⚡ Enhanced Websockets (beta)

Stream real-time transaction and account updates directly to your applications with our enhanced websocket integration.

Helius offers Geyser-enhanced Websockets alongside Solana's standard Websocket methods. These enhanced Websockets deliver faster response times compared to standard RPC Websockets.

These Websockets are currently in beta and may occasionally experience delayed data. To address this, we're developing Laserstream, a new product that offers replay capabilities and ensures real-time delivery without delays.

Join the Laserstream waitlist for updates.

This feature is available exclusively for **business and professional plans**. Please note that this is the **Enhanced Geyser WebSocket**, not the **Yellowstone Geyser**. For Yellowstone Geyser functionality, a Dedicated Node is required.

Subscription Endpoints

Geyser-enhanced websockets are currently available in `mainnet` and `devnet` with the following URLs

Mainnet: `wss://atlas-mainnet.helius-rpc.com?api-key=<API_KEY>`

Devnet: `wss://atlas-devnet.helius-rpc.com?api-key=<API_KEY>`

Transaction Subscribe

The `transactionSubscribe` websocket method enables real-time transaction events. To use it, provide a `TransactionSubscribeFilter` and optionally include

`TransactionSubscribeOptions` for further customization.

Example Payload:

```
{
  "jsonrpc": "2.0",
  "id": 420,
  "method": "transactionSubscribe",
  "params": [
      {
        "vote": false,
        "failed": false,
        "signature": "2dd5zTLrSs2udfNsegFRCnzSyQcPrM9svX6m1UbEM5bSdXXFj3XpqaodtK
        "accountInclude": ["pqx3fvvh6b2eZBfLhTtQ5KxzU3CginmgGTmDCjk8TPP"],
        "accountExclude": ["FbfwE8ZmVdwUbbEXdq4ofhuUEiAxeSk5kaoYrJJekpnZ"],
        "accountRequired": ["As1XYY9RdGkjs62isDhLKG3yxMCMatnbanXrqU85XvXW"]
      },
      {
    "commitment": "processed",
        "encoding": "base64",
        "transaction_details": "full",
        "showRewards": true,
        "maxSupportedTransactionVersion": 0
      }
  ]
}
```

Example Notification:

```
{
    "jsonrpc": "2.0",
    "method": "transactionNotification",
    "params": {
        "subscription": 4743323479349712,
        "result": {
            "transaction": {
                "transaction": [
                    "Ae6zfSExLsJ/E1+q0jI+3ueAtSoW+6HnuDohmuFwagUo2BU4OpkSdUKYNI1
                    "base64"
                ],
                "meta": {
                    "err": null,
                    "status": {
                        "Ok": null
                    },
                    "fee": 5000,
                    "preBalances": [
                        28279852264,
                        158122684,
                        1
                    ],
                    "postBalances": [
                        28279747264,
                        158222684,
                        1
                    ],
                    "innerInstructions": [],
                    "logMessages": [
                        "Program 11111111111111111111111111111111 invoke [1]",
                        "Program 11111111111111111111111111111111 success"
                    ],
                    "preTokenBalances": [],
                    "postTokenBalances": [],
                    "rewards": null,
                    "loadedAddresses": {
                        "writable": [],
                        "readonly": []
                    },
                    "computeUnitsConsumed": 0
                }
            },
            "signature": "5moMXe6VW7L7aQZskcAkKGQ1y19qqUT1teQKBNAAmipzdxdqVLAdG4
            "slot": 224341380
        }
    }
}
```

TransactionSubscribeFilter:

`vote: Option<bool>` : A flag to include/exclude vote-related transactions.

`failed: Option<bool>` : A flag to include/exclude transactions that failed.

`signature: Option<String>` : Filters updates to a specific transaction based on its signature.

`accountInclude: Option<Vec<String>>` : A list of accounts for which you want to receive transaction updates. This means that only one of the accounts must be included in the transaction updates (e.g., Account 1 OR Account 2).

`accountExclude: Option<Vec<String>>` : A list of accounts you want to exclude from transaction updates.

`accountRequired: Option<Vec<String>>` : Transactions must involve these specified accounts to be included in updates. This means that all of the accounts must be included in the transaction updates (e.g., Account 1 AND Account 2).

> You can include up to 50,000 addresses in the accountsInclude, accountExclude and accountRequired arrays.

`commitment: Option<Commitment>` : Specifies the commitment level for fetching data, dictating at what stage of the transactions lifecycle updates are sent. The possible values are

- processed
- confirmed
- finalized

`encoding: Option<UiTransactionEncoding>` : Sets the encoding format of the returned transaction data. The possible values are

- base58
- base64
- jsonParsed

`transactionDetails: Option<TransactionDetails>` : Determines the level of detail for the returned transaction data. The possible values are

- full
- signatures
- accounts
- none

`showRewards: Option<bool>` : A boolean flag indicating if reward data should be included in the transaction updates.

`maxSupportedTransactionVersion: Option<u8>` : Specifies the highest version of transactions for which you want to receive updates.

> `maxSupportedTransactionVersion` is required to return the accounts and full-level details of a given transaction (i.e., `transactionDetails: "accounts" | "full"`).

Transaction Subscribe code example:

In this example, we are subscribing to transactions that contain the Raydium account `675kPX9MHTjS2zt1qfr1NYHuzeLXfQM9H24wFSUt1Mp8` . Whenever a transaction occurs that contains `675kPX9MHTjS2zt1qfr1NYHuzeLXfQM9H24wFSUt1Mp8` in the `accountKeys` of the transaction, we will receive a websocket notification.

Based on the subscription options, the transaction notification will be sent at the `processed` commitment level, `jsonParsed` encoding, `full` transaction details, and will show rewards.

> Websockets have a 10-minute inactivity timer; implementing health checks and sending pings every minute is heavily recommended to keep the websocket connection alive.

```
const WebSocket = require('ws');

// Create a WebSocket connection
const ws = new WebSocket('wss://atlas-mainnet.helius-rpc.com/?api-key=<API_KEY>'

// Function to send a request to the WebSocket server
function sendRequest(ws) {
    const request = {
        jsonrpc: "2.0",
        id: 420,
        method: "transactionSubscribe",
        params: [
            {
                accountInclude: ["675kPX9MHTjS2zt1qfr1NYHuzeLXfQM9H24wFSUt1Mp8"]
            },
            {
                commitment: "processed",
                encoding: "jsonParsed",
                transactionDetails: "full",
                showRewards: true,
                maxSupportedTransactionVersion: 0
            }
        ]
    };
    ws.send(JSON.stringify(request));
}

// Function to send a ping to the WebSocket server
function startPing(ws) {
    setInterval(() => {
        if (ws.readyState === WebSocket.OPEN) {
            ws.ping();
            console.log('Ping sent');
        }
    }, 30000); // Ping every 30 seconds
}

// Define WebSocket event handlers

ws.on('open', function open() {
    console.log('WebSocket is open');
    sendRequest(ws);  // Send a request once the WebSocket is open
    startPing(ws);    // Start sending pings
});

ws.on('message', function incoming(data) {
    const messageStr = data.toString('utf8');
    try {
        const messageObj = JSON.parse(messageStr);
```

```
        console.log('Received:', messageObj);
    } catch (e) {
        console.error('Failed to parse JSON:', e);
    }
});

ws.on('error', function error(err) {
    console.error('WebSocket error:', err);
});

ws.on('close', function close() {
    console.log('WebSocket is closed');
});
```

Account Subscribe

Solana's Websockets supports a method that allows you to subscribe to an account and receive notifications via the websocket connection whenever there are changes to the lamports or data associated with a matching account public key. This method aligns directly with the Solana Websocket API specification.

Example Payload:

```
{
  "jsonrpc": "2.0",
  "id": 420,
  "method": "accountSubscribe",
   "params": [
    "SysvarC1ock11111111111111111111111111111111",
    {
      "encoding": "jsonParsed",
      "commitment": "finalized"
    }
  ]
}
```

Example Notification:

```
{
    'jsonrpc': '2.0',
    'method': 'accountNotification',
    'params':
        {
          'subscription': 237508762798666,
          'result':
           {
            'context': {'slot': 235781083},
            'value':
             {
             'lamports': 1169280,
             'data': 'BvEhEb6hixL3QPn41gHcyi2CDGKt381jbNKFFCQr6XDTzCTXCuSUG9D',
             'owner': 'Sysvar1111111111111111111111111111111111111',
             'executable': False,
             'rentEpoch': 361,
             'space': 40
             }
           }
        }
}
```

Parameters:

`String` : Account public key, sent in base58 format, required

`object: Option<RpcAccountInfoConfig>:` Optional object used to pass in `encoding` for the data returned in the `AccountNotification` and `commitment`

If nothing is passed into the object, the response will default to base58 encoding and a `finalized` commitment level.

```
pub struct RpcAccountInfoConfig {
    pub encoding: Option<UiAccountEncoding>, // supported values: base58, base64
    pub commitment: Option<CommitmentConfig>, // supported values: finalized, co
}
```

Account Subscribe code example:

In this example, we are subscribing to account changes for the account `SysvarC1ock11111111111111111111111111111111` .

We will see an update whenever a change occurs to the account data or the lamports for this account.
This happens at a frequent interval for this specific account as the `slot` and `unixTimestamp` are both a part of the returned account data.

```
// Create a WebSocket connection
const ws = new WebSocket('wss://atlas-mainnet.helius-rpc.com?api-key=<API_KEY>')

// Function to send a request to the WebSocket server
function sendRequest(ws) {
    const request = {
        jsonrpc: "2.0",
        id: 420,
        method: "accountSubscribe",
        params: [
            "SysvarC1ock11111111111111111111111111111111", // pubkey of account
            {
                encoding: "jsonParsed", // base58, base64, base65+zstd, jsonPars
                commitment: "confirmed", // defaults to finalized if unset
            }
        ]
    };
    ws.send(JSON.stringify(request));
}

// Function to send a ping to the WebSocket server
function startPing(ws) {
    setInterval(() => {
        if (ws.readyState === WebSocket.OPEN) {
            ws.ping();
            console.log('Ping sent');
        }
    }, 30000); // Ping every 30 seconds
}

// Define WebSocket event handlers

ws.on('open', function open() {
    console.log('WebSocket is open');
    sendRequest(ws);  // Send a request once the WebSocket is open
    startPing(ws);    // Start sending pings
});

ws.on('message', function incoming(data) {
    const messageStr = data.toString('utf8');
    try {
        const messageObj = JSON.parse(messageStr);
        console.log('Received:', messageObj);
    } catch (e) {
        console.error('Failed to parse JSON:', e);
    }
});

ws.on('error', function error(err) {
```

```
    console.error('WebSocket error:', err);
});

ws.on('close', function close() {
    console.log('WebSocket is closed');
});
```

Last updated 23 hours ago