Datasette Enrichments

Release 0.4.1-3-gad61b35

Simon Willison

Apr 27, 2024

CONTENTS

Datasette Enrichments is a plugin for Datasette that adds support for enriching data in different ways.

An **enrichment** is a bulk operation that can by applied to a set of rows from a table, executing code for each of those rows.

Potential use-cases for enrichments include:

- Geocoding an address and populating a latitude and longitude column
- Executing a template to generate output based on the values in each row
- Fetching data from a URL and populating a column with the result
- Executing OCR against a linked image or PDF file

Each enrichment is implemented as its own plugin.

The Datasette ecosystem includes a growing number of enrichment plugins. They are also intended to be easy to write.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation and setup

To install the Datasette Enrichments plugin, run this:

```
datasette install datasette-enrichments
```

You need to install additional plugins for enrichments that you want to use before this plugin becomes useful. Try datasette-enrichments-jinja for example:

```
datasette install datasette-enrichments-jinja
```

Users with the `enrichments` permission (or the `--root` user) will then be able to select rows for enrichment using the cog actions menu on the table page.

Once you have installed an enrichment you can *run it against some data*.

1.2 Running an enrichment

Enrichments are run against data in a Datasette table.

They can be executed against every row in a table, or you can filter that table first and then run the enrichment against the filtered results.

Start on the table page and filter it to just the set of rows that you want to enrich. Then click the cog icon near the table heading and select "Enrich selected data".

This will present a list of available enrichments, provided by plugins that have been installed.

Select the enrichment you want to run to see a form with options to configure for that enrichment.

Enter your settings and click "Enrich data" to start the enrichment running.

Enrichments can take between several seconds and several minutes to run, depending on the number of rows and the complexity of the enrichment.

1.3 Configuring permissions

Enrichments are only available to users with the `enrichments` permission.

By default the `root` user has this permission. You can execute Datasette like this to sign in with that user:

```
datasette mydb.db --root
```

Then click the link provided to sign in as root.

To use enrichments in a deployed instance of Datasette, you can use an authentication plugin such as datasette-auth-github or datasette-auth-passwords to authenticate users, then grant the `enrichments` permission to those users using permissions in datasette.yaml, or one of the permissions plugins.

1.4 Developing a new enrichment

Enrichments are implemented as Datasette plugins.

An enrichment plugin should implement the `register_enrichments()` plugin hook, which should return a list of instances of subclasses of the `Enrichment` base class.

The function can also return an awaitable function which returns that list of instances. This is useful if your plugin needs to do some asynchronous work before it can return the list of enrichments.

1.4.1 The plugin hook

Your enrichment plugin should register new enrichments using the `register_enrichments()` plugin hook:

```
from datasette import hookimpl

@hookimpl
def register_enrichments():
    return [MyEnrichment()]
```

`register_enrichment()` can optionally accept a `datasette` argument. This can then be used to read plugin configuration or run database queries.

The plugin hook can return an awaitable function if it needs to do some asynchronous work before it can return the list of enrichments, for example:

```
@hookimpl
def register_enrichments(datasette):
    async def inner():
        db = datasette.get_database("mydb")
        settings = [
            row["setting"]
            for row in await db.execute(
                "select setting from special_settings"
            )
        ]
        return [
            MyEnrichment(setting)
            for setting in settings
```

(continues on next page)

(continued from previous page)

```
        ]
    return inner
```

1.4.2 Enrichment subclasses

Most of the code you write will be in a subclass of `Enrichment`:

```
from datasette_enrichments import Enrichment

class MyEnrichment(Enrichment):
    name = "Name of My Enrichment"
    slug = "my-enrichment"
    description = "One line description of what it does"
```

The `name`, `slug` and `description` attributes are required. They are used to display information about the enrichment to users.

Try to ensure your `slug` is unique among all of the other enrichments your users might have installed.

You can also set a `batch_size` attribute. This defaults to 100 but you can set it to another value to control how many rows are passed to your `enrich_batch()` method at a time. You may want to set it to 1 to process rows one at a time.

initialize()

Your class can optionally implement an `initialize()` method. This will be called once at the start of each enrichment run.

This method is often used to prepare the database - for example, adding a new table column that the enrichment will then populate.

```
async def initialize(
    self,
    datasette: Datasette,
    db: Database,
    table: str,
    config: dict
):
```

The named parameters passed to `initialize()` are all optional. If you declare them they will be passed as follows:

- `datasette` is the Datasette instance.
- `db` is the Database instance for the database that the enrichment is being run against.
- `table` is the name of the table.
- `config` is a dictionary of configuration options that the user set for the enrichment, using the configuration form (if one was provided).

enrich_batch()

You must implement the following method:

```
async def enrich_batch(
    self,
    datasette: Datasette,
    db: Database,
    table: str,
    rows: List[dict],
    pks: List[str],
    config: dict,
    job_id: int,
):
    # Enrichment logic goes here
```

Again, you can use just the subset of the named parameters that you need.

This method will be called multiple times, each time with a different list of rows.

It should perform whatever enrichment logic is required, using the `db` object (documented here) to write any results back to the database.

`enrich_batch()` is an `async def` method, so you can use `await` within the method to perform asynchronous operations such as HTTP calls (using HTTPX) or database queries.

The parameters available to `enrich_batch()` are as follows:

- `datasette` is the Datasette instance. You can use this to read plugin configuration, check permissions, render templates and more.
- `db` is the Database instance for the database that the enrichment is being run against. You can use this to execute SQL queries against the database.
- `table` is the name of the table that the enrichment is being run against.
- `rows` is a list of dictionaries for the current batch, each representing a row from the table. These are the same shape as JSON dictionaries returned by the Datasette JSON API. The batch size defaults to 100 but can be customized by your class.
- `pks` is a list of primary key column names for the table.
- `config` is a dictionary of configuration options that the user set for the enrichment, using the configuration form (if one was provided).
- `job_id` is a unique integer ID for the current job. This can be used to log additional information about the enrichment execution.

get_config_form()

The `get_config_form()` method can optionally be implemented to return a WTForms form class that the user can use to configure the enrichment.

This example defines a form with two fields: a `template` text area field and an `output_column` single line input:

```
from wtforms import Form, StringField, TextAreaField
from wtforms.validators import DataRequired

# ...
```

(continues on next page)

(continued from previous page)

```
    async def get_config_form(self):
        class ConfigForm(Form):
            template = TextAreaField(
                "Template",
                description='Template to use',
                default=default,
            )
            output_column = StringField(
                "Output column name",
                description="The column to store the output in - will be created if it␣
↪does not exist.",
                validators=[DataRequired(message="Column is required.")],
                default="template_output",
            )
        return ConfigForm
```

The valid dictionary that is produced by filling in this form will be passed as `config` to both the `initialize()` and `enrich_batch()` methods.

The `get_config_form()` method can take the following optional named parameters:

- `datasette`: a Datasette instance
- `db`: a Database instance
- `table`: the string name of the table that the enrichment is being run against

finalize()

Your class can optionally implement a `finalize()` method. This will be called once at the end of each enrichment run.

```
async def finalize(self, datasette, db, table, config):
    # ...
```

Again, these named parameters are all optional:

- `datasette` is the [Datasette instance]
- `db` is the Database instance
- `table` is the name of the table (a string)
- `config` is an optional dictionary of configuration options that the user set for the run

1.4.3 Tracking errors

Errors that occur while running an enrichment are recorded in the `_enrichment_errors` table, with the following schema:

```
create table _enrichment_errors (
    id integer primary key,
    job_id integer references _enrichment_jobs(id),
    created_at text,
    row_pks text, -- JSON list of row primary keys
```

(continues on next page)

(continued from previous page)

```
    error text
)
```

If your `.enrich_batch()` raises any exception, all of the IDs in that batch will be marked as errored in this table.

Alternatively you can catch errors for individual rows within your `enrich_batch()` method and record them yourself using the `await self.log_error()` method, which has the following signature:

```
async def log_error(
    self, db: Database, job_id: int, ids: List[IdType], error: str
)
```

Call this with a reference to the current database, the job ID, a list of row IDs (which can be strings, integers or tuples for compound primary key tables) and the error message string.

If you set `log_traceback = True` on your `Enrichment` class a full stacktrace for the most recent exception will be recorded in the database table in addition to the string error message. This is useful during plugin development:

```
class MyEnrichment(Enrichment):
    ...
    log_traceback = True
```

1.4.4 Enrichments that use secrets such as API keys

Enrichments often need to access external APIs, for things like geocoding addresses or accessing Large Language Models. These APIs frequently need some kind of API key.

The enrichments system can work in combination with the datasette-secrets plugin to manage secret API keys needed for different services.

Examples of enrichments that use this mechanism include datasette-enrichments-opencage and datasette-enrichments-gpt.

To define a secret that your plugin needs, add the following code:

```
from datasette_enrichments import Enrichment
from datasette_secrets import Secret

# Then later in your enrichments class
class TrainEnthusiastsEnrichment(Enrichment):
    name = "Train Enthusiasts"
    slug = "train-enthusiasts"
    description = "Enrich with extra data from the Train Enthusiasts API"
    secret = Secret(
        name="TRAIN_ENTHUSIASTS_API_KEY",
        description="An API key from train-enthusiasts.doesnt.exist",
        obtain_url="https://train-enthusiasts.doesnt.exist/api-keys",
        obtain_label="Get an API key"
    )
```

Configuring your enrichment like this will result in the following behavior:

- If a user sets a `DATASETTE_SECRETS_TRAIN_ENTHUSIASTS_API_KEY` environment variable, that value will be used as the API key. You should mention this in your enrichment's documentation.

- Otherwise, if the user has configured datasette-secrets to store secrets in the database, admin users with the `manage-secrets` permission will get the option to set that secret as part of the Datasette web UI.
- If neither of the above are true, any time any user tries to run this enrichment they will be prompted to specify a secret which will be used for that run but will not be stored outside of memory used by the code that runs the enrichment.

With a secret configured in this way, your various class methods such as `initialize()`, `enrich_batch()` and `finalize()` can access the secret value using `await self.get_secret(datasette)` like this:

```
api_key = await self.get_secret(datasette, config)
```

You must pass both the `datasette` and the `config` arguments that were passed to those methods.

1.4.5 Writing tests for enrichments

Take a look at the test suite for datasette-enrichments-opencage for an example of how to test an enrichment.

You can use the `datasette_enrichments.wait_for_job()` utility function to avoid having to run a polling loop in your tests to wait for the enrichment to complete.

Here's an example test for an enrichment, using `datasette.client.post()` to start the enrichment running:

```
from datasette.app import Datasette
from datasette_enrichments.utils import wait_for_job
import pytest
import sqlite3

@pytest.mark.asyncio
async def test_enrichment(tmpdir):
    db_path = str(tmpdir / "demo.db")
    datasette = Datasette([db_path])
    db = sqlite3.connect(db_path)
    with db:
        db.execute("create table if not exists news (body text)")
        for text in ("example a", "example b", "example c"):
            db.execute("insert into news (body) values (?)", [text])

    # Obtain ds_actor and ds_csrftoken cookies
    cookies = {"ds_actor": datasette.sign({"a": {"id": "root"}}, "actor")}
    csrftoken = (
        await datasette.client.get(
            "/-/enrich/demo/news/name-of-enrichment",
            cookies=cookies
        )
    ).cookies["ds_csrftoken"]
    cookies["ds_csrftoken"] = csrftoken

    # Execute the enrichment
    response = await datasette.client.post(
        "/-/enrich/demo/news/name-of-enrichment",
        data={
            "source_column": "body",
            "csrftoken": cookies["ds_csrftoken"],
        },
```

(continues on next page)

(continued from previous page)

```
        cookies=cookies,
    )
    assert response.status_code == 302

    # Get the job_id so we can wait for it to complete
    job_id = response.headers["Location"].split("=")[-1]
    await wait_for_job(datasette, job_id, database="demo", timeout=1)
    db = datasette.get_database("demo")
    jobs = await db.execute("select * from _enrichment_jobs")
    job = dict(jobs.first())
    assert job["status"] == "finished"
    assert job["enrichment"] == "name-of-enrichment"
    assert job["done_count"] == 3
    results = await db.execute("select * from news order by body")
    rows = [dict(r) for r in results.rows]
    assert rows == [
        {"body": "example a transformed"},
        {"body": "example b transformed"},
        {"body": "example c transformed"}
    ]
```

The full signature for `wait_for_job()` is:

```
async def wait_for_job(
    datasette: Datasette,
    job_id: int,
    database: Optional[str] = None,
    timeout: Optional[int] = None,
):
```

The `timeout` argument to `wait_for_job()` is optional - this will cause the test to fail if the job does not complete within the specified number of seconds.

If you omit the `database` name the single first database will be used, which is often the right thing to do for tests.