outras informações [30] [31]. Este endpoint *não* executa a transação – apenas simula/calcula a melhor troca possível naquele momento.

- **Parâmetros de entrada (query):**
- `inputMint` (string, obrigatório): Endereço mint do token de **entrada** (que o usuário vai enviar/trocar). Ex.: `So111...11112` para SOL (WSOL) [32].

- `outputMint` (string, obrigatório): Endereço mint do token de **saída** (que o usuário deseja receber). Ex.: `EPjFW...wNYB` para USDC [32].
- `amount` (string, obrigatório): Quantidade do token de entrada a ser trocada, em unidade mínima (inteiro sem decimais). Ou seja, a quantidade em lamports para SOL ou em menor unidade para outros SPL. **Importante:** esse valor é o montante bruto sem considerar casas decimais. Ex.: para 1 SOL, `amount=1000000000` (1e9 lamports); para 5 USDC, `amount=5000000` (5e6, já que USDC tem 6 decimais).
- `slippageBps` (inteiro, recomendado): Tolerância de slippage em **basis points**. 1% = 100 bps. Define quanto a execução pode divergir do calculado antes de falhar. Ex.: `50` = 0,5% de slippage permitido [33]. Se omitido, pode ser considerado um padrão (geralmente 100 bps/1%). Na Jupiter UI, 0% (modo "Exact") significa não aceitar nenhuma derrapagem de preço [34].

- Parâmetros opcionais úteis:

    - `restrictIntermediateTokens` (boolean): Se `true`, **restringe rotas via tokens intermediários pouco líquidos**. Garante que a rota use apenas intermediários de alta liquidez (como SOL, USDC) para melhorar estabilidade e chance de sucesso [35]. Recomenda-se `true` em produção para evitar rotas exóticas suscetíveis a falha.
    - `onlyDirectRoutes` (boolean): Se `true`, **força usar apenas uma única etapa (swap direto)** sem hops intermediários [36]. Útil se quiser evitar rotas multi-hop. *Nota:* se não houver swap direto entre os tokens, nenhuma rota será retornada [37]. Mesmo se houver rota direta ilíquida, ele retornará essa rota (embora possa ser ruim) [38]. Use com cuidado – geralmente resulta em preço pior ou fracasso, por isso não é recomendável exceto em cenários específicos.
    - `maxAccounts` (inteiro): Limita o **número de accounts** utilizados na transação de swap [39]. Isso indiretamente limita complexidade da rota (ex.: impedir rotas com muitos hops ou pools que envolvam muitos accounts). O valor recomendado é 64; reduzir demais pode excluir DEXes e resultar em cotação subótima [40]. Só altere se enfrentar erro de transação grande demais, e reduza gradualmente conforme guia de **requote** [41].
    - `platformFeeBps` (inteiro) e `feeAccount` (string): Parâmetros para cobrar uma taxa do usuário (fee do integrador). `platformFeeBps` é a porcentagem em basis points sobre o montante de saída que será direcionada como fee, e `feeAccount` é o endereço da token account que receberá essa fee [42]. Deve-se ter uma token account de referência já configurada para o token de saída; caso contrário, a transação pode falhar. *(Obs.: A Jupiter não cobra fee própria pelo API, então integradores podem usar esse mecanismo para monetizar)*.
    - `asLegacyTransaction` (boolean): Por padrão, a Jupiter constrói transações no formato **Versioned Transaction** com Address Lookup Tables (ALTs) para incluir muitos accounts. Se o seu usuário estiver usando uma wallet que **não suporta transações versionadas**, ajuste `asLegacyTransaction=true` para forçar uma rota compatível (pode implicar rota mais simples ou com menos accounts) [43].

- **Resposta:** Em caso de sucesso (HTTP 200), retorna um objeto JSON com os seguintes campos principais:

- `inputMint` / `outputMint`: reiteram os mints de entrada e saída recebidos.
- `inAmount`: o valor de entrada fornecido (como string).
- `outAmount`: a **quantidade estimada de token de saída** que seria obtida com a rota ótima encontrada, *antes* de aplicar slippage [44]. Ou seja, é o melhor output possível no momento da cotação. *Importante:* esse valor **não desconta a slippage** – slippage é uma tolerância para execução, não um desconto antecipado [45].

- `otherAmountThreshold`: o valor mínimo de saída caso seja aplicada toda a slippage permitida. Se `swapMode` for `ExactIn`, esse geralmente é `outAmount * (1 - slippage)` arredondado para baixo – representando o **mínimo de saída** que será aceito para a transação não falhar [46]. (No modo inverso ExactOut, esse campo representaria o máximo de entrada permitido).
- `swapMode`: modo da cotação, normalmente `"ExactIn"` (indicando que a quantidade de entrada é fixa e calcula-se a saída). Poderia ser `"ExactOut"` se usarmos parâmetro alternativo para fixar saída e calcular entrada, mas essa API geralmente usa ExactIn.
- `slippageBps`: a slippage utilizada (retorna o mesmo valor enviado).
- `priceImpactPct`: impacto percentual no preço causado pela execução dessa rota com o tamanho especificado. Indica quanto o trade afeta o mercado (0% se desprezível) [47].
- `platformFee` ou `platformFeeBps`: pode retornar detalhes da fee do integrador se aplicada (ex.: conta de fee e bps). Se não usado, pode vir `null`.
- `routePlan`: **lista de etapas da rota** escolhida [48]. Cada elemento do array representa um swap em um DEX específico ou pool:
  - Dentro de cada item, há um objeto `swapInfo` com detalhes da etapa: `ammKey` (identificador do market/pool usado), `label` (nome/label do DEX, ex: "Orca", "Raydium CLMM", "Jupiter Routing" etc.), `inputMint` e `outputMint` daquela etapa (podem ser tokens intermediários em hops), `inAmount` e `outAmount` que fluem nessa perna, `feeAmount` e `feeMint` (taxa da plataforma AMM, se houver, e em qual token) [31].
  - O campo `percent` indica qual porcentagem do volume total do swap passou por essa etapa. No caso de rotas divididas (split between pools), a soma dos percents será 100, e cada item mostra a parte do swap. Se há apenas uma rota (não dividida), haverá um único item com `percent: 100` [31].
- `contextSlot`: o slot do bloco Solana no momento em que a cotação foi calculada (útil para conferir "frescura" dos dados).
- `timeTaken`: tempo em segundos que o backend levou para gerar a cotação (pode ajudar a monitorar performance; geralmente alguns milissegundos).

Exemplo de resposta:


```
{
  "inputMint": "So11111111111111111111111111111111111111112",
  "outputMint": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "inAmount": "100000000",
  "outAmount": "16198753",
  "otherAmountThreshold": "16117760",
  "swapMode": "ExactIn",
  "slippageBps": 50,
  "priceImpactPct": "0",
  "routePlan": [
    {
      "swapInfo": {
        "ammKey": "5BKxfWMbmYBAEWvyPZS9esPducUba9GqyMjtLCfbaqyF",
        "label": "Meteora DLMM",
        "inputMint": "So11111111111111111111111111111111111111112",
        "outputMint": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
        "inAmount": "100000000",
        "outAmount": "16198753",
        "feeAmount": "24825",
```

```
        "feeMint": "So11111111111111111111111111111111111111112"
      },
      "percent": 100
    }
  ],
  "contextSlot": 299283763,
  "timeTaken": 0.0153
}
```


No exemplo, está sendo cotado **100000000 lamports de SOL (~0.1 SOL)** para USDC com *slippage* de 0.5%. A melhor rota encontrada envolve uma única etapa (pool Meteora DLMM SOL/USDC), rendendo ~16.198.753 micro-USDC (16.19 USDC). `outAmount` é 16.198.753 e `otherAmountThreshold` de 16.117.760 indica que, com 0.5% de slippage, o mínimo aceitável de saída seria ~16.117.760 (16.117 USDC). Como `priceImpactPct` é 0%, este trade não afeta o preço perceptivelmente no pool.

- **Possíveis erros:** Diversas condições podem levar a erros (código HTTP 400 ou 500) na cotação:
- **Token não suportado/inválido:** se `inputMint` ou `outputMint` não fizerem parte do ecossistema Jupiter (ou forem inválidos), retorna erro indicando token não negociável (**TOKEN_NOT_TRADABLE** ou similar) [49] .
- **Quantidade inválida:** se `amount` for zero, negativo ou não número, erro 400. Quantidades muito pequenas podem não encontrar rota (ex.: valor menor que mínimos de pool) – pode retornar **NO_ROUTES_FOUND**.
- **Sem rota encontrada:** se não há liquidez/conexão entre os tokens, erro 400 com mensagem **NO_ROUTES_FOUND** ou **COULD_NOT_FIND_ANY_ROUTE** [50] . Isso pode ocorrer para pares exóticos ou se um token está sem pools ativos.
- **Slippage muito baixa:** se `slippageBps` = 0 e o preço oscilar mesmo minimamente entre a cotação e a execução, a transação pode falhar na execução (a cotação em si ainda será dada, mas a responsabilidade de executar dentro do preço exato é do integrador). Não chega a dar erro na *quote*, mas é um ponto a se notar para evitar falhas posteriores.
- **Rate limit excedido:** uso intenso sem API Key pode levar a erros 429.

Obs: O campo `routePlan` vazio (ou ausência de `outAmount`) indica falha em encontrar rota. Em erros com mensagem, a resposta pode vir no formato `{"error": "...", "code": ...}`. Por exemplo, `NO_ROUTES_FOUND` ou `ROUTE_PLAN_DOES_NOT_CONSUME_ALL_THE_AMOUNT` (quando a rota encontrada não consegue absorver todo o montante de entrada) [51] .

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/swap/v1/quote?
inputMint=So11111111111111111111111111111111111111112&outputMint=EPjFWdd5AufqSSqeM2qN1xzy
```

No exemplo, cotamos ~0.1 SOL para USDC, com 0.5% slippage e restringindo tokens intermediários a seguros [52] . Ajuste os parâmetros conforme o par desejado.

Boas práticas: Sempre valide se a resposta contém um `outAmount` antes de prosseguir para construir a transação. Se a rota vier vazia ou ocorrer **NO_ROUTES_FOUND**, informe o usuário que o swap não é possível (possivelmente falta de liquidez) [50] . Use `restrictIntermediateTokens=true` em produção para reduzir falhas, a não ser que deseje explicitamente rotas exóticas. Considere aplicar um slippage padrão (ex: 0.5%–1%) e permitir que usuários avançados ajustem para 0% (execução exata) ou valores maiores se entenderem o risco. Monitore `priceImpactPct` – trades com impacto muito

alto podem indicar pouca liquidez, avise o usuário sobre possível derrapagem grande. Para volumes grandes, talvez seja útil testar múltiplas cotas dividindo o montante (estratégia de **split** manual) ou usar `onlyDirectRoutes` para comparar a diferença. Em caso de erro de tamanho de transação (muitas accounts), utilize o parâmetro `maxAccounts` e o guia de **requote** para encontrar um trade-off [41]. E lembre-se: a cotação é válida no momento obtido; se demorar muito para usar (alguns segundos), os preços podem mudar – é sempre bom reconsultar perto do momento de envio.

Lend API (Beta) – Jupiter Earn (Depósito e Empréstimo)

A **Lend API** da Jupiter fornece acesso programático ao protocolo Jupiter Lend, que inclui o módulo **Earn** (empréstimo de ativos simples estilo depósito e rendimento) e futuramente o módulo **Borrow** (empréstimos colateralizados, ainda não lançado) [53] [54]. Aqui focaremos nos endpoints de dados relacionados a **Earn**, que permitem consultar informações das **"vaults" de earning** (onde usuários depositam tokens para render) e as **posições de usuários** (quanto cada usuário depositou, ganhos acumulados etc.). Também existem endpoints para criar transações de depósito/saque, mas esses envolvem execução e serão apenas mencionados brevemente.

Base URL: Endpoints iniciam com `/lend/v1/earn/...`. (A versão Borrow, quando lançada, provavelmente usará `/lend/v1/borrow/...`, mas no momento está marcada como *em breve*).
Assim como outras, há versão Lite e Pro (ex.: `lite-api.jup.ag/lend/v1/earn/tokens`). Todos requerem método HTTP **POST** para operações de transação (depósitos, saques) e **GET** para consultas de dados.

GET `/lend/v1/earn/tokens`

Descrição: Lista todas as "vaults" de depósito disponíveis no Jupiter Earn, com dados de cada ativo suportado. Cada "vault" corresponde a um token que pode ser depositado isoladamente (por ex.: SOL, USDC etc. cada um com sua pool de empréstimo) [55].

- **Parâmetros de entrada:** *Nenhum*. Basta fazer GET no endpoint.

- **Resposta:** Array JSON onde cada elemento é um objeto representando um token/vault disponível no protocolo Earn, com informações como:

- `asset` ou `mint`: o mint do token subjacente (ex.: SOL ou USDC).
- `symbol` e `name`: símbolo e nome do ativo.
- `supply` ou `vaultSupply`: quantidade total atualmente depositada na vault.
- `apy` ou `yield`: taxa de rendimento anual estimada para depósitos daquele ativo (pode estar embutida ou calculável a partir de outras infos, dependendo do design do protocolo).
- Parâmetros de configuração da vault: possivelmente LTV máximo, limite de retirada por bloco (debt ceiling automático), etc., se aplicável.
- Informações de liquidez: por ex., `availableLiquidity` (quanto disponível para retirada imediata) e outras métricas.

(O documento oficial menciona que esse endpoint retorna "token information such as the underlying token, supply, rates and liquidity information" [55]*, mas não detalha o schema completo. Integradores devem inspecionar a resposta real para ver todos os campos disponíveis.)*

- **Possíveis erros:** Nenhum específico – se o protocolo Lend estiver operacional, retorna 200 mesmo que a lista esteja vazia (por exemplo, se nenhum vault existir, o que é improvável). Em caso de falha interna, retorna erro 500.

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/lend/v1/earn/tokens"
```

Retorna um JSON com todas as vaults de Earn disponíveis e suas informações.

GET `/lend/v1/earn/positions?users={wallet1},{wallet2},...`

Descrição: Consulta as **posições de depósito** de um ou múltiplos usuários nas vaults Jupiter Earn. Permite obter quantos tokens cada usuário depositou e outros dados relacionados às suas contas de depósito.

- **Parâmetros de entrada:**

- `users` (string, obrigatório): Um ou mais endereços de **wallet** (chave pública) separados por vírgula, para os quais deseja buscar as posições [56]. Ex.: `users=4kY7...abc,8fjZ...def`. É possível consultar vários usuários de uma vez.

- **Resposta:** JSON contendo as posições correspondentes. A estrutura pode ser, por exemplo:

```
{
  "{wallet1}": [
     {
       "asset": "{mint do token depositado}",
       "shares": "...",
       "underlyingAmount": "...",
       "yieldEarned": "...",
       "allowance": "..."
     },
     { ... próximo depósito ... }
  ],
  "{wallet2}": [ ... ]
}
```

Onde para cada wallet solicitada, temos uma lista de depósitos (cada objeto representando um token depositado pelo usuário). Possíveis campos em cada posição:

- `shares`: quantidade de **shares** da vault que o usuário possui (representa sua parcela no pool de empréstimo).

- `underlyingAmount`: quantidade equivalente em tokens do ativo subjacente que essas shares representam no momento.
- `yieldEarned`: possivelmente o montante de rendimento acumulado (em token ou separado) até o momento – se o protocolo contabiliza ganhos acumulados.
- `allowance`: se aplicável, pode indicar quanto o usuário ainda pode retirar imediatamente (ou se há travas).
- Outros detalhes como timestamps ou status da posição.

Se um usuário não tiver posição em nenhuma vault, ele aparecerá com uma lista vazia ou não aparecerá no resultado.

- **Possíveis erros:** Fornecer uma chave pública inválida retorna erro 400. Se o parâmetro estiver faltando, também 400. Em casos normais, retorna 200 mesmo que nenhuma posição seja encontrada (array vazio para aquele user).

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/lend/v1/earn/positions?
users=FZz9...YourWalletAddress"
```

Retorna as posições de Earn do usuário especificado [56]. Você pode consultar várias wallets separando por vírgulas.

GET `/lend/v1/earn/earnings?user={wallet}&positions={positionId1},{positionId2},...`

Descrição: Retorna os **ganhos acumulados** (rewards) para posições específicas de um usuário. Esse endpoint é útil para calcular quanto cada depósito já rendeu em juros até o momento, por exemplo para mostrar "juros acumulados" para o usuário ou antes de realizar um saque.

- **Parâmetros de entrada:**
- `user` (string, obrigatório): O endereço de wallet do usuário dono das posições.
- `positions` (string, obrigatório): Um ou mais identificadores de posição, separados por vírgula [57]. Esses identificadores podem ser, por exemplo, o mint do token depositado ou um ID interno da posição. (Provavelmente o mint do token serve, assumindo uma posição por token por user. Se o usuário tiver múltiplos depósitos do mesmo token, talvez haja um ID específico.)

- **Resposta:** JSON indicando o montante de **recompensa/juros** ganho para cada posição solicitada. Formato possível:

```
{
  "{positionId1}": { "earned": "123456" },
  "{positionId2}": { "earned": "7890" }
}
```

onde os valores podem estar em menor unidade do token de depósito ou em outra unidade de reward. Como Jupiter Lend atualmente não menciona um token de reward separado, é provável

que o rendimento seja no próprio token depositado (incrementando o saldo). Assim, `earned` poderia ser 0 (se nada rendeu ainda) ou algum valor acumulado.

- **Possíveis erros:** Se o usuário não tiver aquela posição ou ID inválido, possivelmente retorna 400 ou simplesmente `earned: 0`. Parâmetros ausentes ou mal formatados resultam em erro 400.

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/lend/v1/earn/earnings?
user=FZz9...YourWallet&positions=So11111111111111111111111111111111111111112"
```

Consulta os ganhos da posição do usuário em SOL, por exemplo [57]. (No exemplo usamos o mint WSOL como identificador de posição.)

POST `/lend/v1/earn/deposit` (Execução)

(Endpoint de execução – criação de transação de depósito)
Este endpoint constrói uma **transação para depositar** uma quantia de um token na vault Earn correspondente. Ele **não realiza o depósito automaticamente**, mas retorna a transação pronta para o cliente assinar e enviar on-chain. Os parâmetros incluem: `asset` (mint do token a depositar), `amount` (quantidade mínima, inteiro, a depositar) e `signer` (address da wallet do usuário) [58]. Retorna um JSON com um campo `transaction` (base64) que deve ser assinado pela chave privada do `signer` e enviado a Solana. Se bem-sucedido, o depósito será efetivado e o usuário receberá shares da vault.
Possíveis erros: saldo insuficiente do usuário, vault cheia (se houver limites), ou parâmetros inválidos resultam em erro (ex.: mensagem `"insufficient funds"` ou similar).
Exemplo:

```
curl -X POST "https://lite-api.jup.ag/lend/v1/earn/deposit" \
  -H "Content-Type: application/json" \
  -d '{ "asset": "<mint>", "amount": "100000", "signer": "<userPublicKey>" }'
```

(Retorna `{"transaction":"BASE64...", "message": "...", ...}` *se sucesso.)*

POST `/lend/v1/earn/withdraw` (Execução)

Constrói transação para **sacar** uma quantia de tokens previamente depositados (queima as shares e retorna o subjacente ao usuário). Parâmetros: `asset` (mint do token da posição), `amount` (quantidade a sacar em menor unidade) e `signer` (wallet do usuário) [59]. Retorna transação base64 a assinar. Erros possíveis: sacar mais do que depositado ou além do disponível no momento (devido à mecânica de retirada gradual do protocolo – Jupiter Lend tem um "Automated Debt Ceiling" que limita retiradas instantâneas) [60].
Exemplo: (similar ao deposit, apenas mudando endpoint para `/withdraw` e possivelmente quantidade a retirar).

POST `/lend/v1/earn/mint` (Execução)

Alternativa ao `deposit`, onde em vez de especificar o *montante do ativo*, especifica-se quantas **shares** deseja receber. Parâmetros: `asset` (mint), `shares` (número de shares a mintar) e `signer`. A API calculará quanto do ativo subjacente é necessário depositar para obter aquela quantidade de shares e retorna a transação [61]. Útil para casos onde se quer atingir um certo tamanho de posição em shares. Funciona análogo a `/deposit`.

POST `/lend/v1/earn/redeem` (Execução)

Análogo ao `withdraw`, mas especificando quantas **shares** da vault queimar (redeeming) em vez de um montante exato do ativo. Parâmetros: `asset`, `shares`, `signer` [62]. Retorna transação para sacar aquele número de shares, convertendo-as em tokens de volta ao usuário.

(Os endpoints `mint` e `redeem` são basicamente dualidades de deposit/withdraw. Use o método que for mais conveniente para sua lógica: por montante de token ou por shares.)

POST `/lend/v1/earn/*-instructions` (Execução avançada)

Para cada uma das ações acima (deposit, withdraw, mint, redeem), existe uma variante terminando em `-instructions` que em vez de retornar uma transação completa, retorna a lista de **instruções (opcodes) não assinadas** a serem incluídas numa transação [63]. Isso permite composicionar a instrução de depósito/saque com outras no mesmo TX, ou utilizá-la via CPI em outro programa. O uso é similar: envie os mesmos campos em JSON para, por exemplo, `/deposit-instructions` e receba um JSON com `instructions` (array codificado em base64 e info de accounts) [64] [65]. Desenvolvedores avançados podem então desserializar essas instruções e montá-las manualmente numa `Transaction`. Se não quiser esse nível de controle, use os endpoints diretos que retornam a `transaction` completa.

Boas práticas (Lend API Earn): Ao fornecer opções de Earn aos usuários, sempre informe os **riscos** (Jupiter Lend é um protocolo novo, sujeito a riscos de smart contract e mercado) [66]. Não há taxas de protocolo, então os rendimentos são brutos para o usuário (integradores também não cobram taxa nesse caso). Lembre de usar um RPC confiável para enviar as transações geradas (o Jupiter não as envia automaticamente). Teste primeiro no ambiente devnet ou com pequenas quantias. Utilize o endpoint `/tokens` para exibir APYs atualizados e possibilitar ao usuário escolher onde depositar. Nas retiradas, atente que se o protocolo estiver no limite de liquidez, a retirada total pode não ser imediata – a **Automated Debt Ceiling** aumenta gradualmente a quantia retirável a cada bloco [60]. A API não expõe explicitamente esse dado de forma granular, mas se uma retirada grande falhar, pode ser necessário instruir o usuário a tentar novamente mais tarde ou retirar em partes. Sempre trate erros de transação (como insuficiência de fundos ou falha de assinatura) e mostre feedback ao usuário.

Trigger API – Ordens de Swap Programadas (Limit Orders)

A **Trigger API** permite criar **ordens limite de swap** on-chain: o usuário define um swap que será executado automaticamente apenas quando o preço atingir determinada condição. Essencialmente, uma “limit order” descentralizada entre quaisquer pares de tokens suportados pela Jupiter. A Jupiter cuida de monitorar os preços e executar a ordem no momento certo através de sua infraestrutura, garantindo melhor execução via roteamento aggregado quando o trigger dispara [67] [68].

Características principais incluem possibilidade de cobrar fees customizadas, suporte a qualquer par de tokens, melhor execução usando todos DEXes, monitoramento contínuo de preço e ordem com validade (expiração), além de opção de slippage customizada para priorizar execução (modo “Ultra”) ou preço exato (modo “Exact”) [69] [70] .

Os endpoints do Trigger API permitem **criar ordens**, **executá-las** (caso queira usar a infraestrutura Jupiter para assinar/enviar) e **cancelá-las**, bem como **listar ordens ativas ou históricas** de um usuário.

Base URL: `/trigger/v1/...` (Lite ou Pro). *Observação:* A Trigger API substitui um antigo caminho `/limit/v2` (agora deprecado). Certifique-se de usar os endpoints em `/trigger/v1` conforme descrito aqui [71] .

POST `/trigger/v1/createOrder`

Descrição: Cria uma **nova ordem trigger (limit order)**. Este endpoint gera a transação que coloca a ordem on-chain, mas não a envia automaticamente. Você fornece os parâmetros da ordem desejada e obtém de volta uma transação que deve ser assinada pelo usuário e enviada para efetivamente criar a ordem. Após criada, a ordem ficará pendente na cadeia, monitorada pelo serviço Jupiter até ser executada ou expirar.

- **Parâmetros de entrada (JSON no corpo do POST):** [72]
- `inputMint` (string, obrigatório): Mint do token que o usuário vai fornecer quando a ordem executar (token de entrada da swap futura).
- `outputMint` (string, obrigatório): Mint do token que o usuário quer receber quando a ordem executar (token de saída). Pode ser qualquer par, não precisa ter pool direto pois Jupiter roteará quando for executar.
- `maker` (string, obrigatório): Address (public key) da **wallet do usuário** que cria a ordem. Essa será a conta proprietária da ordem e que fornecerá os fundos no momento da execução (precisa ter saldo do input token).
- `payer` (string, obrigatório): Address que pagará a taxa de rede na criação da ordem. Geralmente é a mesma wallet do usuário (igual a `maker`), então pode repetir.
- `params` (objeto, obrigatório): Define os parâmetros da ordem em si:
  - `makingAmount` (string): Quantidade do token de entrada que o usuário deseja trocar (em unidades mínimas, sem decimais). Ex: `"1000000"` para 1 USDC se USDC tem 6 decimais. É o montante que será *vendido* quando a condição de preço for atendida [73] .
  - `takingAmount` (string): Quantidade do token de saída que o usuário deseja *receber* no mínimo. Esse parâmetro define efetivamente o preço limite. Por exemplo, se input = 1 SOL e output mínimo = 25 USDC, a ordem só executará quando 1 SOL >= 25 USDC. Ou seja, o usuário quer pelo menos 25 USDC por seu 1 SOL. `takingAmount` também é em unidade mínima do token de saída.
  - `slippageBps` (inteiro, opcional): Slippage para execução da ordem quando for disparada. Por padrão (ou se `0`), a Jupiter tratará como modo **Exact** (0 slippage, executa exatamente ao preço pedido) [74] [75] . Se o usuário quiser privilegiar chance de execução, pode definir um slippage maior (modo “Ultra”), assim quando o preço acionar, a ordem permitirá até X bps de slippage adicional para concretizar a troca [74] . (Na UI Jupiter, “Ultra” refere-se a Jupiter escolher um slippage; via API, você deve fornecer).
  - `expiredAt` (inteiro, opcional): Timestamp unix (segundos) de expiração da ordem. Após essa data, se não executada, a ordem torna-se inválida e pode ser cancelada ou removida. Se omitido, pode ficar aberta indefinidamente (ou até cancelamento manual). Recomenda-se definir um prazo razoável para evitar ordens zumbis.

    - `feeBps` (inteiro, opcional): Fee do integrador em basis points, se desejar cobrar do usuário além da fee da Jupiter. Se usar, é necessário especificar `feeAccount` (abaixo) correspondente.
- `computeUnitPrice` (string, opcional): Define o **prioritization fee** da transação *de criação* (não da execução). Use `"auto"` para a Jupiter definir automaticamente uma prioridade adequada (valor default) [76] . (Você também pode especificar um valor manual em micro-lamports/compute unit se quisesse customizar a prioridade de criação, mas normalmente não é necessário).
- `feeAccount` (string, opcional): Se `feeBps` foi fornecido, indique aqui a token account de *referral* do token de saída onde a fee será coletada [76] . Essa conta deve ser derivada do integrador e do token de saída (pode ser criada previamente ou dinamicamente conforme guia de fees [77]). Se omitido, nenhuma fee de integrador será cobrada.

- `wrapAndUnwrapSol` (boolean, opcional): Default `true`. Se verdadeiro, handle automático de wrapping/unwrapping de SOL nativo. Ou seja, se input ou output for SOL (nativo), Jupiter converterá para WSOL durante o processo. Em geral deixe `true` para facilidade, a não ser que queira gerenciar WSOL manualmente.

- **Resposta:** Em caso de sucesso (200), retorna um JSON com:

- `order`: endereço da conta de ordem criada (pubkey do PDA on-chain que representará a limit order) [78] . Você pode armazená-la para referencia futura (cancelamento ou consulta).
- `transaction`: string base64 da transação **não assinada** que irá criar a ordem [78] . Essa transação inclui instruções para transferir os fundos de `makingAmount` para a custodial account do order (ou program Jupiter) e registrar a ordem. **Importante:** Essa TX **deve ser assinada** pela wallet do usuário (`maker`) antes de ser enviada on-chain, caso contrário a ordem não será criada.
- `requestId`: um UUID associado a esta requisição [78] . Ele serve para correlacionar com a chamada de execução (ver `/execute` abaixo) se você optar por usar o serviço Jupiter para enviar a transação. Caso você mesmo vá enviar, o `requestId` não é tão crítico, mas guarde-o por garantia.

Exemplo de resposta (sucesso):

```
{
  "order": "CFG9Bmppz7eZbna96UizACJPYT3UgVgps3KkMNNo6P4k",
  "transaction": "AQAAAAAA...AA==",
  "requestId": "370100dd-1a85-421b-9278-27f0961ae5f4"
}
```

(Exemplo abreviado. `transaction` real seria uma string bem longa em base64.)
Exemplo de resposta (erro):

```
{
  "error": "invalid create order request",
  "cause": "input mint making amount must be at least 5 USD, received: 2",
  "code": 400
}
```

No exemplo de erro, a Jupiter rejeitou a criação porque o valor da ordem (~$2) ficou abaixo do mínimo de ~$5 USD exigido [79] . Há um limite mínimo para evitar spam de ordens muito pequenas.

- **Possíveis erros:**
- Requisição malformada ou faltando campos requeridos => erro **400** (com mensagem `"invalid create order request"` geralmente).
- Valor da ordem muito baixo => erro 400 com mensagem indicando o mínimo (como acima) [79] . A Jupiter impõe mínimo em USD (~$5 para pares estáveis, $10 para outros).
- Wallet do maker sem autorização ou assinatura faltando (se tentasse mandar sem assinar) => erro de assinatura/autoridade (mas note, aqui o endpoint só gera a TX; a assinatura é offline do serviço).

- Qualquer outra falha ao montar a TX retorna erro 500 com descrição.

- **Exemplo de chamada (criação de ordem):**

```
curl -X POST "https://api.jup.ag/trigger/v1/createOrder" \
  -H "Content-Type: application/json" \
  -d '{
        "inputMint": "So11111111111111111111111111111111111111112",
        "outputMint": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
        "maker": "<UserWalletAddress>",
        "payer": "<UserWalletAddress>",
        "params": {
            "makingAmount": "1000000",
            "takingAmount": "300000",
            "slippageBps": 0,
            "expiredAt": 1700000000
        },
        "computeUnitPrice": "auto"
      }'
```

Este exemplo criaria uma ordem para trocar 1 USDC (`makingAmount=1000000` em micro USDC) por pelo menos 0.3 SOL (`takingAmount=300000` lamports) – ou seja, efetivamente vender 1 USDC se 1 USDC >= 0.3 SOL (aprox preço SOL <= $3.33). Slippage 0 indica execução somente no preço exato. A ordem expira em timestamp 1700000000. O resultado será uma transação a ser assinada pelo usuário.

Nota: Após obter a transação, você pode optar por enviá-la por conta própria à rede **ou** usar o endpoint `/execute` da Jupiter, que faz o broadcast para você. De qualquer forma, a assinatura do usuário é necessária. A seguir explicamos o `/execute`.

POST `/trigger/v1/execute`

Descrição: Solicita que a Jupiter **envie uma transação previamente assinada** em nome do integrador. É usado tipicamente logo após o `/createOrder` (ou `/cancelOrder`) quando você prefere delegar o broadcast e acompanhamento para a infraestrutura Jupiter. Ao chamar `/execute`, a Jupiter receberá a TX assinada e a enviará à rede Solana, retornando o resultado (signature e status).

- **Parâmetros de entrada (JSON):** [80]

- `signedTransaction` (string, obrigatório): A transação em base64 já **assinada** pelo usuário (ou pelos devidos signatários) pronta para envio. Normalmente é a `transaction` retornada em `/createOrder`, após você desserializá-la e coletar a assinatura do usuário, então reserializar em base64.

- `requestId` (string, obrigatório): O mesmo `requestId` que foi retornado quando a ordem foi criada [81]. Isso vincula a execução à criação correspondente e ajuda a Jupiter a evitar duplicidade ou atraso no monitoramento.

- **Resposta:**

- Em caso de sucesso (ordem enviada e registrada on-chain), retorna um JSON com:

  - `signature`: o signature (hash) da transação enviada na rede Solana [82]. Você pode usar esse signature para verificar o status final no explorer ou via RPC se quiser.
  - `status`: `"Success"` indicando que a transação foi confirmada e a ordem agora existe on-chain [82]. *(Nota: Aqui "Success" significa que a TX de criação foi confirmada, não que a ordem executou – a ordem permanece aberta aguardando trigger.)*

- Em caso de falha (transação não foi confirmada ou foi rejeitada):

  - `signature`: ainda pode estar presente (pois pode ter sido enviada mas falhou).
  - `status`: `"Failed"` [83].
  - `error`: descrição do erro se disponível (por ex., *"custom program error code: 1"* ou mensagem de erro da runtime) [83].
  - (Código HTTP provavelmente 500 ou 400 dependendo da natureza da falha.)

Exemplo de resposta sucesso:

```
{ "signature": "5Y...Uz", "status": "Success" }
```

Exemplo de resposta falha:

```
{ "signature": "5Y...Uz", "status": "Failed", "error":
"custom program error: 1", "code": 500 }
```

- **Possíveis erros:**
- Transação mal assinada (signature inválida) ou já processada => erro 400/500 com mensagem de rejeição.
- `requestId` incorreto ou não correspondente => possivelmente erro 400 (invalid request).
- Falha na rede ou nos nós RPC => erro 500, podendo incluir mensagem de timeout.

- Caso a transação chegue a ser enviada mas seja revertida (ex: erro do programa de swap, falta de saldo do usuário no token input), o `status` será `"Failed"` com `error` detalhando (como no exemplo) [83].

- **Exemplo de chamada:** (considerando `signedTransaction` obtido previamente)

```
curl -X POST "https://api.jup.ag/trigger/v1/execute" \
  -H "Content-Type: application/json" \
  -d '{ "signedTransaction": "<base64String>", "requestId":
"370100dd-1a85-421b-9278-27f0961ae5f4" }'
```

Dica: você pode optar por **não usar** `/execute` e enviar a transação por conta própria via seu RPC Solana favorito [84]. Nesse caso, cuide de usar um RPC confiável e monitorar a confirmação. A Jupiter oferece `/execute` apenas como comodidade (ela usa seus RPCs otimizados, prioriza a TX com fee se necessário, etc. – útil se você quer garantir envio rápido).

POST `/trigger/v1/cancelOrder`

Descrição: Gera uma transação para **cancelar uma ordem ativa** do usuário. Cancela **uma ordem por vez** (para múltiplas, use `/cancelOrders`). A ordem deve pertencer ao usuário (maker) e estar aberta. O cancelamento efetivamente devolve os fundos presos na ordem de volta ao usuário e marca a ordem como cancelada on-chain.

- **Parâmetros de entrada (JSON):** [85]
- `maker` (string, obrigatório): O endereço da wallet do usuário que criou a ordem (proprietário).
- `order` (string, obrigatório): O endereço (pubkey) da ordem que se deseja cancelar [86]. Você obtém esse endereço quando criou a ordem (campo `order` retornado) ou via consulta.
- `computeUnitPrice` (string, opcional): Prioritization fee para a transação de cancelamento. `"auto"` normalmente, para auto-ajuste similar ao create. (Opcional; se quiser pode deixar auto ou definir manualmente se cancelamento for urgente.)

- **Resposta:**

- `transaction`: base64 da transação não assinada que, quando executada, cancelará a ordem (liberando fundos) [87].
- `requestId`: UUID para acompanhar essa solicitação (pode ser usado se for chamar `/execute` depois) [87].

Exemplo resposta sucesso:

```
{ "transaction": "AQAAAA....QYHAwUIX4Ht8Agx34Q=", "requestId":
"370100dd-1a85-421b-9278-27f0961ae5f4" }
```

Exemplo resposta erro:

```
{ "error": "no matching orders found", "code": 400 }
```

(Erro indica que o `order` passado não existe ou já foi executado/cancelado) [88].

- **Possíveis erros:**
- Order ID inválido ou não pertencente ao maker => erro 400 `"no matching orders found"` [88].

- Campos faltando => 400 pedido inválido.
- Se a ordem já estiver em processo de execução, o cancelamento poderá falhar (ex: se tentar cancelar exatamente quando a ordem acabou de executar). Nesse caso, provavelmente erro do programa on-chain ou transação irá falhar na hora do envio.

- *Obs:* Cancelar uma ordem incorre em taxas de rede (é uma transação Solana) e há limite de 5 ordens por transação em cancelamentos múltiplos.

- **Exemplo de chamada:**

```
curl -X POST "https://lite-api.jup.ag/trigger/v1/cancelOrder" \
  -H "Content-Type: application/json" \
  -d '{ "maker":"<UserWallet>", "order":"<OrderAddress>",
"computeUnitPrice":"auto" }'
```

Isso irá retornar a transação para cancelar a ordem especificada. Lembre de assinar (pelo usuário) e enviar ou usar `/execute` em seguida.

Dica: Para cancelar uma ordem, primeiro pode ser útil listar as ordens ativas do usuário via `/getTriggerOrders` para pegar o ID correto (veja abaixo) [89].

POST `/trigger/v1/cancelOrders`

Descrição: Semelhante a `/cancelOrder`, porém permite cancelar **várias ordens de uma vez** com menos chamadas. Você fornece uma lista de order IDs e ele gera transações (uma ou mais) que cancelam até 5 ordens cada (por limitação de tamanho de TX) [90].

- **Parâmetros de entrada:**
- `maker` (string, obrigatório): Wallet do usuário dono das ordens.
- `orders` (array de strings, opcional): Lista de até algumas dezenas de order IDs a cancelar [91].
- `computeUnitPrice` (string, opcional): Fee priority (auto ou custom).

Comportamento especial: Se **nenhuma ordem** for especificada em `orders`, o sistema interpretará como "cancelar todas as ordens abertas desse usuário" [90]. Nesse caso, ele automaticamente agrupa as ordens em lotes de 5 por transação. Útil para um “cancel all”.

- **Resposta:**
- `transactions`: array de strings base64, cada uma correspondendo a uma transação que cancela até 5 ordens [92]. Se menos de 6 ordens, geralmente retorna uma transação; se, por exemplo, 6 ordens, retornará 2 transações (uma com 5 cancelamentos, outra com 1).

- `requestId`: UUID para referência (pode ser reutilizado ao executar múltiplas TX; se usar `/execute`, deve chamar para cada TX ou passar o mesmo `requestId` em sequência) [93].

- **Possíveis erros:** Semelhante a cancelOrder – ordens inválidas ou já fechadas são ignoradas ou resultam em erro se nenhuma pôde ser cancelada. Se alguma ordem na lista não pertencer ao maker, essa específica não será cancelada (pode haver erro parcial). Em geral, se pelo menos uma transação é gerada, vem 200 OK.

- **Exemplo de chamada:**

```
curl -X POST "https://api.jup.ag/trigger/v1/cancelOrders" \
  -H "Content-Type: application/json" \
  -d '{ "maker": "<UserWallet>", "orders":
["OrderID1","OrderID2","OrderID3"] }'
```

Resposta pode ser:

```
{ "transactions": ["AQAAAA...34Q=", "..."], "requestId": "370100dd-...-
ae5f4" }
```

Lembre-se que precisará assinar **cada transação** listada e enviá-las. Se usar `/execute`, pode enviar uma por vez ou talvez enviar em paralelo (utilizando o mesmo requestId conforme documentação sugere) [93].

Notas sobre cancelamento múltiplo: Caso `orders` seja omitido, todas as ordens abertas serão incluídas automaticamente (atenção: se o usuário tiver muitas ordens, ele retornará várias transações; priorize para evitar expirar blockhash). As ordens são canceladas em grupos de 5 por TX por razões de tamanho [94]. Portanto, gerencie as transações retornadas adequadamente.

GET `/trigger/v1/getTriggerOrders?user={wallet}&orderStatus={status}&page={n}&inputMint={mint}&outputMint={mint}`

Descrição: Recupera a lista de **ordens trigger** de um usuário, podendo filtrar por status ativo ou histórico. Esse endpoint permite à interface mostrar quais ordens de limite o usuário tem abertas e quais já foram executadas ou canceladas.

- **Parâmetros de consulta (query):** [95] [96]
- `user` (string, obrigatório): O endereço da wallet do usuário cujas ordens deseja ver.
- `orderStatus` (string, obrigatório): Pode ser `"active"` para ordens abertas/pendentes, ou `"history"` para ordens concluídas (executadas ou canceladas) [97] [98].
- `page` (inteiro, opcional): Paginação dos resultados. A API retorna 10 ordens por página. Comece em `page=1`; se a resposta indicar `hasMoreData: true`, incremente para pegar próximas páginas. Se omitido, default page 1 [99].
- `inputMint` / `outputMint` (string, opcional): Filtra as ordens retornadas para apenas as que tenham aquele token de entrada ou saída [100]. Útil se quiser, por exemplo, exibir só ordens vendendo SOL, etc. Esses filtros são opcionais e podem ser usados combinados.

- *Outros:* A API V1 unificou openOrders e orderHistory neste endpoint. O formato de dados retornado é diferente da API antiga de limite v2, então adapte-se ao novo formato [101].

- **Resposta:** Um objeto JSON contendo possivelmente:

- `orders`: array de ordens (cada uma com detalhes). O formato de cada ordem inclui campos como: `orderId` (endereço da ordem), `inputMint`, `outputMint`, `remainingAmount` (quanto do input ainda resta – igual ao total inicial se não executada), possivelmente `minOutputAmount` (o critério takingAmount), `owner` (maker), status, e timestamps (criação, execução/cancelamento), etc. Pode também trazer informações de execução se history (como quanto foi realmente recebido, signature da swap executada, etc.).

- `currentPage`: número da página retornada, `totalPages` se disponível, e `hasMoreData` (boolean indicando se há mais páginas) [99].

- *Obs:* O conteúdo exato pode ser atualizado, mas basicamente fornece os dados necessários para listar as ordens do usuário e seus estados.

- **Possíveis erros:** Parâmetros faltantes (user ou status) => 400. Se o usuário não tiver ordens, retorna lista vazia (200 OK). Valores inválidos para status -> 400. Page fora do intervalo retorna vazio sem erro.

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/trigger/v1/getTriggerOrders?
user=FZz9...YourWallet&orderStatus=active"
```

Obtém as ordens ativas da wallet.
Para histórico (ordens executadas/canceladas):

```
curl "https://lite-api.jup.ag/trigger/v1/getTriggerOrders?
user=FZz9...YourWallet&orderStatus=history&page=1"
```

Boas práticas (Trigger API): Lembre de avisar usuários que **ordens trigger consomem saldo do token de input**: ao criar a ordem, o montante `makingAmount` sai da wallet e fica custodiado no programa até execução ou cancelamento. Portanto, exiba o status "locked" desses fundos e permita cancelamento fácil. Use `getTriggerOrders` para atualizar a UI em tempo real (por exemplo, após criar ou cancelar, refaça a consulta). Ao criar ordens, siga o mínimo de valor (~$5) para evitar erros [79]. Recomende sempre definir expiração (`expiredAt`) para evitar ordens esquecidas. Para execução, a infraestrutura Jupiter monitora preços – geralmente a execução ocorrerá próximo do preço limite exato; contudo, com slippage >0, pode executar um pouco antes de atingir exatamente o preço (para garantir fill). Integradores não precisam se preocupar em disparar nada – apenas crie a ordem e espere. Caso queiram, podem monitorar via websocket ou polling o momento em que uma ordem some de "active" (indicando execução ou cancelamento externo). As fees da Jupiter para ordens trigger são **0.03% para pares estáveis e 0.1% para demais** [102] (descontadas no momento da execução). Integradores podem adicionar sua fee via param `feeBps` se desejado [75]. Por fim, note as diferenças entre **modo Exact vs Ultra**: por padrão, slippage 0 (Exact) pode fazer com que algumas ordens não executem mesmo tocando o preço, se não houver liquidez exata naquele preço – nesse caso, oriente usar um slippage pequeno (Ultra) para aumentar chance de execução [70].

Recurring API – Ordens de Compra/Venda Recorrentes (DCA)

A **Recurring API** permite criar **ordens recorrentes automatizadas** no Solana, ou seja, agendar trocas de tokens que ocorrem repetidamente segundo um cronograma de tempo ou condições de preço. É ideal para implementações de **Dollar-Cost Averaging (DCA)** – ex: comprar $100 de SOL todo dia – ou estratégias de **value averaging** e automação de tesouraria [103] [104]. Dois tipos de ordem são suportados: - **Ordens baseadas em tempo:** executam em intervalos regulares (ex: diariamente, semanalmente). Opcionalmente podem ter parâmetros de preço mínimo/máximo para pausar se o preço sair de certo range.
- **Ordens baseadas em preço:** (Depreciadas) executam continuamente enquanto o preço do token

estiver dentro de certas condições, usando um pool de fundos do usuário que vai sendo gasto aos poucos conforme o mercado se move.

A Jupiter cuida de executar automaticamente usando seu roteamento quando as condições são satisfeitas, semelhante às trigger orders mas em vez de uma única execução, aqui podem ser **múltiplas execuções parceladas ao longo do tempo**.

Os endpoints são similares à Trigger API: criar ordem, executar transação de criação, cancelar ordem, e operações específicas de depositar/retirar fundos em ordens de preço, além de listar histórico.

Nota: A funcionalidade de ordens por preço via API está **deprecated**; a Jupiter recomenda usar as ordens de tempo para novas integrações, pois as de preço podem ser descontinuadas [105] [106] .

Base URL: `/recurring/v1/...`.

POST `/recurring/v1/createOrder`

Descrição: Cria uma **nova ordem recorrente** (seja de tempo ou de preço). O funcionamento é análogo ao createOrder de Trigger: recebe os parâmetros, retorna uma transação para ser assinada e enviada que registra a ordem on-chain. Dependendo dos campos em `params`, a ordem será interpretada como **Time-based** ou **Price-based** [105] .

- **Parâmetros de entrada (JSON):** [107] [108]
- `user` (string, obrigatório): Address da wallet do usuário criando a ordem (e que fornecerá os fundos iniciais).
- `inputMint` (string, obrigatório): Mint do token que o usuário **vai gastar** recorrentemente (ex: USDC se vai comprar uma quantia de SOL periodicamente).
- `outputMint` (string, obrigatório): Mint do token que o usuário **vai receber** em cada execução (ex: SOL no caso acima).
- `params` (objeto, obrigatório): Define os detalhes da ordem. Deve conter **ou** um objeto `time` **ou** um objeto `price`. A presença de um define o tipo:
  - **Caso `time` (Ordem por Tempo):**
  - `inAmount` (string): Montante total do token de input que o usuário deseja alocar para essa estratégia DCA (em unidades mínimas) [109] . Ex: `"104000000"` micro-USDC ($104) no exemplo.
  - `numberOfOrders` (inteiro): Número total de execuções que deverão ocorrer [110] . Ex: `2` ordens totais. Cada execução gastará `inAmount/numberOfOrders` de input.
  - `interval` (inteiro): Intervalo de tempo entre cada execução, em segundos [111] . Ex: `86400` (24 horas) significa executar diariamente.
  - `minPrice` (string ou null, opcional): Preço mínimo aceitável do par para executar. Se o preço de mercado (calculado pelo Jupiter Price API ou oráculos) estiver abaixo deste, a ordem pode pular/pausar execuções até voltar ao range. Se não deseja limitar por preço mínimo, passe `null`.
  - `maxPrice` (string ou null, opcional): Preço máximo aceitável para executar. Semelhante ao acima, mas se o preço estiver *acima* desse valor, não executa. Use `null` se não quiser teto de preço.
  - `startAt` (inteiro ou null, opcional): Timestamp Unix para quando iniciar a primeira execução [112] . Se `null`, começa imediatamente (ou no próximo intervalo calculado a

partir de agora). Pode usar isso para agendar a primeira compra para um horário específico.

Exemplo: Se `inAmount = 1000 USDC`, `numberOfOrders = 10`, `interval = 86400`, então será agendado para comprar $100 USDC em output token por dia, durante 10 dias [113]. O total de tempo será ~10 dias (10 * 86400) para concluir todas execuções. Durante a vigência, $100 sairá da conta a cada dia em troca do output token, respeitando minPrice/maxPrice se definidos. Quando terminar 10 execuções, a ordem conclui automaticamente.

- **Caso `price` (Ordem por Preço – *Depreciada*):**
- `depositAmount` (string): Montante inicial do token de input a depositar no contrato para essa estratégia [114]. Esse será o "caixa" do usuário para executar as ordens.
- `incrementUsdcValue` (string): Valor incrementado em USD a cada ciclo de compra/venda [115]. Explicando: ordens de preço buscam comprar mais do token de saída quando ele cai de preço e menos quando sobe, tentando manter um investimento fixo em USD por intervalo. Esse parâmetro define de quanto em USD variar o aporte por ciclo. (É complexo, e por isso a feature foi depreciada).
- `interval` (inteiro): Intervalo de tempo (segundos) entre checagens ou execuções [116]. Semelhante ao time-based, define uma frequência de avaliação do preço para possivelmente executar.
- `startAt` (inteiro ou null): Timestamp de início (ou null para imediato).

Sobre ordens de preço: Elas **não têm número definido de execuções** – rodam indefinidamente até serem canceladas, pois teoricamente o usuário colocou, digamos, 100 USDC e vai usando aquilo para comprar o token de saída quando o preço cai (comprando mais unidades) e menos quando sobe, etc., de forma aberta [117]. O usuário pode **depositar mais fundos** se acabar (usando endpoint de depósito de price order) ou **sacar** sobrante a qualquer momento (endpoint de withdraw), sem fechar a ordem [117]. A cada execução, se tokens de saída forem comprados, eles são **automaticamente sacados para a wallet do usuário** (não ficam retidos) [118]. Isso difere de ordens de tempo, onde o output poderia ficar custodiado até final? (Creio que não, no time-based possivelmente também depositam output na wallet a cada vez). No geral, price-based DCA era experimental e foi depreciado, então para novos integradores foca-se no time-based.

• Campos opcionais globais:

- (Não há slippageBps aqui – a Jupiter provavelmente usa um padrão interno ou considera sempre 0 slippage para cada swap programado, ou talvez aplique algum interno. A documentação não menciona slippage na Recurring API, então assume-se 0 ou automático.)

• `computeUnitPrice` (string, opcional): Semelhante às outras APIs, define fee de prioridade para criação da ordem. `"auto"` por default.

• *Não há* `feeBps` *aqui* – a Recurring API atualmente **não suporta integrator fees** (diferente da Trigger) [119].

• **Resposta:** Retorna:

• `transaction`: TX base64 não assinada que, quando enviada, criará a ordem recorrente [120].

- `requestId`: UUID para acompanhar.
  (Não retorna diretamente um `order` *ID aqui na resposta de criação, diferentemente da Trigger API. A identificação da ordem criada poderá ser obtida depois de executada a TX, ou pelo response do /execute ou via getRecurringOrders.)* [121]

Exemplo de resposta sucesso:

```
{ "requestId": "1d1f3586-eb72-4337-8c7e-1bbb9870ee4b", "transaction":
"AQAAAAA...AAA==" }
```

(Transaction abreviada; note que não há campo `order` listando o ID da ordem aqui. O ID será conhecido após a TX confirmar.) [120] [122]

Exemplo de resposta erro:

```
{ "code": 400, "error": "Order is valued at 2.99 USDC, minimum is 100.00
USDC", "status": "Bad Request" }
```

Indica que a ordem planejada envolvia valor muito baixo – no caso $2.99, abaixo do mínimo de $100 para ordens recorrentes [123]. A Jupiter impõe um mínimo relativamente alto para DCA (talvez para evitar txs frequentes de pouco valor).

- **Possíveis erros:**
- Valores mínimos não atendidos: conforme o exemplo, se a ordem (especialmente time-based) envolve menos do que ~$100 no total ou por ciclo, pode ser rejeitada (limite pode mudar, mas doc exemplifica 100 USDC) [123].
- Campos faltantes ou incoerentes: ex: passando ambos `time` e `price` em params, ou nenhum – erro 400.
- Se tentar `price` (deprecated) e o sistema eventualmente remover suporte, pode retornar erro indicando não suportado. Por ora, ainda funciona mas com warnings de deprecação.

- Qualquer erro ao montar a TX retorna code 500.

- **Exemplo de chamada:**
  Time-based DCA (por ex, comprar SOL diariamente com USDC):

```
curl -X POST "https://api.jup.ag/recurring/v1/createOrder" \
  -H "Content-Type: application/json" \
  -d '{
        "user": "<YourWallet>",
        "inputMint": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
        "outputMint": "So11111111111111111111111111111111111111112",
        "params": {
          "time": {
            "inAmount": "100000000",
            "numberOfOrders": 10,
            "interval": 604800,
            "minPrice": null,
```

```
            "maxPrice": null,
            "startAt": null
          }
        },
        "computeUnitPrice": "auto"
      }'
```

O exemplo acima agendaria gastar 100 USDC por semana (`interval` 604800 s = 7 dias) durante 10 semanas comprando SOL, sem limites de preço (minPrice/maxPrice null) e começando imediatamente. Cada semana ~10 USDC seriam usados (100/10), totalizando 100 USDC ao final.

Como funciona: Para time-based, a Jupiter vai dividir o montante igualmente por execuções [113]. O exemplo 100 USDC / 10 ordens => 10 USDC por semana. O usuário, ao criar, já deposita os 100 USDC inicial (inAmount) na ordem. A cada semana, 10 USDC serão trocados por SOL e depositados direto na wallet do usuário (ou na account de output?). Ao final das 10 execuções, a ordem se completa. Se o preço sair de algum limite (se tivesse min/max), a execução daquele intervalo é **pulada**; possivelmente a ordem pode terminar com execuções a menos ou esperar até voltar dentro do range. Para price-based (deprecated), a mecânica é diferente (ordem contínua até cancelar, depositAmount vai sendo usado conforme preço se move; incrementUsdcValue define a curva de compra/venda). *Para novos sistemas, foque em time-based.*

POST `/recurring/v1/execute`

Descrição: Semelhante ao execute da Trigger, este endpoint envia a transação assinada de criação/cancelamento de ordem recorrente para a rede via infraestrutura Jupiter.

- **Parâmetros de entrada:** [122]
- `signedTransaction` (string, obrigatório): Transação em base64 assinada pelo usuário (a transação retornada por `/createOrder` após assinatura).
- `requestId` (string, obrigatório): O `requestId` dado no createOrder correspondente [122].
- **Resposta:**
- `signature`: signature da transação enviada [124].
- `status`: `"Success"` ou `"Failed"` indicando se a TX foi confirmada [124] [125].
- `order`: se sucesso, traz o **ID da ordem recorrente criada** (endereço no programa JupiterRecurring) [124]. Agora você tem o identificador único da ordem para consultas futuras.
- `error`: em caso de falha, a mensagem de erro (e possivelmente `order: null`) [125].

Exemplo sucesso:

```
{ "signature": "3xyz...ABc", "status": "Success", "order":
"4DWzP4TdTsuwvYMaMWrRqzya4UTFKFoVjfUWNWh8zhzd", "error": null }
```

Exemplo falha:

```
{ "signature": "3xyz...ABc", "status": "Failed", "order": null, "error":
"Insufficient funds for the operation requested." }
```

(Falha indicou que provavelmente o usuário não tinha saldo suficiente do input token depositAmount/inAmount na hora de enviar) [125].

- **Possíveis erros:** Mesma lógica do Trigger execute. Assinatura inválida, requestId errado => erro imediatamente. TX enviada mas falha programa => status Failed + error. Falta de saldo do usuário na criação é um caso típico de falha (como no exemplo, fundos insuficientes) [125].

- **Exemplo de chamada:**

```
curl -X POST "https://lite-api.jup.ag/recurring/v1/execute" \
  -H "Content-Type: application/json" \
  -d '{ "signedTransaction": "<base64SignedTx>", "requestId": "<UUID-
from-create>" }'
```

(O uso é igual ao do Trigger – assine offline e então envie via Jupiter ou manualmente. Via Jupiter você ganha potencial priorização e confiabilidade de envio.)

POST `/recurring/v1/cancelOrder`

Descrição: Gera transação para **cancelar uma ordem recorrente** ativa. Somente uma ordem por vez (não há versão multi-cancel, já que ordens recorrentes não devem ser tão numerosas por usuário).

- **Parâmetros de entrada (JSON):** [126]
- `order` (string, obrigatório): ID da ordem recorrente a cancelar (obtido via listagem ou após criação/execução bem-sucedida).
- `user` (string, obrigatório): Wallet do usuário dono da ordem (criador).
- `recurringType` (string, obrigatório): `"time"` ou `"price"`, indicando o tipo da ordem que está cancelando [127]. Isso é necessário porque internamente as instruções diferem para cada tipo. *(Ordens de preço via API, lembre, estão depreciadas mas ainda canceláveis).*

- **Resposta:**

- `transaction`: base64 da TX de cancelamento (não assinada) [128].
- `requestId`: UUID para referência [128].

Exemplo sucesso:

```
{ "requestId": "36779346-ae51-41e9-97ce-8613c8c50553", "transaction":
"AQAAAAA...AAA==" }
```

Exemplo falha:

```
{ "code": 400, "error": "Failed to deserialize account data: failed to fill whole buffer", "status": "Bad Request" }
```

(Esse erro pode indicar que a ordem já não existe ou dados incorretos – ex: tipo errado ou ordem já cancelada) [129] .

- **Possíveis erros:**
- `order` não encontrado (ID errado ou já encerrado) => erro 400 "no matching order" ou similar.
- `recurringType` incorreto (ex: passar `"time"` pra uma ordem de price) => erro ao desserializar dados (como acima) [129] .
- Campos faltantes => 400.

- Não suportado: se tentar cancelar uma ordem de tipo deprecado e eventualmente não for mais possível, retornaria erro (por ora, contanto que forneça recurringType correto, deve cancelar).

- **Exemplo de chamada:**

```
curl -X POST "https://api.jup.ag/recurring/v1/cancelOrder" \
  -H "Content-Type: application/json" \
  -d '{ "order": "<OrderId>", "user": "<UserWallet>", "recurringType": "time" }'
```

Após gerar a transação, siga assinando e enviando (ou use `/execute`). Após confirmação, a ordem recorrente é removida e quaisquer fundos remanescentes depositados (no caso de ordens de tempo, se havia execuções pendentes, ou ordens de preço, saldo restante) serão devolvidos ao usuário.

POST `/recurring/v1/priceDeposit` (Depositar em ordem de preço)

Descrição: (Depreciado junto com ordens de preço) Gera transação para **adicionar fundos** a uma ordem recorrente do tipo price-based ativa. Caso o usuário tenha quase esgotado os fundos depositados inicialmente, pode usar este endpoint para colocar mais tokens de input na ordem sem precisar criar outra ordem.

- **Parâmetros:** [130]
- `order` (string, obrigatório): ID da ordem de preço a depositar.
- `user` (string, obrigatório): Wallet do usuário (dono da ordem).

- `amount` (string, obrigatório): Quantidade adicional do token de input a depositar (em unidades mínimas) [131] .

- **Resposta:**

- `transaction`: transação base64 de depósito gerada [132] .

- `requestId`: UUID da requisição.

- **Exemplo de uso:** digamos a ordem price-based do usuário estava usando USDC e ficou sem saldo após várias execuções; ele quer adicionar mais 10 USDC. Chamaria priceDeposit com esse valor.

- **Possíveis erros:** Ordem não encontrada ou não do tipo price => erro. Se passar ordem do tipo time, erro. Montante zero ou inválido => erro 400. Fora isso, semelhante aos anteriores.

(Como ordens por preço estão depreciadas, novos usuários provavelmente não precisarão usar isso. É suportado para compatibilidade com ordens antigas.)

POST `/recurring/v1/priceWithdraw` **(Retirar de ordem de preço)**

Descrição: Gera transação para **retirar fundos remanescentes** de uma ordem price-based sem fechá-la. O usuário pode querer resgatar parte ou todo o saldo não utilizado da ordem.

- **Parâmetros:** [133]
- `order` (string, obrigatório): ID da ordem price.
- `user` (string, obrigatório): Wallet do usuário.
- `inputOrOutput` (string, obrigatório): Indica qual moeda retirar – `"In"` para retirar o token de input (que ainda não foi gasto), ou `"Out"` para retirar algum token de saída acumulado (embora no design atual, token de saída é sempre retirado automaticamente a cada execução, então provavelmente este param existe por completude, mas geralmente você retiraria o que sobrou do input) [134].
- `amount` (string, opcional): Quantidade a retirar (unidades mínimas). Se omitido, **retira tudo disponível** [135] [136].

- **Resposta:**

- `transaction`: TX base64 de retirada gerada [137].
- `requestId`: UUID.

- **Possíveis erros:** Semelhante a deposit: ordem errada, param incorreto => erro. Se `amount` maior que disponível, provavelmente ainda gera TX para retirar tudo (ou erro se sistema não lidar). `inputOrOutput` deve ser "In" ou "Out" – qualquer outro valor => erro 400. Se tentar retirar output mas não há (porque é auto-retirado), possivelmente transação faz nada ou erro.

(Novamente, ordens price-based são legadas, integradores novos podem ignorar a menos que suportem usuários existentes com elas.)

GET `/recurring/v1/getRecurringOrders?user={wallet}&orderStatus={status}&recurringType={type}&page={n}&includeFailedTx={bool}`

Descrição: Lista as ordens recorrentes de um usuário, semelhante ao getTriggerOrders. Pode pegar ordens ativas (em andamento) ou históricas (concluídas/canceladas).

- **Parâmetros de consulta:** [138] [139]
- `user` (string, obrigatório): Wallet do usuário.

- `orderStatus` (string, obrigatório): `"active"` ou `"history"` para ordens em andamento vs concluídas [140].
- `recurringType` (string, obrigatório): `"time"` ou `"price"` – você deve especificar qual tipo de ordem quer listar. Não é possível listar ambos tipos simultaneamente, então se o usuário tiver ambos, chame duas vezes ou priorize um. (Note: price está depreciado, mas para histórico possivelmente terá valor) [138] [139].
- `page` (inteiro, opcional): Paginação, 10 ordens por página.
- `includeFailedTx` (boolean, opcional): Indica se deve incluir também tentativas de execução que falharam nas ordens listadas. Por padrão, `false`. Se `true`, o histórico pode mostrar execuções que não consumiram fundos por algum erro, para transparência. (Campo disponível para debug principalmente) [138].

- **Resposta:** Semelhante a Trigger: retorna um JSON com lista de ordens e indicadores de paginação. Cada ordem incluirá detalhes específicos:

- Para ordens time-based: provavelmente campos como total planejado (`totalInAmount`), quanto já executado, quantas execuções restam, intervalo, próximos horários agendados, etc., além de tokens e status.
- Para ordens price-based: saldo atual, increment value, etc.
- `status` pode indicar se está ativa, pausada por preço-limite, concluída, cancelada.

- Pode haver sublistas de execuções realizadas com timestamps e resultados (especialmente em histórico).

- **Possíveis erros:** Similar ao getTriggerOrders. Parâmetro faltante => 400. Se usuário não tem ordens do tipo/status, retorna lista vazia.

- **Exemplo de chamada:**

```
curl "https://lite-api.jup.ag/recurring/v1/getRecurringOrders?
user=FZz9...YourWallet&orderStatus=active&recurringType=time"
```

Obtém ordens recorrentes de tempo ativas do usuário.
Para histórico de ordens de preço (legado):

```
curl "https://lite-api.jup.ag/recurring/v1/getRecurringOrders?
user=FZz9...YourWallet&orderStatus=history&recurringType=price"
```

Boas práticas (Recurring API): As ordens recorrentes de tempo são excelentes para implementar DCA – sempre deixe claro ao usuário o compromisso total (ex: "Você está agendando investir 100 USDC ao longo de 10 semanas"). Apresente o cronograma (próxima execução em X, última em Y). Ofereça botão de **cancelar** para parar futuras execuções – use `/cancelOrder` para isso. Informe também que se o preço estiver fora dos limites definidos, a compra/venda do ciclo pode ser pulada (e talvez extendida? No design atual, creio que se pula, ele espera o próximo intervalo – possivelmente a ordem acaba não usando toda quantia se condições nunca voltarem ao range). Reforce que há uma taxa de 0.1% cobrada pela Jupiter por cada execução recorrente [119] (no momento integradores não podem acrescentar fee). Em termos de implementação, monitore o endpoint de listagem para atualizar status pós-execução. Por exemplo, quando uma execução ocorre, a ordem pode permanecer ativa mas com `numberOfOrders`

decrementado ou similar – usando `getRecurringOrders?status=active` você verá as mudanças (ou pode receber evento via webhook se disponível). Para ordens price-based, se optar dar suporte apesar de deprecação, informe que essa modalidade pode ser removida no futuro e incentive migração para time-based. E **importante**: aplique limites mínimos – a Jupiter impõe $100 mínimo para ordens de tempo [123], então evite chamadas abaixo disso. Se usuário tentar, mostre mensagem antes mesmo de enviar à API. Além disso, note que ordens recorrentes envolvem reentradas múltiplas – sempre verifique o `includeFailedTx` se precisar depurar por que alguma execução não aconteceu. Use `page` se usuário tiver muitas ordens (pouco provável no caso médio).

Autenticação e Rate Limits: Todos os endpoints acima (Trigger e Recurring) seguem o mesmo esquema de autenticação por API Key (para Pro tier) se quiser altas taxas. As transações criadas envolvem assinaturas do usuário, portanto **não requerem nenhuma assinatura do lado do servidor Jupiter** – a API Key serve apenas para controle de rate limit. Os limites de chamada são as padrão do Jupiter (60 rpm free, etc.) com janelas de 10s para Pro [2]. Criar/Cancelar ordens não deve ser feito em massa exagerada; se precisar criar muitas ordens, considere as implicações on-chain (cada ordem é uma account PDA ocupando rent, etc.).

Em resumo, a Jupiter oferece um conjunto robusto de APIs de dados de mercado que permitem desde montar interfaces de swap inteligentes, listar tokens com métricas ricas, consultar preços confiáveis, até recursos avançados como ordens limite e estratégias DCA automáticas. Ao integrá-las, siga as melhores práticas de uso de parâmetros, trate os erros comuns com mensagens claras ao usuário, e utilize API Key para volumes maiores para garantir performance [1] [141]. Com essas APIs, é possível construir agentes e aplicações DeFi que aproveitam a liquidez agregada de Solana de forma segura e eficiente.

Referências: Documentação oficial Jupiter (APIs de Token [3] [142], Preço [143], Swap Quote [32] [48], Lend [55] [56], Trigger [78] [87], Recurring [113] [124]). Cada seção acima citou trechos relevantes para confirmação dos comportamentos e limites descritos.

---

[1] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [142] V2
https://dev.jup.ag/docs/token-api/v2

[2] [28] [141] API Rate Limiting
https://dev.jup.ag/docs/api-rate-limit

[19] About Price API
https://dev.jup.ag/docs/price-api/

[20] [21] [22] [23] [24] [25] [26] [27] [143] Price API V3
https://dev.jup.ag/docs/price-api/v3

[29] [30] [31] [32] [33] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [52] Get Quote
https://dev.jup.ag/docs/swap-api/get-quote

[34] [67] [68] [69] [70] [74] [102] Trigger API
https://dev.jup.ag/docs/trigger-api/

[49] [50] [51] Common Errors
https://dev.jup.ag/docs/swap-api/common-errors

[53] [54] [60] [66] Lend API
https://dev.jup.ag/docs/lend-api/

55 56 57 58 59 61 62 63 64 65 Earn (Beta)
https://dev.jup.ag/docs/lend-api/earn

71 72 73 75 76 77 78 79 101 Create Order
https://dev.jup.ag/docs/trigger-api/create-order

80 81 82 83 84 Execute Order
https://dev.jup.ag/docs/trigger-api/execute-order

85 86 87 88 89 90 91 92 93 94 Cancel Order
https://dev.jup.ag/docs/trigger-api/cancel-order

95 96 97 98 99 100 Get Trigger Orders
https://dev.jup.ag/docs/trigger-api/get-trigger-orders

103 104 119 Recurring API
https://dev.jup.ag/docs/recurring-api/

105 106 107 108 109 110 111 112 113 114 115 116 117 118 120 123 Create Order
https://dev.jup.ag/docs/recurring-api/create-order

121 122 124 125 Execute Order
https://dev.jup.ag/docs/recurring-api/execute-order

126 127 128 129 Cancel Order
https://dev.jup.ag/docs/recurring-api/cancel-order

130 131 132 Deposit Price Order
https://dev.jup.ag/docs/recurring-api/deposit-price-order

133 134 135 136 137 Withdraw Price Order
https://dev.jup.ag/docs/recurring-api/withdraw-price-order

138 139 140 Get Recurring Orders
https://dev.jup.ag/docs/recurring-api/get-recurring-orders