Send Swap Transaction

Transaction sending can be very simple but optimizing for transaction landing can be challenging. This is critical in periods of network congestion when many users and especially bots are competing for block space to have their transactions processed.

IMPROVE TRANSACTION LANDING TIP

By using Jupiter Swap API, you can enable Dynamic Slippage, Priority Fee estimation and Compute Unit estimation, all supported on our backend and served directly to you through our API.

Let’s Get Started

In this guide, we will pick up from where **Get Quote** and **Build Swap Transaction** guide has left off.

If you have not set up your environment to use the necessary libraries, the RPC connection to the network and successfully get a quote from the Quote API, please start at Environment Setup or get quote.

Prepare Transaction

WHO IS THE SIGNER?

The most important part of this step is to sign the transaction. For the sake of the guide, you will be using the file system wallet you have set up to sign and send yourself.

However, for other production scenarios such as building your own program or app on top of the Swap API, you will need the user to be the signer which is often through a third party wallet provider, so do account for it.

In the previous guide, we are able to get the `swapTransaction` from the Swap API response. However, you will need to reformat it to sign and send the transaction, here are the formats to note of.

Formats	Description
Serialized Uint8array format	The correct format to send to the network.
Serialized base64 format	This is a text encoding of the Uint8array data, meant for transport like our Swap API or storage. You should not sign this directly.
Deserialized format	This is the human-readable, object-like format before serialization. This is the state you will sign the transaction.

Here's the code to deserialize and sign, then serialize.

1. `swapTransaction` from the Swap API is a serialized transaction in the **base64 format**.
2. Convert it to **Uint8array (binary buffer) format**.
3. Deserialize it to a **VersionedTransaction** object to sign.
4. Finally, convert it back to **Uint8array** format to send the transaction.

```
const transactionBase64 = swapResponse.swapTransaction
const transaction =
VersionedTransaction.deserialize(Buffer.from(transactionBase64,
'base64'));
console.log(transaction);

transaction.sign([wallet.payer]);

const transactionBinary = transaction.serialize();
console.log(transactionBinary);
```

BLOCKHASH VALIDITY

If you look at the response of `console.log(transaction);`, you can see that our backend has already handled the blockhash and last valid block height in your transaction.

The validity of a blockhash typically lasts for 150 slots, but you can manipulate this to reduce the validity of a transaction, resulting in faster failures which could be useful in certain scenarios.

Read more about transaction expiry here.

Send Transaction

Transaction Sending Options

Finally, there are a 2 transaction sending options that we should take note of. Depending on your use case, these options can make a big difference to you or your users. For example, if you are using the Swap API as a payment solution, setting higher `maxRetries` allows the transaction to have more retries as it is not as critical compared to a bot that needs to catch fast moving markets.

▸ **Transaction Sending Options**

```
const signature = await
connection.sendRawTransaction(transactionBinary, {
    maxRetries: 2,
    skipPreflight: true
});
```

Transaction Confirmation

In addition, after sending the transaction, it is always a best practice to check the transaction confirmation state, and if not, log the error for debugging or communicating with your users on your interface. Read more about transaction confirmation tips here.

```
const confirmation = await connection.confirmTransaction({signature,
"finalized");

if (confirmation.value.err) {
    throw new Error(`Transaction failed:
${JSON.stringify(confirmation.value.err)}\nhttps://solscan.io/tx/${s
} else console.log(`Transaction successful:
https://solscan.io/tx/${signature}/`);
```

Swap Transaction Executed!

If you have followed the guides step by step without missing a beat, your transaction *should* theoretically land and you can view the link in console log to see the transaction.

Oh? Transaction Not Landing?

As the Solana network grew and increased in activity over the years, it has become more challenging to land transactions. There are several factors that can drastically affect the success of your transaction:

- Setting competitive priority fee
- Setting accurate amount of compute units
- Managing slippage effectively
- Broadcasting transaction efficiently
- Other tips

How Jupiter Estimates Priority Fee?

You can pass in `prioritizationFeeLamports` to Swap API where our backend will estimate the Priority Fee for you.

We are using Triton's `getRecentPrioritizationFees` to estimate using the local fee market in writable accounts of the transaction (comparing to the global fee market), across the past 20 slots and categorizing them into different percentiles.

Read more about Priority Fee here.

Parameters	Description
`maxLamports`	A maximum cap applied if the estimated priority fee is too high. This is helpful when you have users using your

Parameters	Description
	application and can be a safety measure to prevent overpaying.
`global`	A boolean to choose between using a global or local fee market to estimate. If `global` is set to `false`, the estimation focuses on fees relevant to the **writable accounts** involved in the instruction.
`priorityLevel`	A setting to choose between the different percentile levels. Higher percentile will have better transaction landing but also incur higher fees. • `medium`: 25th percentile • `high`: 50th percentile • `veryHigh`: 75th percentile

```
const swapResponse = await (
  await fetch('https://lite-api.jup.ag/swap/v1/swap', {
      method: 'POST',
      headers: {
      'Content-Type': 'application/json'
      },
      body: JSON.stringify({
          quoteResponse,
          userPublicKey: wallet.publicKey.toBase58(),
          prioritizationFeeLamports: {
              priorityLevelWithMaxLamports: {
                  maxLamports: 10000000,
                  global: false,
                  priorityLevel: "veryHigh"
              }
          }
      })
  })
).json();
```

How Jupiter Estimates Compute Unit Limit?

You can pass in `dynamicComputeUnitLimit` to Swap API where our backend will estimate the Compute Unit Limit for you.

When `true`, it allows the transaction to utilize a dynamic compute unit rather than using incorrect compute units which can be detrimental to transaction prioritization. Additionally, the amount of compute unit used and the compute unit limit requested to be used are correlated to the amount of priority fees you pay.

Read more about Compute Budget, Compute Unit, etc here.

```
const swapTransaction = await (
  await fetch('https://lite-api.jup.ag/swap/v1/swap', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      quoteResponse,
      userPublicKey: wallet.publicKey.toBase58(),
      dynamicComputeUnitLimit: true
    })
  })
).json();
```

How Jupiter Estimates Slippage?

Apart from the static `slippageBps` parameter, Jupiter has iterated on different designs to estimate slippage better.

You can pass in `dynamicSlippage=true` to Swap API where our backend will estimate a slippage value by simulating the swap transaction closer to execution and calculate an optimal value based on the token category, historical swap's slippage data and other heuristics.

INFO

The Dynamic Slippage implementation on the Swap API is different from the Real Time Slippage Estimator (RTSE) on the Ultra API.

To use RTSE, you will need to use the Ultra API.

WARNING

To use Dynamic Slippage, you will need to pass in `dynamicSlippage=true` to both the `/swap/v1/quote` and `/swap/v1/swap` endpoints.

```
const quoteResponse = await (
  await fetch(
    'https://lite-api.jup.ag/swap/v1/quote?
inputMint=So11111111111111111111111111111111111111112&outputMint=EPjI
  )
).json();

const swapTransaction = await (
  await fetch('https://lite-api.jup.ag/swap/v1/swap', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      quoteResponse,
      userPublicKey: wallet.publicKey.toBase58(),
      dynamicSlippage: true,
    })
  })
).json();
```

How Jupiter Broadcast Transactions?

Transaction broadcasting is the process of submitting a signed transaction to the network so that validators can verify, process, and include it in a block.

Broadcasting Through RPCs

After you've built and signed your transaction, the signed transaction is serialized into a binary format and sent to the network via a Solana RPC node. The RPC node will verify and relay the transaction to the leader validator responsible for producing the next block.

Read more about how RPC nodes broadcast transactions.

This is the most typical method to send transactions to the network to get executed. It is simple but you need to make sure the transactions are:

- Send in the serialized transaction format.
- Use fresh blockhash and last valid blockheight.
- Use optimal amount of priority fees and compute unit limit.
- Free of error.
- Utilize retries.
- Configure your RPCs
  - Optional but you can send your transaction to a staked RPC endpoint also known as Stake-Weighted Quality of Service (SWQoS).
  - Used dedicated RPC services versus free or shared, depending on how critical your usage is.
  - Propagate to multiple RPC rather than reliant on one.

Broadcasting Through Jito

To include Jito Tips in your Swap transaction, you can do specify in the Swap API parameters. However, please take note of these when sending your transaction to Jito and you can find thsese information in their documentation:

- You need to submit to a Jito RPC endpoint for it to work.
- You need to send an appropriate amount of Jito Tip to be included to be processed.

MORE ABOUT JITO

You can leverage Jito to send transactions via tips for faster inclusion and better outcomes. Similar to Priority Fees, Jito Tips incentivize the inclusion of transaction bundles during block production, enhancing users' chances of securing critical transactions in competitive scenarios.

Additionally, Jito enables bundling transactions to ensure they execute together or not at all, helping protect against front-running and other MEV risks through “revert protection” if any part of the sequence fails, all while reducing transaction latency for timely execution.

Read more about how Jito works and other details here.

```
const swapTransaction = await (
  await fetch('https://lite-api.jup.ag/swap/v1/swap', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      quoteResponse,
      userPublicKey: wallet.publicKey.toBase58(),
      prioritizationFeeLamports: {
        jitoTipLamports: 1000000 // note that this is FIXED
LAMPORTS not a max cap
      }
    })
  })
).json();
```

Edit this page