All You Need to Know About Solana’s New DAS API

Overview

Retrieving NFTs and tokens on Solana is now simplified thanks to the introduction of the Digital Asset Standard (DAS) API. The DAS API, a recent addition to the Solana developers' toolkit, offers a unified interface for retrieving digital assets on Solana. Instead of dealing with multiple endpoints to interact with different asset types, developers can now leverage a single API to acquire the data they need for their applications.

This interactive guide will cover:

1. An understanding of the available asset types on Solana.
2. A comprehensive look at the methods provided by the DAS API.
3. Demonstrations of real use cases for each endpoint that are easily customizable.

This guide will allow you to follow along each use case, and by the end, you will be equipped to utilize DAS masterfully.

Prerequisites

- Node.js installed (v18.0 required to use built-in fetch)
- Helius RPC
- Basic knowledge of JavaScript

Environment Setup

1. Create a project folder named functions

.

1. For each example, create a new file within this folder.

Asset Types

In the Solana ecosystem, an "asset" can be any digital item of value, such as tokens or non-fungible tokens (NFTs), that exist on the blockchain. Solana supports a wide range of these assets. Understanding the type of data the DAS API returns when interacting with these assets is crucial.

Let's take a closer look at each asset type.

Non-Fungible

Non-Fungible assets follow the standard NFT model, storing metadata in a token account. This data resides in a Program Derived Address (PDA), an address owned by a program and not a specific user. They feature a Metadata PDA and a Master Edition PDA on the Solana blockchain.

Fungible

Fungible assets are SPL tokens with limited metadata. They can represent tokens like USDC or community/project tokens. A token will conform to the Fungible standard if its decimal value is greater than 0 during creation.

Fungible Asset

Fungible Assets represent items rather than individual units. They can hold more metadata than a standard Fungible asset. If the decimal is set to 0 during the creation of a Fungible standard item, it transforms into a Fungible Asset standard.

Programmable Non-Fungible

Programmable Non-Fungible assets mirror the Non-Fungible standard but remain in a frozen token account. This state prevents users from burning, locking, or transferring programmable assets without interacting with the Token Metadata program. This standard was a response to the royalty debate on Solana.

Methods Available

The DAS API offers a variety of methods tailored to different use cases, including:

1. getAsset

: Retrieve an asset by its ID.

1. searchAssets

: Locate assets using various parameters.

1. getAssetProof

: Obtain a merkle proof for a compressed asset by its ID.

1. getAssetsByGroup

: Acquire a list of assets by a group key and value.

1. getAssetsByOwner

: Retrieve a list of assets owned by an address.

1. getAssetsByCreator

: Get a list of assets created by an address.

1. getAssetsByAuthority

: Find a list of assets with a specific authority.

For detailed information about each method, refer to our

[documentation

](https://docs.helius.dev/compression-and-das-api/digital-asset-standard-das-api).

1. Get Asset

The getAsset

endpoint allows you to retrieve a specific asset by its ID. This ID can represent an on-chain token address or the ID on the merkle tree for compressed assets.

For more information, refer to the

[getAsset documentation

](https://docs.helius.dev/compression-and-das-api/digital-asset-standard-das-api/get-asset).

Example

Suppose we want to fetch the metadata of the Rank 1 Claynosaurz asset. In this case, we need to locate the asset's ID, set up our function to make the call to the DAS, and arrange our response to parse the exact object we need from the results.

Follow the steps below:

1. Start by setting up the function in a new file named “getAsset.js”:
2. Next, set up your fetch and await operations. Include the required ID parameter in the body of your request:

In this step, we're sending a POST request to the DAS API with the asset's unique ID in the request body.

1. Parse the results and display the metadata information:

This code snippet parses the API response into JSON format and logs the result to the console.

You can then run the script using the command node getAsset.js

.

Note: This method retrieves data for a single asset. If you need to search for a group of assets, the DAS API provides additional methods which we will discuss later.

Result

Executing this script will display the metadata of the specified asset:

This output provides detailed information about the asset, including the interface type, ID, metadata, creators, authorities, and ownership.

Full Code

Below is the complete code for fetching an asset using its ID:

Ensure to replace

with your actual API key.

In this code, we define an asynchronous function, getAsset

, which sends a POST request to the DAS API. We pass the asset's ID in the request body. Once the request completes, the function parses the response as JSON and prints the asset data to the console. Finally, we invoke the getAsset

function to execute this process.

2. Get Asset Proof

The getAssetProof

endpoint is used to retrieve an asset proof necessary for modifications to the compression program. Such modifications include actions like transfer, burn, update creator, update collection, and decompress of compressed assets.

For a detailed list of modifications using the asset proof, please refer to the

[documentation

](https://docs.helius.dev/compression-and-das-api/digital-asset-standard-das-api/get-asset-proof).

Example

To fetch the asset proof needed to modify a compressed asset, follow these steps:

1. Begin by setting up the function in a new file named “getAssetProof.js”:
2. Proceed with setting up your fetch and await. Include your required ID parameter in the body of your request:
3. Parse the result and extract the root:

Here, we are extracting the root from the asset proof, which can then be used to make additional modifications to the compressed asset.

Run node getAssetProof.js

in your terminal to get a return on the asset you set here.

Result

The output will include the asset proof information:

Full Code

Here is the complete code to fetch an asset proof using its ID:

Make sure to replace

with your actual API key.

In this script, we define an asynchronous function, getAssetProof

, which sends a POST request to the Helius API. We pass the asset's ID in the request body. Once the request is complete, the function parses the response into JSON and prints the asset proof and its root to the console. Finally, we call the getAssetProof

function to execute this process.

3. Search Assets

The searchAssets

method retrieves digital assets based on specified search parameters, providing a flexible approach to fetch data. It allows users to customize the search, thus providing a more detailed control over the assets returned.

Detailed parameters are available in the

[searchAssets documentation

](https://docs.helius.dev/compression-and-das-api/digital-asset-standard-das-api/search-assets).

Example

Let's go through an example where we want to display an assets image and name from a user wallet, that belongs to the Drip Haus collection, and only display the compressed items within it. For brevity of this tutorial, we will post this in a JSON file with the name and image of each Drip asset in the example wallet.

1. Begin by creating a new file named searchAssets.js

with the following asynchronous function:

1. Implement the function to send a POST request with the specified search parameters. In this case we are using the compressed, owner address, and collection grouping parameters to achieve our desired result:
2. Parse the response, group assets by their ID, and handle potential duplicates by detailing the exact items we need from the response of each item:
3. The next function is finding any repeats here and removing them from our list, you can decide to not use this function if you want to show repeat assets. This will also add to a new asset group when it is determined it's not a repeat:
4. Save the search results into a JSON file named searchResults.json

:

To execute the script, run node searchAssets.js

. This will populate the searchResults.json

file with the search results.

Result

Full Code

Make sure to replace

with your actual API key.

In this script, we define an asynchronous function, searchAssets

, which sends a POST request. It uses various search parameters in the request body, such as compression, owner address, and collection ID. Once the request is complete, the function parses the response into JSON and extracts the relevant asset information (ID, name, JSON URI). It groups these assets by ID and saves this organized data into a JSON file named searchResults.json

. Finally, we call the searchAssets

function to initiate this process.

4. Get Assets by Owner

The getAssetsByOwner

endpoint provides a list of digital assets owned by a specific address. This is currently the fastest way to retrieve specific ownership information for digital assets using Helius.

Example

To fetch the assets owned by a certain address, follow these steps:

1. Begin by setting up the function in a new file named “getAssetsByOwner.js”:
2. Set up your fetch and await, and include your parameters in the body of your request, for this endpoint we are using ownerAddress

:

1. Then, parse your response and extract the necessary asset information to post to the JSON. We will use the asset ID, name, and json URI to group to the wallet we are searching:
2. We can now set up our function to post to a JSON:

Result

The output will include the digital assets owned by the specified address:

Full Code

Here is the complete code to fetch the assets owned by a specific address:

Make sure to replace

with your actual API key.

In this script, we define an asynchronous function, getAssetsByOwner

, which sends a POST request to the Helius API. We pass the owner's address in the request body. Once the request is complete, the function parses the response into JSON and prints the assets owned by the specified address to the console. Finally, we call the getAssetsByOwner

function to execute this process.

5. Get Assets by Group

The getAssetsByGroup

endpoint is used to retrieve digital assets associated with a specific collection ID. This endpoint is crucial when you need to fetch items specific to a collection or require a token-gated dApp to be associated with a certain on-chain collection.

Example

In this scenario, we are going to set up a collection snapshot for SMB’s. It's important to parse the response to extract the asset owner from the ownership object, as detailed in our documentation. The objective is to filter for owners of multiple SMBs, and only show them once in our JSON, effectively creating a “snapshot”.

1. Start by setting up the function in a new file named “getAssetsByGroup.js”. Make use of a Set

to ensure we only store unique owners:

1. Proceed by setting up your fetch and await. Set your parameters listed above in the body of your request. Set the groupKey and groupValue in the request. Initialize page

to 1 and hasMoreResults

to true to set up for pagination. This will later set up a pagination function to return false if the results for page are less than 1000 (the max limit per request):

1. Parse the results to extract the owner string needed to create the snapshot of the current owners:
2. Set up pagination by increasing the page

parameter if there are 1000 results. If there are fewer, set hasMoreResults

to false to stop the pagination:

1. Now we can set our owners to be an array, and setup our root value to post to a JSON with the number of holders, and each owner wallet that is unique:
2. Convert the Set

of unique owners to an array. Set up the root

value to post to a JSON with the number of holders and each unique owner wallet:

You can then run the command node getAssetsByGroup.js

to populate the "ownerResults.json" file with your results.

Result

Your result would be a JSON equivalent of a holder snapshot, with the count of unique owners and a list of all unique owners:

Full Code

Make sure to replace

with your actual API key.

You've now learned how to retrieve digital assets associated with a specific collection ID using the getAssetsByGroup

endpoint. By effectively using the pagination and Set data structure to ensure unique owner entries, you can create a snapshot of current holders of any collection with an on-chain ID.

6. Get Assets by Creator

The getAssetsByCreator

endpoint is used to retrieve assets created by a specific public key address. This endpoint is useful when you want to find assets related to a specific artist or project on Solana.

Example

For this example, we are going to return assets created by [Zen0

](https://exchange.art/zen0/nfts). We can use the creator address and set the onlyVerified

parameter to true to only retrieve assets created by a verified wallet. We will parse the results to display each asset's ID and its owner.

1. Start by setting up the function in a new file named "getAssetsByCreator.js". We will use the fs

module to post our results to a JSON file:

1. Now, we can set up the fetch request and await

the response. Set the necessary parameters in the body of the request, including the creatorAddress

and onlyVerified

:

Don't forget to replace

with the actual address of the creator whose assets you wish to retrieve.

1. Parse the response to extract the result and store it in an array:
2. Group the assets based on the ownership ID (since a single person can own multiple assets), and display the assets owned by each ID. If a new owner is detected, a new group will be created in the JSON file:
3. The next step is to structure the results. In this case, we're interested in the address of the NFT and the owner of each returned asset. We use root to specify the length of the return and store our modified results for output:
4. Finally, we save the results to a file named "creatorResults.json" using the fs

module:

Finally, run the command node getAssetsByCreator.js

to execute the script and populate the "creatorResults.json" file with the retrieved results.

Result

The resulting JSON file contains the public key address of each owner and the respective assets they own. Each asset is represented by its ID, providing a clear snapshot of the ownership status of the assets created by the specific creator.

The 'ownershipId' represents the owner's public key address on the Solana blockchain, and the assets are the NFTs linked to the address.

Full Code

Make sure to replace

with your actual API key.

In this example we are making an asynchronous request to the DAS API for the “getAssetsByCreator” endpoint. Then, we are passing in our creator address and implying that they are a verified creator. Finally, we are then setting up our response to parse the owner of each asset that is under the creator address, and posting to an external JSON file.

Overall, this is a valuable tool for anyone interested in tracking or analyzing the movement of assets on the Solana blockchain, especially those tied to a specific creator or project.

7. Get Assets by Authority

The function getAssetsByAuthority

fetches assets associated with a specific update authority. An update authority is an address that possesses the rights to modify a collection. This feature becomes particularly valuable when you need to fetch a set of assets in cases where a collection ID doesn't exist. Additionally, it offers the advantage of retrieving a larger set of assets associated with an authority address, extending beyond the confines of a single collection ID.

Example

In the following example, we'll retrieve each collection NFT and their respective owner from Taiyo Robotics, Pilots, and Infants. Given that they all share a common update authority, it's possible to return all the results pertinent to the project.

This is an upgrade from the getAssetsByGroup

function, as you can group assets linked to a set authority CDgbhX61QFADQAeeYKP5BQ7nnzDyMkkR3NEhYF2ETn1k

.

1. Initiate by setting up the function in a newly created file named "getAssetsByAuthority.js":
2. Configure the fetch request and incorporate necessary parameters in the body, such as authorityAddress

, page

, and limit

:

Ensure to substitute

with the actual authority address whose assets you wish to extract.

1. It's now possible to arrange our return to cycle through results if there are 1000 results.

If the number of results is less, the flag will set to false, and terminating the pagination, in which hasMoreResults will return false.

1. Handle the assets to derive the required information. Here, we aim to display the ID of the asset and its owner. We'll also craft a root object to contain the count of assets and the processed asset array:
2. We're now prepared to set our returned items to post to a JSON called authorityResults.json, and we'll log once the process concludes.

Lastly, execute the command node getAssetsByAuthority.js

to run the script and populate the "authorityResults.json" file with the fetched results.

Result

The output for the getAssetsByAuthority

endpoint includes a count of assets and an array of objects, each representing an asset and containing its ID and owner. Here's a sample:

Full Code

Ensure to substitute

with your actual API key.

In this example, we employ the "getAssetsByAuthority" method to return all assets tied to an authority address across 3 collections. We utilize only page

and limit

as supplementary parameters for the request. Subsequently, we parse the results to extract the assets under the authority, and store them in an external JSON file.

Conclusion

You should now have a comprehensive understanding of how to use the Digital Asset Standard (DAS) API, along with several practical applications for each method. This marks a significant departure from previous practices of needing multiple endpoints to fetch specific asset information.

These methods cater to both compressed and regular assets, offering a unified interface for making these requests in various ways.

For more information, you can always refer to our extensivedocumentation on the Digital Asset Standard (DAS) API, and explore our Open API widget that provides a detailed breakdown of request and response parameters.

As always, we invite you to join our [Discord

](https://discord.gg/VbxaGJwmEr) community. Don't hesitate to ping me there if you have any queries!