Project Overview

Starlight started life as a local Django project running with `manage.py runserver` and a simple SQLite database. The assignment was to turn this into something that looks like a real production deployment: containerized, running in the cloud, with CI/CD, a separate database service, secrets handled properly, and basic monitoring.

I packaged the app into a Docker image, pushed it to Azure Container Registry (ACR), and deployed it to Azure Container Apps. A GitHub Actions workflow now builds and deploys a new version whenever I push to `main`, so the app is no longer "just local," it's actually hosted and reproducible.

The original plan was to have two containers in Azure Container Apps: one running Django and one running PostgreSQL, with Django talking to Postgres over the internal network. I did succeed in provisioning the Postgres Container App and wiring up its storage, but in the end I could not get Django to connect reliably to it from within Container Apps. After a lot of timed-out connections, I decided to keep the app on SQLite in production so that the site is stable and usable. The Postgres container is still there and working on its own, but it is not the live database (yet).

The app is deployed at:

https://starlight-webapp.thankfulbush-e9327f34.westeurope.azurecontainerapps.io/

and updates automatically via GitHub Actions when I push new code.

Architecture

The final architecture has a few main pieces that work together:

- GitHub hosts the code and runs CI/CD workflows.
- Azure provides a resource group `BCSAI2025-DEVOPS-STUDENTS-B`, an Azure Container Apps environment called `starlight-env`, and an Azure Container Registry at `starlightsofiia.azurecr.io`.
- The web application runs as a Container App called `starlight-webapp`. It runs my Docker image, exposes port 8000 internally, and is fronted by HTTPS at the public URL shown above.
- A separate Container App called `starlight-db` runs a `postgres:16` container with its data directory backed by an Azure File share so that it would persist across restarts.

Inside the web container, Django is configured to look at database settings from environment variables. In `satellite_tracker/settings.py` I read `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_HOST`, and `POSTGRES_PORT`. If they are all present, Django configures `DATABASES` to use `django.db.backends.postgresql`. If any of them are missing, it falls back to the original SQLite configuration and uses a local `db.sqlite3` file.

Right now, in the Container App configuration, I am not injecting the `POSTGRES_` environment variables, so Django chooses SQLite. This is intentional: whenever I tried to point it at the Postgres Container App host, the connection timed out from inside the web container. As soon as I turned those env vars off again, migrations and catalog pages became fast and stable.

Aside from database choice, the rest of the runtime environment is more "production-like" than the original project. Gunicorn runs the app instead of `runserver`, static files are collected at build time and served by Whitenoise, secrets such as `SECRET_KEY` and `DJANGO_ALLOWED_HOSTS` come from environment variables, and `DEBUG` is set to `0` in production.

Configuration and Code Quality

One of the biggest improvements compared to the earlier version is how configuration is handled. Previously a lot of things were basically magic constants scattered through the project: the secret key, allowed hosts, debug flags, sometimes even URLs. Now almost all runtime configuration flows through environment variables, which are then read in `satellite_tracker/settings.py`.

For example, `SECRET_KEY`, `DEBUG`, `DJANGO_ALLOWED_HOSTS`, and `DJANGO_CSRF_TRUSTED_ORIGINS` all come from environment variables with sensible defaults for local development. That means I can run the same code locally and in Azure without editing settings between environments. In production, the Container App injects the real values (taken from GitHub Secrets), and in local dev I just use a `.env` or export them in the shell.

This approach lines up nicely with the idea of single responsibility and the "open/closed" principle: the code that implements the app doesn't need to change when the environment changes, it just reads different configuration. I also kept modules focused: views handle web logic, services (like `satellites.services.catalog` and tracking helpers) encapsulate domain logic, and cross-cutting concerns like metrics are handled by middleware.

Health checking and metrics are also separated out. I added a dedicated `health_status` view that returns a JSON object like `{"app": "ok", "database": "ok"}` when everything works, or a degraded status with HTTP 503 if the database check fails. On top of that, I integrated `django-prometheus` so that instrumentation lives in middleware instead of being scattered in every view. That keeps request handlers focused on what they should do, instead of manually tracking timings or counters.

Overall, moving configuration to env vars, splitting responsibilities into smaller modules, and using middleware for cross-cutting concerns removed a lot of "code smells" from the old version and made the project much easier to understand and to test.

Testing

I also worked on expanding the test suite beyond simple smoke tests. At the service level, there are unit tests for catalog operations, tracking logic, TLE fetching, propagation math, and the favorites serialization. These tests don't care about HTTP details; they just make sure the pure Python logic behaves correctly.

On top of that, there are integration-style tests that hit the REST API endpoints directly, such as the favorites API, the single-satellite position endpoint, and the batch positions endpoint. Those tests go through Django's request/response cycle and catch issues at the view layer.

The signup flow is also tested end-to-end, to make sure that user creation, redirects, and form validation work. I added a test for the management command that imports the TLE catalog, so that if the import process breaks (for example, because of a change in input format), it doesn't silently fail.

All tests run through `coverage run manage.py test`. I configured coverage to require at least 70% line coverage (the course requirement), and the latest run shows about 82.63% coverage. The coverage artifacts go into a `reports/` folder: the raw `.coverage` file, `coverage.xml`, and an HTML report under `reports/coverage_html/`.

This matters because the CI workflow in GitHub Actions uses the exact same coverage command. If I push something that reduces coverage below 70% or breaks a test, the workflow fails and blocks the change. So the tests aren't just there "locally," they are enforced by the pipeline.

CI Pipeline

For continuous integration I set up a workflow in `.github/workflows/ci.yml`. Whenever I push or create a pull request against `main`, GitHub:

1. Spins up a fresh Ubuntu runner.

2. Installs Python 3.11 and the project requirements, plus `coverage`.

3. Runs `coverage run manage.py test`.

Because I configured coverage with `fail_under = 70`, the command fails automatically if line coverage drops below that threshold. After the tests pass, the workflow produces coverage reports under `reports/` and uploads them as build artifacts, so I can download them and inspect coverage in HTML form if I want.

As a final step, the CI workflow builds the Docker image locally on the runner with `docker build -t starlight-app .`. This is a small but important check: it confirms that the Dockerfile works and the image is buildable in a clean environment, not just on my laptop.

If any step fails-tests, coverage, or Docker build-GitHub marks the run as failed. That ties CI directly into the development process: no change gets merged into `main` unless it passes tests, keeps coverage above the limit, and still builds as a container.

CD Pipeline and Runtime Behaviour

The continuous deployment side lives in `.github/workflows/cd.yml` and triggers on pushes to `main`. Once the code reaches that branch, the CD workflow:

- Logs in to Azure using `azure/login@v2` and the `AZURE_CREDENTIALS` service principal JSON.
- Logs in to ACR using `az acr login` and the registry name stored in `AZURE_CONTAINER_REGISTRY`.
- Builds the Docker image using the full login server from `AZURE_CONTAINER_REGISTRY_LOGIN_SERVER` (for example `starlightsofiia.azurecr.io`), tags it with both the commit SHA and `latest`, and pushes both tags.
- Makes sure the `containerapp` extension is installed in the Azure CLI
- Deploys the image to the `starlight-webapp` Container App.

If the app already exists, the workflow calls `az containerapp registry set` to wire ACR credentials to the app, then runs `az containerapp update` to switch the image and update environment variables like `SECRET_KEY`, `DJANGO_ALLOWED_HOSTS`, `DEBUG`, and some Python flags. If it doesn't exist, it runs `az containerapp create` with similar parameters, plus ingress setup (external HTTPS on port 8000) and scaling limits.

At an earlier stage I also injected `POSTGRES_HOST`, `POSTGRES_DB`, `POSTGRES_USER`, and `POSTGRES_PASSWORD` into the app from GitHub Secrets. After that, the container started timing out whenever it tried to connect to the Postgres Container App, and running `manage.py migrate` via `az containerapp exec` ended with `OperationalError:

connection timed out` to the Postgres FQDN. Gunicorn workers would then hit their timeout and be killed.

To stabilise things, I removed the Postgres env vars from the deployment and let Django fall back to SQLite. Once I did that, migrations ran instantly and catalog pages no longer hung. So the CD pipeline still exists and works, but in the current configuration it's deploying a web app that uses SQLite, not Postgres.

Monitoring and Observability

To handle monitoring, I focused on two main parts: health checks and metrics.

First, I added a `/health/` endpoint in Django that returns a JSON payload such as `{"app": "ok", "database": "ok"}` when the app is functioning normally. Internally, it does a simple database check; if the DB is unreachable or misconfigured, it returns HTTP 503 and a degraded status like `{"app": "degraded", "database": "error"}`. This is the kind of endpoint that could be wired into Azure Container Apps as a readiness or liveness probe.

Second, I integrated `django-prometheus` and exposed a `/metrics` endpoint. That endpoint returns Prometheus-formatted metrics, including counters and histograms for HTTP requests, response latencies, and database usage. To prove that this works beyond just "it starts", I ran a Prometheus container locally and configured it to scrape `https://starlight-webapp.thankfulbush-e9327f34.westeurope.azurecontainerapps.io/metrics`. In the Prometheus UI I could see the target become healthy and query metrics like `django_http_requests_total_by_method`.

I also ran Grafana in a local Docker container, added the Prometheus instance as a data source, and built a small dashboard that graphs request metrics for the Starlight app. This shows an end-to-end monitoring pipeline: Azure hosts the app and exposes `/metrics`, Prometheus scrapes that endpoint, and Grafana visualises the collected data. Screenshots of the `/metrics` output, the Prometheus target page, and the Grafana dashboard will be included in the appendix as evidence.

On the Azure side, the Container Apps environment also sends stdout/stderr logs to a Log Analytics workspace. That's how I could see gunicorn logs (worker timeouts, restarts) and Postgres startup logs when debugging.

You can find the screenshots that prove the monitoring attached in the appendix.

Problems and Unresolved Issues

I ran into a lot of issues on the way, and most of them got solved: Azure provider registration permissions, Terraform committing huge provider binaries, ACR authentication, mismatched CLI flags, missing Gunicorn, and CSRF/allowed hosts problems. One thing I could not fully resolve, and that is important to call out, is the network connectivity between the web Container App and the Postgres Container App.

Even though the Postgres container (`starlight-db`) starts fine, logs "database system is ready to accept connections", and listens on port 5432, every attempt from the web app to connect to the internal FQDN on port 5432 resulted in a timeout from inside the Container Apps environment. That broke migrations and caused gunicorn workers to hang until they were killed. Because I couldn't fix that within the constraints of the project and time, I made the trade-off to keep the Postgres container deployed but not use it, and stick with SQLite for the live app so that everything else works correctly.

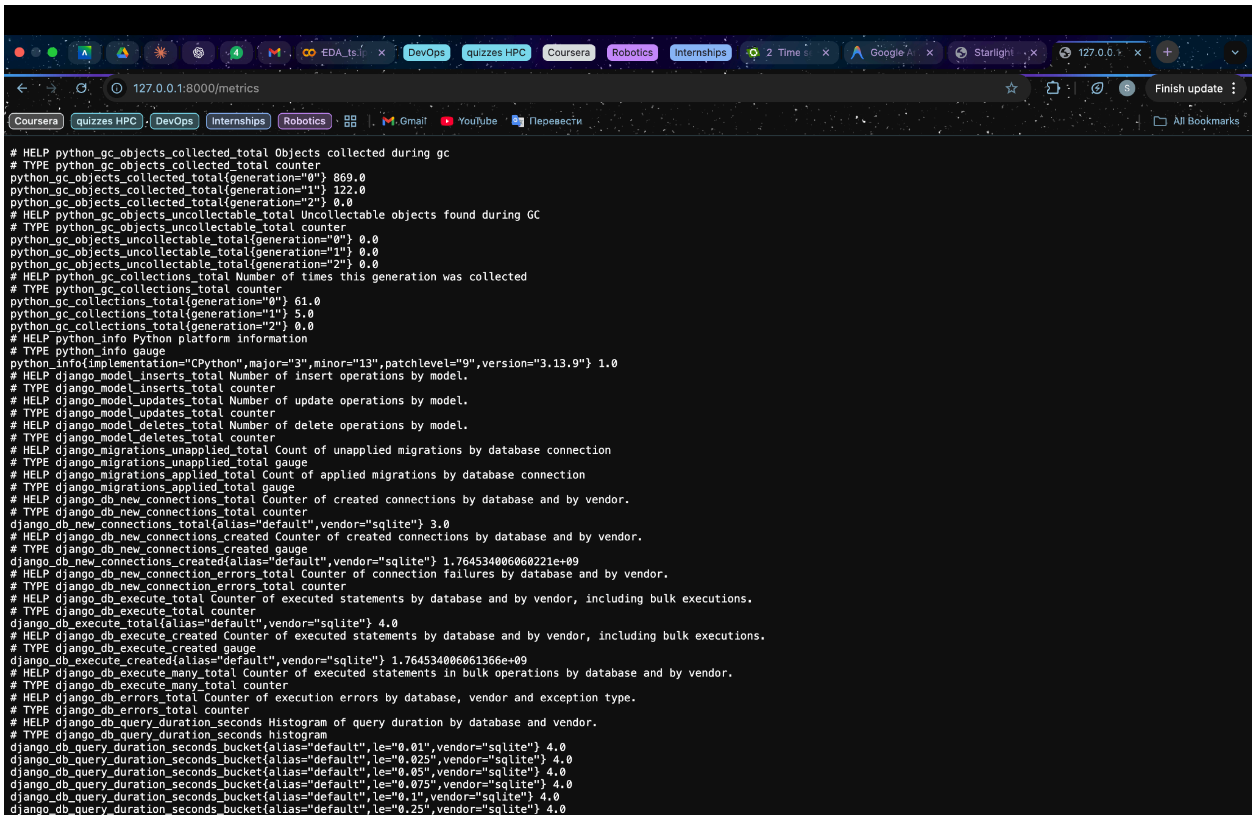
This is why in the final version the architecture is "prepared for Postgres" but actually runs on SQLite.

Conclusion and Future Work

To summarise, Starlight has been moved from a local Django + SQLite app into a cloud-hosted, containerised service with proper CI/CD, environment-based configuration, monitoring endpoints, and a decent test suite. The GitHub Actions CI workflow enforces tests and coverage, the CD workflow builds and deploys Docker images to Azure Container Apps, Gunicorn serves the app, and Whitenoise handles static files. Secrets are no longer hard-coded but flow through environment variables wired from GitHub Secrets and Azure configuration.

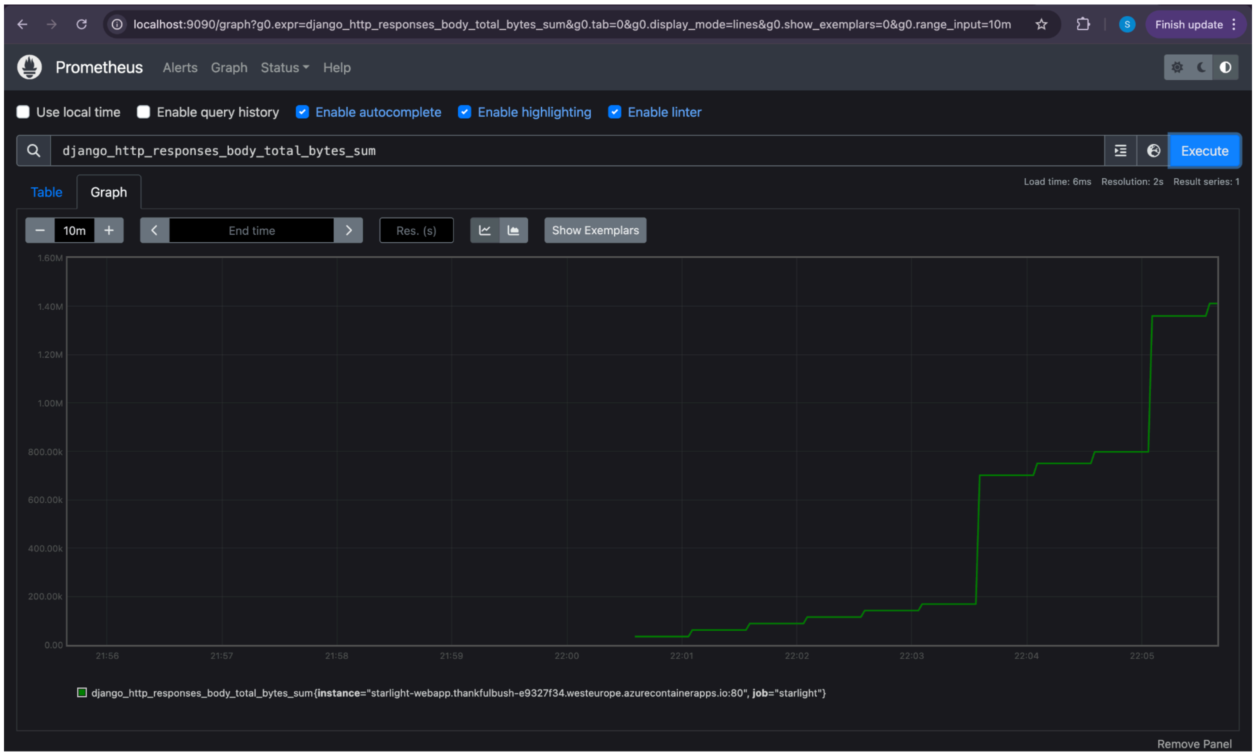
Appendix

Here you can find attached the screenshots of dashboards, Prometheus, and Grafana.



127.0.0.1:8000/metrics

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 869.0
python_gc_objects_collected_total{generation="1"} 122.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 61.0
python_gc_collections_total{generation="1"} 5.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="13",patchlevel="9",version="3.13.9"} 1.0
# HELP django_model_inserts_total Number of insert operations by model.
# TYPE django_model_inserts_total counter
# HELP django_model_updates_total Number of update operations by model.
# TYPE django_model_updates_total counter
# HELP django_model_deletes_total Number of delete operations by model.
# TYPE django_model_deletes_total counter
# HELP django_migrations_unapplied_total Count of unapplied migrations by database connection
# TYPE django_migrations_unapplied_total gauge
# HELP django_migrations_applied_total Count of applied migrations by database connection
# TYPE django_migrations_applied_total gauge
# HELP django_db_new_connections_total Counter of created connections by database and by vendor.
# TYPE django_db_new_connections_total counter
django_db_new_connections_total{alias="default",vendor="sqlite"} 3.0
# HELP django_db_new_connections_created Counter of created connections by database and by vendor.
# TYPE django_db_new_connections_created gauge
django_db_new_connections_created{alias="default",vendor="sqlite"} 1.764534006060221e+09
# HELP django_db_new_connection_errors_total Counter of connection failures by database and by vendor.
# TYPE django_db_new_connection_errors_total counter
# HELP django_db_execute_total Counter of executed statements by database and by vendor, including bulk executions.
# TYPE django_db_execute_total counter
django_db_execute_total{alias="default",vendor="sqlite"} 4.0
# HELP django_db_execute_created Counter of executed statements by database and by vendor, including bulk executions.
# TYPE django_db_execute_created gauge
django_db_execute_created{alias="default",vendor="sqlite"} 1.764534006061366e+09
# HELP django_db_execute_many_total Counter of executed statements in bulk operations by database and by vendor.
# TYPE django_db_execute_many_total counter
# HELP django_db_errors_total Counter of execution errors by database, vendor and exception type.
# TYPE django_db_errors_total counter
# HELP django_db_query_duration_seconds Histogram of query duration by database and vendor.
# TYPE django_db_query_duration_seconds histogram
django_db_query_duration_seconds_bucket{alias="default",le="0.01",vendor="sqlite"} 4.0
django_db_query_duration_seconds_bucket{alias="default",le="0.025",vendor="sqlite"} 4.0
django_db_query_duration_seconds_bucket{alias="default",le="0.05",vendor="sqlite"} 4.0
django_db_query_duration_seconds_bucket{alias="default",le="0.075",vendor="sqlite"} 4.0
django_db_query_duration_seconds_bucket{alias="default",le="0.1",vendor="sqlite"} 4.0
django_db_query_duration_seconds_bucket{alias="default",le="0.25",vendor="sqlite"} 4.0
```



localhost:9090/graph?g0.expr=django_http_responses_body_total_bytes_sum&g0.tab=0&g0.display_mode=lines&g0.show_exemplars=0&g0.range_input=10m

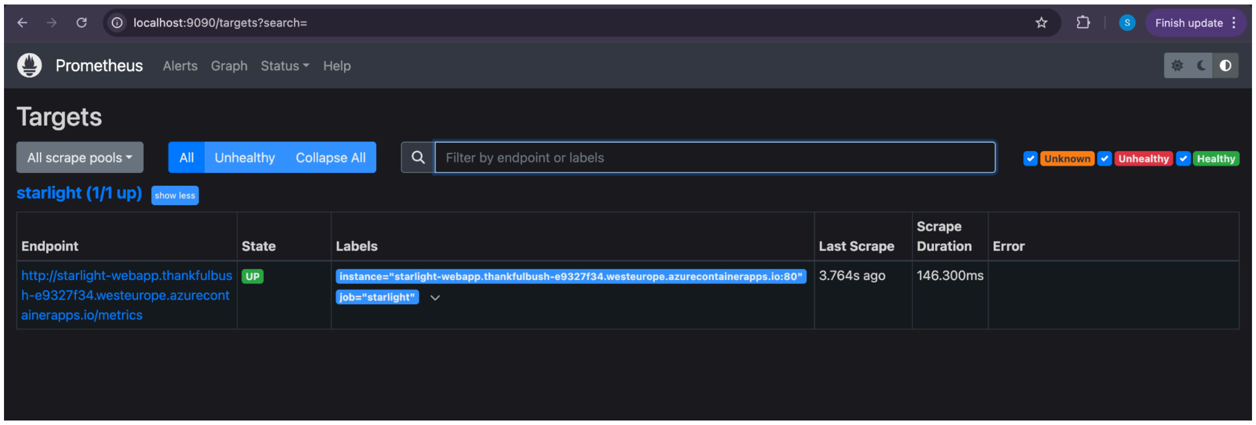
☐ Use local time ☐ Enable query history ✓ Enable autocomplete ✓ Enable highlighting ✓ Enable linter

```
django_http_responses_body_total_bytes_sum
```

Load time: 6ms Resolution: 2s Result series: 1

10m

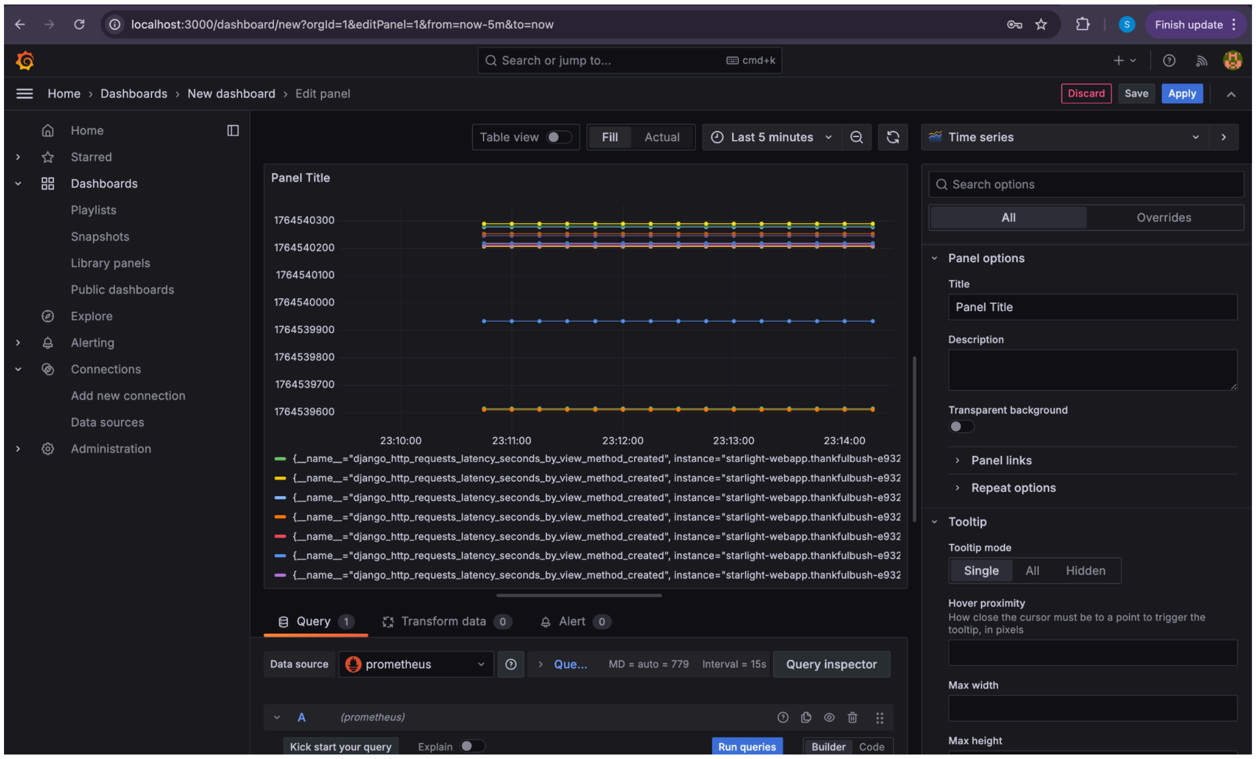
🟩 django_http_responses_body_total_bytes_sum{**instance**="starlight-webapp.thankfulbush-e9327f34.westeurope.azurecontainerapps.io:80", **job**="starlight"}


localhost:9090/targets?search=

Targets

starlight (1/1 up)

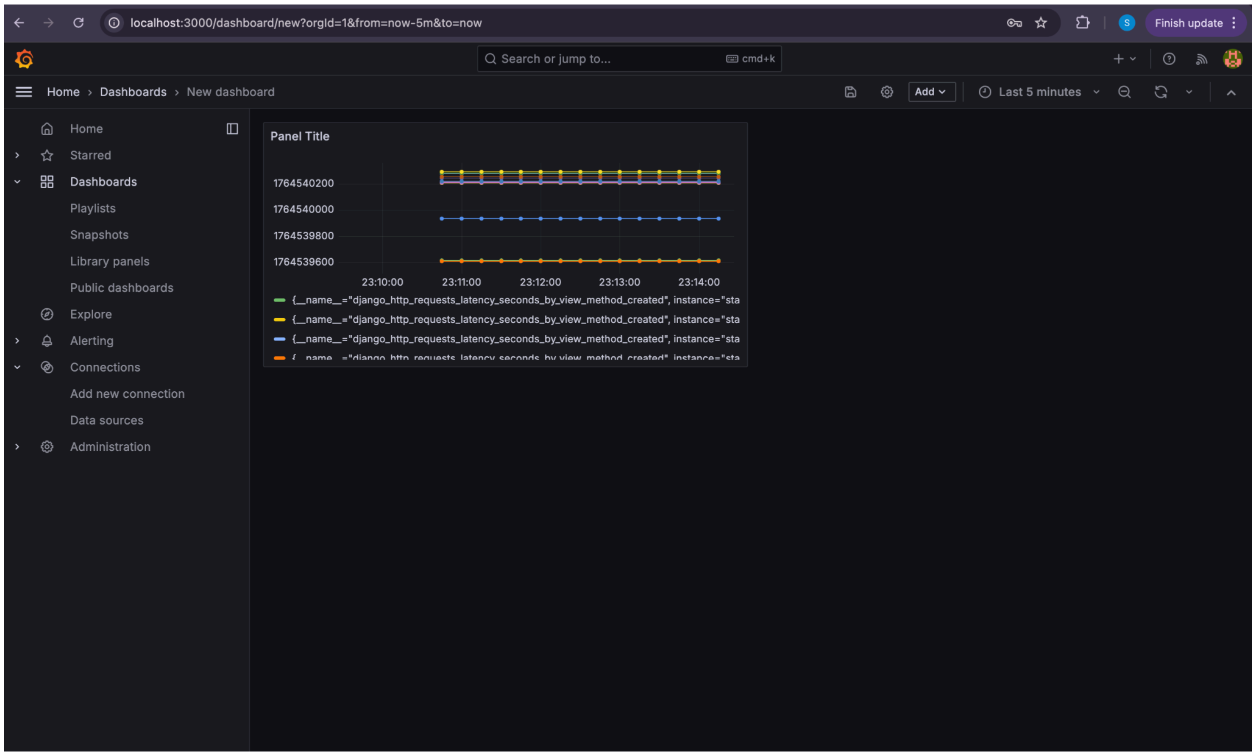
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://starlight-webapp.thankfulbush-e9327f34.westeurope.azurecontainerapps.io/metrics	UP	instance="starlight-webapp.thankfulbush-e9327f34.westeurope.azurecontainerapps.io:80" job="starlight"	3.764s ago	146.300ms	



```
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="starlight-webapp.thankfulbush-e932
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="starlight-webapp.thankfulbush-e932
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="starlight-webapp.thankfulbush-e932
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="starlight-webapp.thankfulbush-e932
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="starlight-webapp.thankfulbush-e932
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="starlight-webapp.thankfulbush-e932
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="starlight-webapp.thankfulbush-e932
```

Data source: prometheus
Que... MD = auto = 779 Interval = 15s
Query inspector
A (prometheus)

Title: Panel Title
Description:
Transparent background
Tooltip mode: Single
Hover proximity
How close the cursor must be to a point to trigger the tooltip, in pixels
Max width
Max height



```
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="sta
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="sta
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="sta
{__name__="django_http_requests_latency_seconds_by_view_method_created", instance="sta
```