ChatGPT

Extracting Data from Dune's Hyperliquid Dashboard and Building a Replit Dashboard

Overview: This guide provides a comprehensive workflow for extracting all data from the **Hyperliquid Stats + User Analytics** Dune dashboard (created by *x3research*) using Dune's API and integrating it into a custom web dashboard on Replit. We'll cover identifying the dashboard's datasets (queries), using the Dune API with the provided API key, choosing an appropriate data format (JSON), building the Replit-hosted dashboard to display the data, and adding proper credit to the original author. Code examples (for API calls, JSON handling, and front-end integration) are included to illustrate each step.

1. Identifying Datasets and Charts on the Dune Dashboard

Each visualization (chart or table) on a Dune dashboard is powered by an underlying SQL query. To extract the dashboard's data, you first need to identify all the queries used in the **x3research/hyperliquid** dashboard:

- **Access the Dashboard:** Open the Hyperliquid dashboard on Dune (at `dune.com/x3research/hyperliquid`). Ensure you have a Dune account if needed (the dashboard appears to be public, but logging in may enable extra features).

- **List Query IDs via GitHub Integration:** Dune provides a "GitHub" button on the dashboard (usually in the top-right) that lists all query IDs in that dashboard. Click this button to get a list of query IDs used in the Hyperliquid dashboard [1] [2]. Each ID corresponds to a dataset/chart. Note down all the query IDs from this list.

- **OR, Find Query IDs Manually:** If the GitHub button isn't available, you can manually retrieve each query ID:

- Click on a chart or table in the dashboard (e.g., click the chart's title or the *view query* option). This opens the query editor page for that visualization.
- Look at the URL of the query page. It will have a format like:

```
https://dune.com/queries/<query_id>/<some-chart-id>
```

  The first number in this URL is the query's unique ID [3]. For example, a URL `dune.com/queries/1443379/2445683` indicates a query ID of **1443379** [3]. Record each query ID.
- Repeat for every chart on the dashboard to collect all query IDs (each chart corresponds to one query). The Hyperliquid dashboard likely includes multiple metrics (e.g., cumulative volume, total inflow, trade counts, user stats, etc.), so be sure to capture each relevant query ID.

By the end of this step, you should have a list of all **query IDs** that power the Hyperliquid dashboard. These IDs will be used to fetch data via the API.

2. Using the Dune API (with API Key) to Retrieve Chart Data

With the query IDs in hand, you can programmatically fetch their results using Dune's API. The provided API key (`pZBvRD0acWVAtWRwnTtOuZgUvuETutIt`) will be used to authenticate requests. Here's how to proceed:

- **API Authentication:** Dune's API uses API keys for authentication. You can provide the key in one of two ways:
- **HTTP Header:** Include an `X-DUNE-API-KEY` header in your requests [4]. This is the recommended secure method.
- **Query Parameter:** Alternatively, append `?api_key=<YOUR_API_KEY>` to the API URL [5]. This is convenient for quick tests or environments where adding headers is difficult.

For example: Both of the following are valid ways to authenticate:
Header method: `X-DUNE-API-KEY: pZBvRD0acWVAtWRwnTtOuZgUvuETutIt`
Query param method: `...?api_key=pZBvRD0acWVAtWRwnTtOuZgUvuETutIt`

- **Choosing an API Endpoint:** Dune offers endpoints to either **execute a query** or **fetch the latest results** of a query. For our use-case (extracting dashboard data), using the "get latest result" endpoint is usually simplest:
- **Get Latest Result** (`GET /api/v1/query/{query_id}/results`): This returns the most recent result of the query in JSON format [6] without rerunning it. It will retrieve whatever data is currently displayed on the dashboard for that query. (Using this does **not** trigger a new execution of the query, but note that it still consumes API credits based on data size [7].)

- **Execute Query** (`POST /api/v1/query/{query_id}/execute`): This triggers a fresh execution of the query on Dune's engines [8]. You would use this if you need up-to-the-minute data and want to re-run the query. The execute endpoint returns an `execution_id` and an initial status (e.g. "QUERY_STATE_PENDING") [9]. After executing, you'd poll the execution status and then fetch results via the execution ID. This approach is more complex (requiring multiple calls), so unless real-time freshness is required, the simpler "latest result" call is preferred.

- **Fetch Data via API Calls:** For each query ID from step 1, make an API call to retrieve its data. Below are examples using both Python and JavaScript:

- *Python example (using `requests`):*

```python
import requests

query_id = 1443379  # example query ID
url = f"https://api.dune.com/api/v1/query/{query_id}/results"
headers = {"X-DUNE-API-KEY": "pZBvRD0acWVAtWRwnTtOuZgUvuETutIt"}
response = requests.get(url, headers=headers)
```

```
data = response.json()  # parse the JSON response
print(data)
```

In this code, we send a GET request to the Dune API's query results endpoint with our API key in the header. The result (in `data`) will be a JSON object containing the query's results [6]. Each result includes metadata (column names/types, row count, etc.) and the actual rows of data.

- *JavaScript example (using `fetch` in Node or browser):*

```
const queryId = 1443379; // example query ID
const apiKey = "pZBvRDOacWVAtWRwnTtOuZgUvuETutIt";
const url = `https://api.dune.com/api/v1/query/${queryId}/results?api_key=${apiKey}`;

fetch(url)
  .then(res => res.json())
  .then(data => {
    console.log(data);
    // TODO: handle the data (e.g., save it or send it to frontend)
  })
  .catch(err => console.error(err));
```

Here we append the API key as a query parameter for simplicity [5]. The `data` received will be the JSON result for that query.

- **Handling Large Results:** If a query returns a very large dataset (e.g., thousands of rows), the API may paginate the output. The JSON response includes fields like `next_offset` and `next_uri` when not all rows are returned in one go [10]. To get all data, you can follow the `next_uri` link or make subsequent requests with `offset` parameters until all pages are retrieved [11]. (The Dune API has a maximum result size per request; for very large results consider using pagination or the CSV download, as discussed below.)

- **Repeating for All Queries:** Loop through each identified query ID and fetch its JSON result as above. You might write a script to gather all data and store each dataset, or fetch on-demand in your application (more on that in the Replit integration section).

By following these steps, you programmatically obtain all the datasets behind the Hyperliquid dashboard, using the API key to authenticate each request. You now have the raw data for each chart in JSON form, ready for use in your custom dashboard.

3. Choosing the Best Data Format (JSON vs. CSV)

For integration into a web dashboard (especially in a Replit environment), **JSON** is the default and recommended format. The Dune API's query result endpoint returns data as JSON by default, which is ideal for web development: - JSON is easy to consume in JavaScript (you can directly use `fetch` and

`response.json()` as shown, and then manipulate the data) and in Python (via the `json()` method in `requests` or using the built-in `json` module). - Nested JSON structures can represent complex data (tables, arrays of records) in a format that is readily accessible in code.

Dune does offer results in CSV as well – for example, there is a corresponding endpoint `GET /api/v1/query/{query_id}/results/csv` to fetch CSV format [12]. However, using CSV would require an extra parsing step (converting CSV text into usable data structures) and is generally less convenient for web apps. CSV might be useful if you plan to import data into spreadsheet software, but for a Replit-hosted **web dashboard**, JSON is more straightforward.

Recommendation: Use JSON for all data transfers between Dune and your application. All examples in this guide assume JSON. (If you later need CSV for some reason, you can call the CSV endpoint or use the Dune Python client's `run_query_csv()` method, but this is optional.)

4. Building the Custom Dashboard on Replit

With data available via the Dune API, the next step is to build a web dashboard on Replit that fetches and displays this data. The process involves setting up a web application (Node.js or Python) on Replit, retrieving the data from Dune (using the API key), and rendering it in a user-friendly format (charts, tables, etc.) in a web page.

Step 4.1: Set Up a Replit Project – Log in to Replit and create a new project. You can choose a **Node.js** template (if you plan to use JavaScript/Node for the backend) or **Python (Flask)** if you prefer Python. For a web dashboard, Node.js with Express or a simple static file server is common, but either can work. Replit will provide an online IDE where you can edit code and will host your app at a URL once it's running.

Step 4.2: Securely Store the API Key – In your Replit project, add the Dune API key as a secret environment variable instead of hard-coding it. In Replit's interface, open **Secrets / Environment Variables**, and create a new variable (e.g. name it `DUNE_API_KEY`) with the value `pZBvRD0acWVAtWRwnTtOuZgUvuETutIt`. This keeps the key hidden in your code (important since your Replit might be public). In your application code, read this environment variable: - In Node.js: `const apiKey = process.env.DUNE_API_KEY;` - In Python: `api_key = os.environ.get('DUNE_API_KEY')`

Replit automatically injects secret vars into your app's environment. Using `process.env` or `os.environ` ensures the key stays out of your source code [13].

Step 4.3: Backend – Fetch Data from Dune – Next, write server-side code to call the Dune API for the queries you need, using the environment API key. You have a couple of options for how and when to fetch data: - *On-Demand fetching:* Your frontend can request data from your backend, and the backend in turn calls the Dune API each time. This ensures data is relatively fresh (though Dune's API might return cached results unless you force a refresh). - *Pre-fetch and store:* The server can fetch all needed data at startup (or on a schedule) and cache it in memory or a database. The frontend then simply reads the cached data. This reduces API calls but means data might not update until you refresh the cache.

For simplicity, we'll outline an on-demand approach using Node.js and Express: 1. **Initialize an Express Server:** Install Express (`npm install express node-fetch`) if not already. Then create a basic Express app:

```
const express = require('express');
const fetch = require('node-fetch');  // if Node 18+, fetch is global
const app = express();
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

In Replit, it's important to listen on `process.env.PORT` (which Replit defines) or a default like 3000, so that the web server is exposed properly. 2. **Define an API route to get Dune data:** Create endpoints that the frontend can call to retrieve each dataset. For example, to serve the "Hyperliquid stats" data:

```
app.get('/api/hyperliquid', async (req, res) => {
  try {
    const apiKey = process.env.DUNE_API_KEY;
    const queryId = 1443379;  // replace with actual ID
    const url = `https://api.dune.com/api/v1/query/${queryId}/results?api_key=$
{apiKey}`;
    const duneRes = await fetch(url);
    const data = await duneRes.json();
    res.json(data);  // Forward the JSON directly to the client
  } catch (err) {
    console.error('Dune API fetch error:', err);
    res.status(500).send('Error fetching data');
  }
});
```

This route calls Dune's API for a specific query and returns the result JSON to the browser. You can create multiple routes (or a single route that takes a query ID as a parameter) for each dataset in the dashboard. For instance, you might have `/api/hyperliquid/volume`, `/api/hyperliquid/users`, etc., each hitting the corresponding query ID. The example above uses the query ID directly; you can substitute it with variables or route params as needed.
Note: We use the query parameter authentication in the URL for simplicity. Alternatively, you could use `fetch(url, { headers: { 'X-DUNE-API-KEY': apiKey } })` to keep the key out of the URL.

1. **Enable CORS (if needed):** If your frontend is served from a different domain or port than the backend, enable CORS so the browser can request the API. Since in Replit the frontend and backend are typically the same app/origin, you might not need this. But if you do, consider using the `cors` package (`npm install cors`) and `app.use(cors())`.

2. **Test the data endpoint:** Run the Replit project. Open the web URL (Replit provides a preview or live link) and try hitting the `/api/hyperliquid` endpoint (e.g., `https://<your-repl>.repl.co/`

`api/hyperliquid`). You should see a JSON response containing the data for that query (with fields like `column_names`, `rows`, etc.). This verifies your server can reach Dune and retrieve data.

Step 4.4: Frontend – Display the Data – Now create a front-end interface to present the data nicely to users. This typically involves an HTML file, some JavaScript to fetch data from your backend, and possibly a charting library for visualizations. In an Express app, you can serve static files or use a template engine; the simplest approach is to serve an `index.html` from a `public` directory or via a route.

Minimal example of an HTML + JS front-end (you can place an `index.html` in a public folder and use `app.use(express.static('public'))` to serve it):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Hyperliquid Dashboard</title>
  <!-- You can include a CSS framework or custom styles here -->
</head>
<body>
  <h1>Hyperliquid Dashboard Metrics</h1>

  <div>
    <h3>Cumulative Volume:</h3>
    <p id="cumulative-volume">Loading...</p>
  </div>
  <div>
    <h3>Total Users Over Time:</h3>
    <canvas id="users-chart" width="400" height="200"></canvas>
  </div>
  <!-- ... other sections for each metric/chart ... -->

  <footer id="footer"></footer>

  <script>
    // Fetch cumulative volume (example for a single-value metric)
    fetch('/api/hyperliquid')  // assuming this returns a JSON with the needed
data
      .then(res => res.json())
      .then(data => {
        // Suppose the query returns a single-row result with a column "Volume"
        const volume = data.result.rows[0]?.Volume;
        document.getElementById('cumulative-volume').textContent = volume ? `$$
{volume}` : 'N/A';
      });
```

```
    // Fetch user stats (example for a time-series dataset)
    fetch('/api/hyperliquid/
users')  // a different endpoint for user stats, for example
      .then(res => res.json())
      .then(data => {
        // Assume this query returns multiple rows with columns "date" and
"new_users"
        const rows = data.result.rows;
        const dates = rows.map(r => r.date);
        const newUsers = rows.map(r => r.new_users);
        // Plot using Chart.js (if included via <script src="https://
cdn.jsdelivr.net/npm/chart.js"></script>)
        const ctx = document.getElementById('users-chart').getContext('2d');
        new Chart(ctx, {
          type: 'line',
          data: {
            labels: dates,
            datasets: [{ label: 'Daily New Users', data: newUsers }]
          }
        });
      });

    // Set footer credit (we'll fill this in step 5 below)
    document.getElementById('footer').innerHTML = "…";
  </script>
</body>
</html>
```

In the above front-end code, we perform AJAX calls to our own API endpoints: - We update a text placeholder with the cumulative volume (fetched from our `/api/hyperliquid` route). We assume the JSON has a key `data.result.rows` which is an array of result rows. We take the first row's `Volume` field and insert it into the page. - We fetch a hypothetical `/api/hyperliquid/users` endpoint that returns a time-series of daily new users. We then use Chart.js (a popular chart library) to plot a line chart of new users over time. We mapped the JSON rows into two arrays: `dates` and `newUsers`, then fed those to Chart.js.

You will need to adjust the parsing logic based on the actual column names in the JSON data. The Dune query result JSON includes `result.rows` (an array of objects, where each object's keys are the column names). Use `console.log(data)` during development to inspect the structure and adjust indices or keys accordingly.

Step 4.5: Iterate and Refine: Repeat the process of creating front-end elements and fetch calls for each dataset you want to display. For multiple charts, consider organizing the code and maybe using a framework or at least separating data fetching into functions to keep it clean. Ensure the page is responsive (for Replit, you can just open in a new window to view it as a normal website).

At this point, your Replit app should be pulling data from the Dune dashboard's queries and displaying key metrics and charts on a live webpage. Each time the page loads (or on a set interval, if you choose to auto-refresh data), the site will contact your backend, which in turn calls the Dune API to get the latest figures.

5. Adding Attribution to the Original Dashboard Author

It's important to credit the source of the data and the original Dune dashboard author, both for transparency and courtesy. We'll add a footer note in the dashboard for this purpose.

In the HTML footer (as shown by the `<footer>` element in the example above), include a line of credit to *x3research* and Dune. For example:

```
<footer>
  Data sourced from Dune – "<a href="https://dune.com/x3research/hyperliquid"
target="_blank">Hyperliquid Stats</a>" dashboard by <strong>x3research</strong>.
</footer>
```

This footer will appear at the bottom of your dashboard, with a link to the original Dune dashboard. We use the `target="_blank"` so that clicking the link opens Dune in a new tab (keeping your dashboard open).

By including this attribution, viewers of your Replit dashboard know where the data comes from, and you acknowledge the work of the Dune analyst who created the queries (x3research). This is considered good practice when reusing data or queries from others. *(Note: If the Dune dashboard has a specific license or Dune's terms of service require attribution, this step helps comply with those as well.)*

Finally, update the footer via script if needed (in the example above, we left a placeholder). For instance:

```
document.getElementById('footer').innerHTML =
  'Data sourced from Dune – "<a href="https://dune.com/x3research/hyperliquid"
target="_blank">Hyperliquid Stats</a>" by <strong>x3research</strong>.';
```

This will inject the credit text into the page. (You can also just hard-code it in the HTML as shown earlier.)

By completing all the steps above, you now have:

- Identified all relevant query IDs from the Hyperliquid Dune dashboard [1] [3].
- Used the Dune API with your API key to programmatically retrieve each query's data in JSON format [6].
- Chosen JSON as the data format for easy integration (with awareness of other options like CSV).
- Built a web application on Replit that fetches the data from Dune and displays interactive charts/metrics.
- Added proper credit to the original dashboard author (x3research) in your dashboard's footer.

This setup allows you to run your own custom dashboard, powered by Dune's on-chain data, on a Replit-hosted site. The dashboard will reflect the data from the Hyperliquid Stats Dune dashboard and can be styled or expanded as you wish. By periodically refreshing the data (you could implement a refresh button or an interval timer in your JS), your dashboard can stay up-to-date with minimal manual effort.

References and Further Reading:

- Dune Analytics API documentation – covering query execution and results retrieval [6] [8].
- Dune API Authentication methods (using API key in header or URL) [4] [5].
- Dune Query Management guide – how to get query IDs from dashboards [1].
- Artemis (Excel/Sheets) guide – example of finding query IDs from a dashboard [3].
- Replit documentation on using environment variables – storing secrets like API keys [13].

---

[1] [2] Query Management - Dune Docs
https://docs.dune.com/api-reference/quickstart/queries-eg

[3] Dune - Artemis API Docs
https://app.artemisanalytics.com/docs/artemis-sheets/usage/dune

[4] [5] Authentication - Dune Docs
https://docs.dune.com/api-reference/overview/authentication

[6] [7] [10] [11] [12] Get Latest Query Result - Dune Docs
https://docs.dune.com/api-reference/executions/endpoint/get-query-result

[8] [9] Execute Query - Dune Docs
https://docs.dune.com/api-reference/executions/endpoint/execute-query

[13] Use .env file in Replit - JavaScript - The freeCodeCamp Forum
https://forum.freecodecamp.org/t/use-env-file-in-replit/506681