grist-kit

Release 0.1.2

Jul 05, 2026

CLI

---

grist-kit is a CLI and type-safe client library for Grist[1], written in TypeScript. It provides:

- An agent-friendly **CLI** for inspecting and querying Grist documents from the shell: list tables, fetch records, run SQL queries.
- A **type-safe client library**: generate a TypeScript schema from a live doc, then read and write records with column names, filter values, and write payloads checked at compile time.

1 Quick Start

You can query any public Grist document with just `--doc-url` — no configuration needed. The examples below use the public Inventory Manager[2] template.

List the tables and their columns:

```
npx -y grist-kit --doc-url
↪https://templates.getgrist.com/api/docs/sXsBGDTKau1F3fvxkCyoaJ tables
```

Fetch records as CSV:

```
npx -y grist-kit --doc-url
↪https://templates.getgrist.com/api/docs/sXsBGDTKau1F3fvxkCyoaJ \
  records All_Products --limit 3 --format csv
```

```
id,SKU,Product,In_Stock,Stock_Alert,...
1,VEG-BLCK-28,Men's Stretch Five-Pocket Pants,8,In Stock,...
2,VEG-BLCK-30,Men's Stretch Five-Pocket Pants,8,In Stock,...
3,VEG-BLCK-32,Men's Stretch Five-Pocket Pants,9,In Stock,...
```

Filter by a column value:

[1] https://www.getgrist.com

[2] https://templates.getgrist.com/sXsBGDTKau1F/Inventory-Manager

```
npx -y grist-kit --doc-url
↪https://templates.getgrist.com/api/docs/sXsBGDTKau1F3fvxkCyoaJ \
  records All_Products --filter Stock_Alert="Low Stock" --format csv
```

```
id,SKU,Product,In_Stock,Stock_Alert,...
5,VEG-BLCK-36,Men's Stretch Five-Pocket Pants,2,Low Stock,...
6,VEG-BLCK-38,Men's Stretch Five-Pocket Pants,2,Low Stock,...
...
```

For private documents, set up credentials — see `grist-kit help configuration`.

2 Configuration

All CLI commands require a Grist document URL. Credentials are read from environment variables or flags.

2.1 Environment variables

- *GRIST_DOC_URL* — base doc URL, e.g. `https://grist.example.com/api/docs/xxxx`. Required for all commands. Find it in Grist under **Settings** → **API** → **Document ID** → **Base doc URL**.
- *GRIST_API_KEY* / *GRIST_ACCESS_TOKEN* — provide one (not both). Neither is needed for public documents. See `grist-kit help authentication` for the difference.

The CLI reads from environment variables. It also supports loading a `.env` file, though real environment variables take precedence.

2.2 Flags

These flags are accepted by every command and override the corresponding environment variable.

- `--doc-url <url>` — overrides *GRIST_DOC_URL*.
- `--api-key <key>` — overrides *GRIST_API_KEY*.
- `--access-token <token>` — overrides *GRIST_ACCESS_TOKEN*.
- `--env-file <path>` — load a specific `.env` file instead of the default.

3 Listing Tables

The `tables` command lists all tables in a Grist document, along with their columns and types.

```
grist-kit tables [options]
```

Options:

- `--format <text|json>` — output format. Defaults to `text`.

3.1 Text output

```
grist-kit tables
```

```
All_Products
  SKU Text
  Product Text
  Size Choice
  In_Stock Numeric
  Stock_Alert Choice
  Brand Ref:Add_Products
  Color Ref:Color
  ...
Incoming_Orders
  Order_Date Date
  Status Choice
  Total Numeric
  ...
```

3.2 JSON output

Use `--format json` to get the full column metadata from the Grist API, including labels, formulas, and widget options. Useful for inspecting the document schema.

```
grist-kit tables --format json
```

```
[
  {
    "id": "All_Products",
    "fields": { "tableRef": 1, "primaryViewId": 1, ... },
    "columns": [
      {
        "id": "SKU",
        "fields": {
          "label": "SKU",
          "type": "Text",
          "isFormula": false,
          "formula": "",
          ...
        }
      },
      ...
    ]
  },
  ...
]
```

4 Fetching Records

The `records` command fetches rows from a table and prints them as JSON (default) or CSV.

```
grist-kit records <table> [options]
```

Options:

- `--filter <key=value>` — filter rows by column value. Repeatable; repeating the same key matches rows where the column equals any of the given values. Values `true`, `false`, `null`, and numbers are coerced to their JSON types; everything else is treated as a string.
- `--limit <n>` — maximum number of rows to return.
- `--format <json|csv>` — output format. Defaults to `json`.

Examples:

```
grist-kit records All_Products --limit 2
```

```
[
  {
    "id": 1,
    "SKU": "VEG-BLCK-28",
    "Product": "Men's Stretch Five-Pocket Pants",
    "In_Stock": 8,
    "Stock_Alert": "In Stock"
  },
  ...
]
```

```
grist-kit records All_Products --filter Stock_Alert="Low Stock" --format csv
```

For anything beyond equality filters, use SQL — see `grist-kit help sql`.

5 Running SQL Queries

The `sql` command runs a read-only SQL query against the document and prints the results.

```
grist-kit sql "<query>" [options]
```

Options:

- `--format <json|csv>` — output format. Defaults to `json`.

Example:

```
grist-kit sql "SELECT SKU, In_Stock FROM All_Products ORDER BY In_Stock LIMIT 3" --
↪format csv
```

```
SKU,In_Stock
VEG-BLCK-34,0
```

(continues on next page)

(continued from previous page)

```
SEW-LTBL-36,0
SEW-LTBL-28,0
```

6 Generating a Type Definition File

Before using the type-safe client library, generate a TypeScript type definition file from your live Grist document. This file describes your tables and columns, and is passed as a type parameter to `gristDoc` to enable typesafe column access, filters, and inserts.

The `generate` command reads the document schema via the Grist API and writes the type definition file. See `grist-kit help configuration` for how to configure the document URL and API key.

```
grist-kit generate [options]
```

Options:

- `--out <path>` — output file path. Defaults to `./grist-schema.ts`.
- `--type-name <name>` — name of the exported TypeScript type. Defaults to `GristSchema`.

Example:

```
npx grist-kit generate --out grist-schema.ts
```

Re-run this command whenever the document's columns change.

> **Tip:** Save it as a `package.json` script so it's easy to remember:
>
> ```
> pnpm pkg set scripts.update-schema="grist-kit generate --out grist-schema.ts"
> ```

7 Using the Client Library

The grist-kit client library lets you query and manipulate Grist documents from TypeScript with full type safety. Column names, filter values, and write payloads are all checked against a generated schema at compile time.

7.1 Installation

```
npm install grist-kit
```

7.2 Prerequisites

Before using the client, generate a schema file from your live doc:

```
npx grist-kit generate --out grist-schema.ts
```

See `grist-kit help generate` for details.

7.3 Creating a client

```ts
import { gristDoc } from "grist-kit";
import type { GristSchema } from "./grist-schema.ts";

const doc = gristDoc<GristSchema>({
  baseDocUrl: process.env.GRIST_DOC_URL!,
  apiKey: process.env.GRIST_API_KEY,
});
```

Pass the generated schema as the type parameter to enable type inference across all table and column operations.

Options:

- `baseDocUrl` — base URL of the Grist document. Required.
- `apiKey` / `accessToken` — one of these is required unless the document is public. See `grist-kit help authentication` for the difference and security checklist.
- `fetchOptions` — additional options forwarded to the underlying `ofetch` client — useful for custom headers or timeouts.

Next steps:

- Read and write rows — see `grist-kit help crud`.
- Work with attachments — see `grist-kit help attachments`.

7.4 API reference

- gristDoc[3] — factory function
- GristDoc[4] — document client

8 Authentication

grist-kit supports two ways to authenticate against a Grist document: an API key, or a Grist access token. They differ in lifetime, scope, and how they are sent on the wire. Pick the one that matches your use case.

8.1 API key

```ts
import { gristDoc } from "grist-kit";

const doc = gristDoc({
  baseDocUrl: process.env.GRIST_DOC_URL!,
  apiKey: process.env.GRIST_API_KEY,
});
```

[3] https://apiref-site.vercel.app/package/grist-kit/gristDoc
[4] https://apiref-site.vercel.app/package/grist-kit/GristDoc

API keys are long-lived credentials tied to a single Grist user. They are sent as `Authorization: Bearer <key>` on every request. Generate one in Grist's **Profile Settings** → **API**. Use this when you have a stable server-to-server trust relationship with Grist.

8.2 Access token

```
import { gristDoc } from "grist-kit";

const doc = gristDoc({
  baseDocUrl: tokenInfo.baseUrl,
  accessToken: tokenInfo.token,
});
```

Access tokens are short-lived (15-minute default TTL), document-scoped JWTs. They are sent as a `?auth=<token>` query parameter on every request. They are the auth mechanism used by Grist's custom-widget plugin API (`grist.docApi.getAccessToken()`).

A function is also accepted, invoked **per request**:

```
const doc = gristDoc({
  baseDocUrl: tokenInfo.baseUrl,
  accessToken: () => tokenInfo.token,
});
```

Use a function when the token may need to be refreshed (e.g., a widget whose user keeps the tab open for more than 15 minutes). Use a plain string when the token arrives from outside grist-kit (e.g., a backend that receives it in a request from a widget) and is used for a single short-lived operation.

8.3 Mutual exclusion

You may not set both `apiKey` and `accessToken` at the same time. `createRequester` throws:

```
Specify either apiKey or accessToken, not both.
```

An empty string for either field is treated as "not set", so `apiKey: ""` + `accessToken: "tok"` is valid (only the access token is used).

8.4 Unauthenticated

If neither field is set, requests are sent without any auth header or query parameter. This works for public Grist documents.

9 Reading and Writing Records

Access a table via `doc.table("TableName")`. All methods are typed against the generated schema.

9.1 list

Fetches rows matching the given options.

```
const products = await doc.table("All_Products").list({
  filter: { Stock_Alert: ["Low Stock", "OUT OF STOCK"] },
  sort: "-In_Stock",
  limit: 100,
});
```

- `filter` — equality-style filters keyed by column name. Each value is an array of allowed values.
- `sort` — column name to sort by. Prefix with `-` for descending order. Accepts a single value or an array.
- `limit` — maximum number of rows to return.
- `hidden` — include hidden columns in the response.

9.2 insert

Inserts one or more rows and returns their row IDs.

```
const [orderId] = await doc
  .table("Incoming_Orders")
  .insert([{ Order_Date: Math.floor(Date.now() / 1000), Status: "Order Placed" }]);
```

Formula columns are excluded from the insert payload type automatically.

9.3 update

Updates existing rows by ID.

```
await doc.table("Incoming_Orders").update([{ id: orderId, Status: "Received" }]);
```

9.4 upsert

Creates or updates rows using Grist's upsert endpoint. Each record specifies `require` (match criteria) and `fields` (values to write).

```
await doc
  .table("Customers")
  .upsert([{ require: { Email: "alice@example.com" }, fields: { Name: "Alice", Tier:
↪"pro" } }]);
```

- `onMany` — behavior when multiple rows match: `"first"`, `"none"`, or `"all"`.
- `noCreate` — do not create a new row when no match is found.
- `noUpdate` — do not update an existing row when a match is found.

9.5 delete

Deletes rows by row ID.

```
await doc.table("Incoming_Orders").delete([orderId]);
```

9.6 API reference

- GristTable[5] — table client
- GristTable.list[6]

10 Working with Attachments

Attachment operations are available via `doc.attachments`.

10.1 upload

Uploads one or more files and returns their attachment IDs. These IDs can be stored in `Attachments`-typed columns.

```
const [id] = await doc.attachments.upload([
  { filename: "report.pdf", data: pdfBuffer, type: "application/pdf" },
]);
```

Accepts `File`, `Blob`, or an object with `filename`, `data` (a `Blob`, `Uint8Array`, or `ArrayBuffer`), and an optional `type`.

10.2 get

Retrieves metadata for an attachment by ID.

```
const meta = await doc.attachments.get(id);
console.log(meta.fileName, meta.fileSize, meta.timeUploaded);
```

10.3 download

Downloads an attachment as a `Blob`.

```
const blob = await doc.attachments.download(id);
```

10.4 downloadStream

Downloads an attachment as a readable byte stream.

[5] https://apiref-site.vercel.app/package/grist-kit/GristTable
[6] https://apiref-site.vercel.app/package/grist-kit/GristTable/list

```
const stream = await doc.attachments.downloadStream(id);
```

10.5 API reference

- GristAttachments[7] — attachment client

11 Full Example

This example shows a complete script that connects to a Grist doc, reads low-stock products, and places a refill order. It uses the Inventory Manager[8] template and covers the full workflow: setup, type generation, and a typesafe read/write script.

Prerequisites: Node.js 24+[9] and pnpm[10].

11.1 Setup

Create a project and install grist-kit:

```
mkdir grist-inventory && cd grist-inventory
pnpm init --init-type module
pnpm add grist-kit
pnpm add -D typescript @types/node @tsconfig/node24
```

Create `tsconfig.json`:

```
{
  "extends": "@tsconfig/node24/tsconfig.json"
}
```

Open the Inventory Manager template doc[11]. In the left sidebar, click **Settings** → **API** → expand **Document ID** → copy the **Base doc URL**.

Create `.env`:

```
GRIST_DOC_URL=https://templates.getgrist.com/api/docs/sXsBGDTKau1F3fvxkCyoaJ
```

Generate types from the live doc:

```
pnpm exec grist-kit generate --out grist-schema.ts
```

11.2 The script

Create `refill.ts`:

[7] https://apiref-site.vercel.app/package/grist-kit/GristAttachments
[8] https://www.getgrist.com/templates/inventory-manager/
[9] https://nodejs.org/docs/latest/api/typescript.html
[10] https://pnpm.io/
[11] https://templates.getgrist.com/sXsBGDTKau1F/Inventory-Manager

```ts
import { parseArgs } from "node:util";
import { gristDoc } from "grist-kit";
import type { GristSchema } from "./grist-schema.ts";

const { values } = parseArgs({
  options: { "dry-run": { type: "boolean", default: false } },
});

const doc = gristDoc<GristSchema>({
  baseDocUrl: process.env.GRIST_DOC_URL!,
  apiKey: process.env.GRIST_API_KEY,
});

const needsRefill = await doc.table("All_Products").list({
  filter: { Stock_Alert: ["Low Stock", "OUT OF STOCK"] },
});

console.log(`${needsRefill.length} products need a refill:\n`);
for (const p of needsRefill) {
  console.log(`  [${p.Stock_Alert}] ${p.SKU} - ${p.Product} (in stock: ${p.In_Stock}
↪)`);
}

if (needsRefill.length === 0 || values["dry-run"]) {
  if (values["dry-run"]) console.log("\n(dry run - no order created)");
  process.exit(0);
}

const [orderId] = await doc
  .table("Incoming_Orders")
  .insert([{ Order_Date: Math.floor(Date.now() / 1000), Status: "Order Placed" }]);

await doc.table("Incoming_Order_Line_Items").insert(
  needsRefill.map((p) => ({
    Order_Number: orderId,
    SKU: p.id,
    Qty: 10,
  })),
);

console.log(`\nCreated order ${orderId} with ${needsRefill.length} line items.`);
```

A few things worth noticing:

- The `filter` argument is fully typed against the schema. Try misspelling `"OUT OF STOCK"` — TypeScript will reject it, because `Stock_Alert`'s allowed values are baked into the generated schema.
- `In_Stock` and `Stock_Alert` are formula columns. The schema marks them as such, so they don't appear in the `insert()` payload type — you can't accidentally try to write them.
- `Order_Date` is a Grist `Date`, encoded as epoch seconds.
- Every Grist row has a numeric `id`. `Order_Number` is a `Ref` to `Incoming_Orders`, so it expects that row id.

11.3 Run it

Test against the public template (read-only, no API key needed):

```
node --env-file=.env refill.ts --dry-run
```

To actually place the order, make a copy of the template doc under your own account, generate an API key in **Profile Settings** → **API**, and update `.env`:

```
GRIST_DOC_URL=https://<your-host>/api/docs/<your-doc-id>
GRIST_API_KEY=<your-api-key>
```

Then run:

```
node --env-file=.env refill.ts
```

Open your doc — there's a new row in **Incoming Orders** with line items for each low-stock product.