**An Introduction to Orchestrating Data Assets with Dagster**



Dagster is an open-source data orchestrator: a framework for building and running data pipelines, similar to how PyTorch and TensorFlow are frameworks for building neural networks.

Dagster focuses on orchestrating assets, not just tasks. Data assets can be many things, but they’re usually machine learning models, tables in a data warehouse, or reports. In order to build a data asset, you basically need to do four things:

* Ingest data from external sources or other data assets.
* Combine and transform the data in a meaningful way.
* Store the asset in a place where it can be used
* Re-run this process incrementally whenever the asset is out of date — either on a schedule or when an external system triggers the run.

In this post, we’ll introduce Dagster and demonstrate how to use it to build a quick but realistic data pipeline.

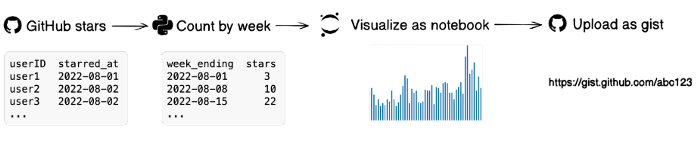
**How is Dagster different?**

Workflow engines like Airflow are also often used for building and running data pipelines. Compared to them, Dagster is different in three main ways:

* **Local development and testing.** Dagster was built from the ground up to make local development and automated testing easy through its emphasis on separating business logic from I/O concerns such as storage and interacting with external systems.
* **Software-defined assets (SDAs).** Dagster’s primary abstraction is the SDA: a declarative, pure Python function that computes the value of an asset and has associated metadata. Other orchestrators use imperative tasks as their primary abstraction, which is much more primitive on a number of dimensions:
* Engineers generally find the declarative mental model of SDAs much easier to work with.
* SDAs unambiguously document which assets are meant to exist.
* SDAs have clear, fine-grained data lineage that makes debugging and understanding the state of an asset easy.
* SDAs decouple the business logic for computing the asset’s value from the I/O logic to read to and write from storage ([docs](https://docs.dagster.io/concepts/io-management/io-managers))
* SDAs can be imported from any tool in your stack, so if you use an external tool like [dbt](https://www.getdbt.com/) that creates multiple tables in your data warehouse, Dagster can track the lineage of every individual table (other orchestrators will simply have a “black box” dbt task in the graph).
* SDAs support rich, searchable metadata and grouping tools to support scaling to large, complex organizations.
* SDAs support time partitioning and backfills out of the box.
* **Decoupling pipelines from the environment.** Dagster was built from the ground up to abstract away the environment from the business logic in your data pipeline, which leads to a number of elegant capabilities that are clunky or nonexistent in other orchestrators:
* Staging and Testing environments are much easier to set up by swapping out external services ([docs](https://docs.dagster.io/guides/dagster/transitioning-data-pipelines-from-development-to-production))
* The underlying runtime can be swapped out without changing any user code (see the docs on [run launchers](https://docs.dagster.io/deployment/run-launcher) and [executors](https://docs.dagster.io/deployment/executors) if you want the gritty details)
* Dagster was built with containers in mind from day 1, so you don’t have to deal with pip-hell managing conflicting Python environments in large projects ([docs](https://docs.dagster.io/concepts/repositories-workspaces/workspaces))

**Getting started with Dagster**

Let’s build a quick, realistic example that pulls some data from GitHub and visualizes it. This is an example of an[ETL pipeline](https://www.snowflake.com/guides/etl-pipeline).



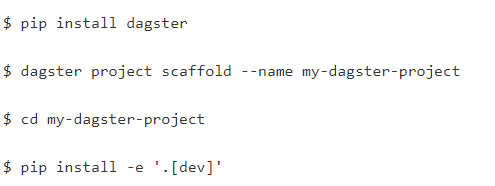
This guide assumes you have basic familiarity with Python and Python data tools like Jupyter and Pandas.

If you want to just see the code, it’s[available on GitHub](https://github.com/petehunt/dagster-github-stars-example).

**Installing Dagster**

⚠️ You may have to adapt these instructions depending on your environment. However, these instructions have been tested on[Gitpod](https://gitpod.io/#https://github.com/petehunt/empty), a free cloud development environment, which is the easiest way to get started.[Click here](https://gitpod.io/#https://github.com/petehunt/empty) to launch a fresh development environment.

Let’s start by[following the setup instructions](https://docs.dagster.io/getting-started/create-new-project#option-1-starting-with-the-default-project-skeleton). tl;dr:



**Installing the dependencies for this example**

For this tutorial, we’ll need to install a few dependencies. Modify your[setup.py](http://setup.py/) file to add the required dependencies:

if \_\_name\_\_ == "\_\_main\_\_":  
 setup(  
 name="my\_dagster\_project",  
 packages=find\_packages(exclude=["my\_dagster\_project\_tests"]),  
 install\_requires=[  
 "dagster",  
 "PyGithub",  
 "matplotlib",  
 "pandas",  
 "nbconvert",  
 "nbformat",  
 "ipykernel",  
 "jupytext",  
 ],  
 extras\_require={"dev": ["dagit", "pytest"]},  
 )

Once this is done, install by running pip install -e ‘.[dev]’ and restart dagit.

**Creating an asset for GitHub stars**

Before we begin, go to GitHub and[generate a personal access token](https://github.com/settings/tokens) with the gist permission. Then, let’s create an asset that fetches the GitHub stars for the Dagster repo by updating the my\_dagster\_project/assets/**init**.py file:

from dagster import asset  
from github import Github

ACCESS\_TOKEN = "ghp\_YOUR\_TOKEN\_HERE"@asset  
def github\_stargazers():  
 return   
list(Github(ACCESS\_TOKEN).get\_repo("dagster-io/dagster").get\_stargazers\_with\_dates())

🚨 There is obviously a big problem with this code: it includes a very sensitive secret right in the source, and the token has broad permissions. Don’t do this in production!

**Aggregate the GitHub stars by week**

Let’s add a second asset that aggregates the raw stargazers data into a weekly count and stores it in a pandas.DataFrame. Let’s add some more code to my\_dagster\_project/assets/**init**.py:

import pandas as pd  
from datetime import timedelta

@asset  
def github\_stargazers\_by\_week(github\_stargazers):  
 df = pd.DataFrame(  
 [  
 {  
 "users": stargazer.user.login,  
 "week": stargazer.starred\_at.date()  
 + timedelta(days=6 - stargazer.starred\_at.weekday()),  
 }  
 for stargazer in github\_stargazers  
 ]  
 )  
 return df.groupby("week").count().sort\_values(by="week")

Most of this code is just data transformation using pandas; see[the pandas docs](https://pandas.pydata.org/docs/index.html) for more information.

Notice that this asset takes an argument called github\_stargazers. Dagster will automatically find the asset named github\_stargazers and materialize it before calling github\_stargazers\_by\_week. This might seem like magic at first, but it’s very easy to get used to, and extremely convenient when you’re building large pipelines.

**Visualize the GitHub stars**

Now that we have a dataset of GitHub stars per week, let’s visualize it as a bar chart.[Jupyter Notebooks](https://jupyter.org/) are a great tool for this. We’ll use a neat library called[jupytext](https://github.com/mwouts/jupytext) which lets us author notebooks as Markdown strings instead of using raw .ipynb files. Add the following to my\_dagster\_project/assets/**init**.py to create an asset representing the notebook:

import nbformat  
 from nbconvert.preprocessors import ExecutePreprocessor  
 import pickle  
 import jupytext

@asset  
 def github\_stars\_notebook(github\_stargazers\_by\_week):  
 markdown = f"""  
 # Github Stars ```python  
 import pickle  
 github\_stargazers\_by\_week = pickle.loads({pickle.dumps(github\_stargazers\_by\_week)!r})  
 ```  
 ## Github Stars by Week, last 52 weeks  
 ```python github\_stargazers\_by\_week.tail(52).reset\_index().plot.bar(x="week", y="users")  
 ```  
 """  
 nb = jupytext.reads(markdown, "md")  
 ExecutePreprocessor().preprocess(nb)  
 return nbformat.writes(nb)

There are a few things going on here.

* We create a markdown string representing our notebook.
* We use pickle to pass the DataFrame to the notebook.
* We use pandas to plot the last 52 weeks as a bar chart.
* We use jupytext to convert the markdown string to a Jupyter NotebookNode
* We use ExecutePreprocessor().preprocess() to execute the notebook in a new kernel
* And we use nbformat.writes() to write out the NotebookNode as ipynb file contents.

**Share the notebook as a GitHub gist**

Now we have a notebook. How can we view it?

One easy way is to upload the ipynb as a GitHub gist. GitHub has built-in support for visualizing notebooks, and they’re very easy to share with stakeholders. Update my\_dagster\_project/assets/**init**.py with the following:

from github import InputFileContent

@asset  
def github\_stars\_notebook\_gist(context, github\_stars\_notebook):  
 gist = (  
 Github(ACCESS\_TOKEN)  
 .get\_user()  
 .create\_gist(  
 public=False,  
 files={  
 "github\_stars.ipynb": InputFileContent(github\_stars\_notebook),  
 },  
 )  
 )  
 context.log.info(f"Notebook created at {gist.html\_url}")  
 return gist.html\_url

This is a fairly straightforward asset that simply takes the github\_stars\_notebook asset contents, attaches it to a new GitHub gist, and returns the URL.

Note the context argument. This is a special argument that does not correspond to the name of an asset. It contains various useful pieces of information and utilities, including context.log — the primary way to log information to the user in Dagster. Read[the docs](https://docs.dagster.io/_apidocs/execution#dagster.OpExecutionContext) for more information.

**Adding a schedule**

Finally, let’s be sure that we refresh the notebook every day, so we always have the latest numbers. We can use[Schedules](https://docs.dagster.io/concepts/partitions-schedules-sensors/schedules) to do this.

Update your my\_dagster\_project/repository.py file to read:

from dagster import (  
 load\_assets\_from\_package\_module,  
 repository,  
 define\_asset\_job,  
 ScheduleDefinition,  
)  
from my\_dagster\_project import assets

daily\_job = define\_asset\_job(name="daily\_refresh", selection="\*")  
daily\_schedule = ScheduleDefinition(  
 job=daily\_job,  
 cron\_schedule="@daily",  
)@repository  
def my\_dagster\_project():  
 return [  
 daily\_job,  
 daily\_schedule,  
 load\_assets\_from\_package\_module(assets),  
 ]

We define two new entities:

* daily\_job is a Dagster [job](https://docs.dagster.io/concepts/ops-jobs-graphs/jobs) that materializes all of the assets in the project.
* daily\_schedule runs daily\_job once a day

Finally, we add them to our Dagster[repository](https://docs.dagster.io/concepts/repositories-workspaces/repositories) (which is just Dagster’s word for “project”).

At this stage, your my\_dagster\_project/assets/**init**.py should contain the following and your my\_dagster\_project/repository.py file should be as per the code shown in the prior paragraph.

from dagster import asset  
from github import Github

import pandas as pd  
from datetime import timedeltaimport nbformat  
from nbconvert.preprocessors import ExecutePreprocessor  
import pickle  
import jupytextfrom github import InputFileContentACCESS\_TOKEN = "ghp\_YOUR\_ACCESS\_TOKEN"@asset  
def github\_stargazers():  
 return list(Github(ACCESS\_TOKEN).get\_repo("dagster-io/dagster").get\_stargazers\_with\_dates())@asset  
def github\_stargazers\_by\_week(github\_stargazers):  
 df = pd.DataFrame(  
 [  
 {  
 "users": stargazer.user.login,  
 "week": stargazer.starred\_at.date()  
 + timedelta(days=6 - stargazer.starred\_at.weekday()),  
 }  
 for stargazer in github\_stargazers  
 ]  
 )  
 return df.groupby("week").count().sort\_values(by="week")@asset  
def github\_stars\_notebook(github\_stargazers\_by\_week):  
 markdown = f"""  
# Github Stars ```python  
 import pickle  
 github\_stargazers\_by\_week = pickle.loads({pickle.dumps(github\_stargazers\_by\_week)!r})  
 ```## Github Stars by Week, last 52 weeks  
 ```python github\_stargazers\_by\_week.tail(52).reset\_index().plot.bar(x="week", y="users")  
 ```  
 """  
 nb = jupytext.reads(markdown, "md")  
 ExecutePreprocessor().preprocess(nb)  
 return nbformat.writes(nb)@asset  
def github\_stars\_notebook\_gist(context, github\_stars\_notebook):  
 gist = (  
 Github(ACCESS\_TOKEN)  
 .get\_user()  
 .create\_gist(  
 public=False,  
 files={  
 "github\_stars.ipynb": InputFileContent(github\_stars\_notebook),  
 },  
 )  
 )  
 context.log.info(f"Notebook created at {gist.html\_url}")  
 return gist.html\_url

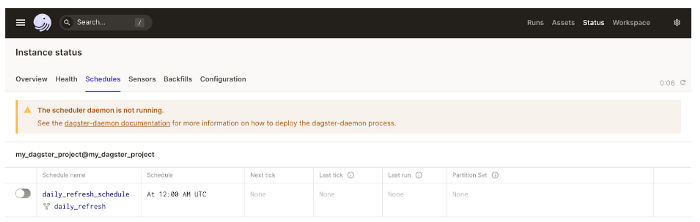
**Actually run the job**

Now it’s time to run the job with Dagster. First, launch the Dagster UI (called dagit) at[http://localhost:3000](http://localhost:3000/).

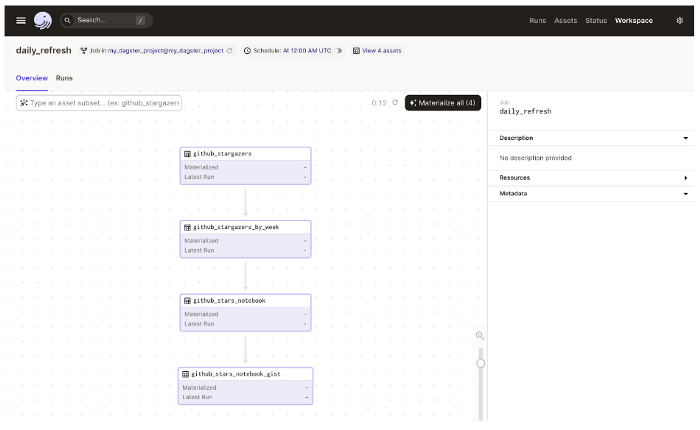
$ dagit

Open the UI by going to<http://localhost:3000/>.

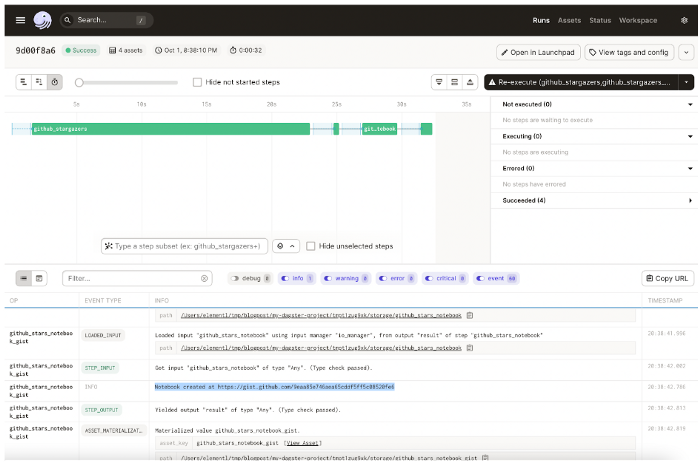
Next, click “status” in the upper right nav, and select the “schedules” tab. We should see our daily schedule. You’ll see a warning that your[daemon](https://docs.dagster.io/deployment/dagster-daemon) isn’t running; that’s fine to ignore for this tutorial.



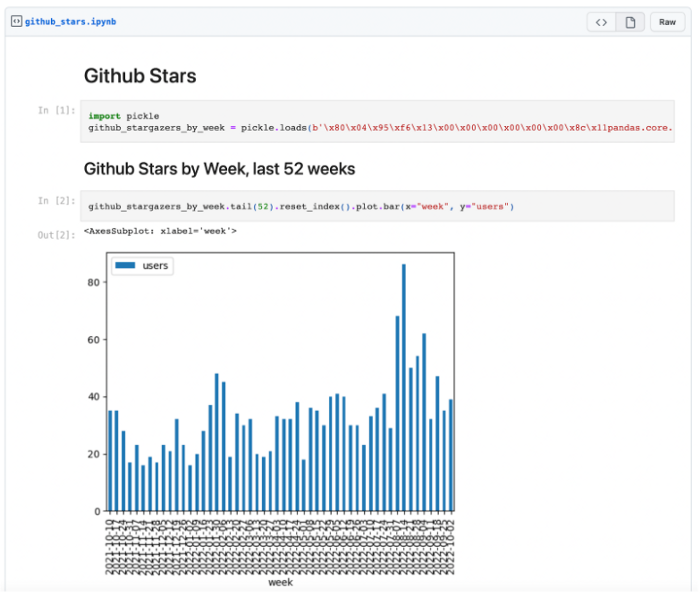
Click the job corresponding to the schedule: daily\_refresh. Then hit “materialize all” to run the job.



The process will run for a bit, and when it completes, you should see a GitHub gist URL printed to the log in the Dagit UI. Note that the first step of this pipeline can take a while; as you iterate, you only have to materialize that asset once and subsequent runs can reuse it.



And when you navigate to the gist, it should look something like this:



**👨‍🏫 Learning more**

Hopefully, this is enough to get you up and running with building a real-ish data pipeline with Dagster.