✦ Member-only story

How to embed a Dash app into an existing Flask app

Oleg Komarov · Follow

6 min read · Jan 2, 2019

Photo by Jezael Melgoza on Unsplash

Dash (by Plotly) is an open source, simple-to-use Python framework for building beautiful data-driven web applications.

If you're interested in knowing more about the framework, head over to "Introducing Dash", the official release announcement, or check out the many examples from their gallery.

What's relevant though is that Dash uses Flask for the back end.

If you're familiar with Flask, you will know that it has a minimalistic core, it's highly customizable and comes with easy-to-add extensions.

> And so, it's only natural to think that Dash apps should be easy to integrate into an existing Flask app.

Table of Contents

Existing solutions

Resources on Dash integration with an existing Flask app are **sparse and incomplete.**

Searching for *"dash integration flask"* brings up code from the official docs on deployment. ~~But it's hardly even an example.~~ The Dash team did a great job and addressed requests by the community. Notheless, the new examples are only using a bare-bones Flask setup.

Other search results point to StackOverflow questions like Running a Dash app within a Flask app or to the Dash's github issue that requests A simple working example for embedding dash in flask under a path.

These resources are still insufficient and usually have several of the following problems:

- examples are not self-contained, difficult to test, disorganized
- require an unrealistically simplified Flask app
- do not address specific basic requirements like authentication

And so, if you also have spent hours searching, reading and testing through dozens of nested links, without much success, you probably came to the same conclusion.

We need a documented strategy for embedding a Dash app into an existing Flask app!

A self-contained example

I created an easily testable and self-contained example which integrates a Dash app into a realistic Flask project.

To test it

Clone the github repository:

```
git clone https://github.com/okomarov/dash_on_flask
cd dash_on_flask
touch .env
```

Add config details to the `.env` file (not committed):

```
export FLASK_APP=dashapp
export FLASK_ENV=development

export DATABASE_URL=sqlite:///${PWD}/app.db

export SECRET_KEY=secret_key_change_as_you_wish_make_it_long_123
```

Now, either with Docker:

```
docker-compose build
docker-compose run
```

Or... load config details, create a virtual environment (optional, will get back to this), install python dependencies, initialize the database, and run the app:

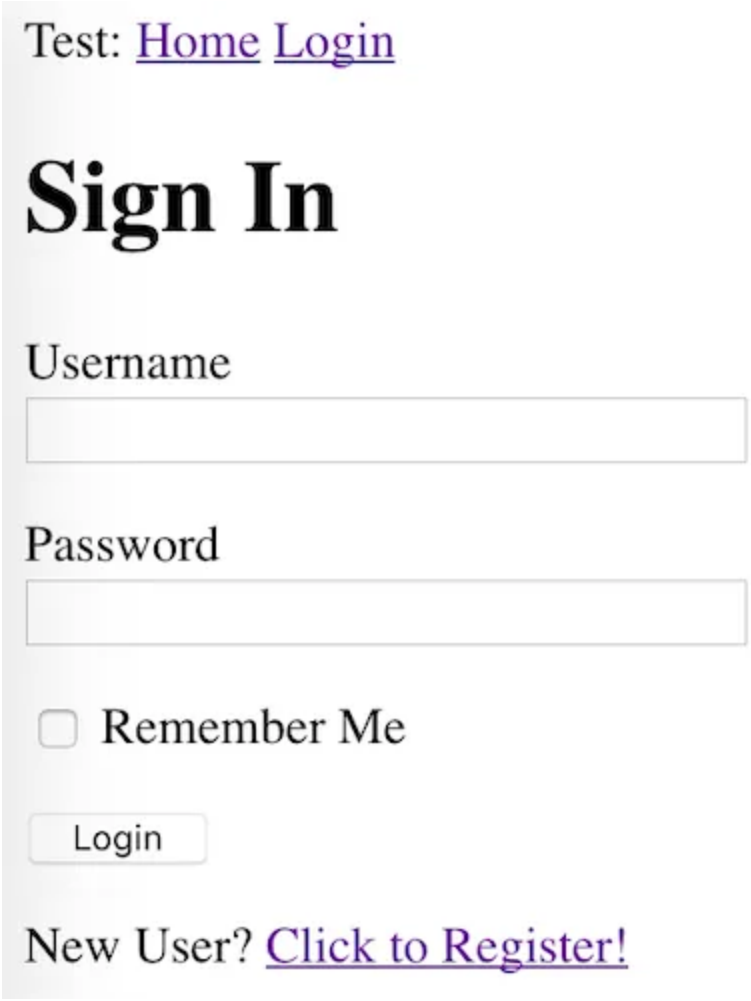
```
source .env
pip install -r requirements.txt

flask db init
flask db migrate -m 'init'
flask db upgrade

flask run
```

Finally, check the app at http://127.0.0.1:5000/dashboard!

You should be re-directed to a sign-in form, with a link to register first. That is, the **dashboard requires authentication**:



Sample from sign in form

This is exactly what we want! Register first, then sign in, and go back to to the */dashboard*.

Now we have an independently built Dash app, served by our Flask app, playing well with our authentication and other extensions.

Implementation details

The app adds the simple stock plotter displayed below (modified to use yahoo prices):

Adapted from Dash's Stock Ticker example

to a Flask app which uses:

- the application factory pattern and blueprints
- a database to manage users (sqlite with Flask-SQLAlchemy and Flask-Migrate)
- authentication (Flask-Login)

This is a standard setup with most Flask applications. Steps and code on how to implement the infrastructure used in my repo are adapted directly from the excellent Flask Mega Tutorial by Miguel Grinberg.

There are a couple of things to note. So let's have a look at the folder structure:

```
.
├── app/
│   ├── __init__.py
│   ├── extensions.py
│   ├── forms.py
│   ├── models.py
│   ├── templates/
│   │   └── ...
│   └── webapp.py
├── app.db
├── config.py
├── dashapp.py
├── migrations/
│   └── ...
├── .env
└── requirements.txt
```

First, the `app/` folder contains everything related to our Flask app, or the server, as referred by the Dash documentation. We create the *main* Blueprint and the routes in `webapp.py` (edited for clarity):

```python
# Imports here
# ....

server_bp = Blueprint('main', __name__)

@server_bp.route('/')
def index():
    return render_template("index.html", title='Home Page')

@server_bp.route('/login', methods=['GET', 'POST'])
def login():
# ...

@server_bp.route('/logout')
@login_required
def logout():
    logout_user()

    return redirect(url_for('main.index'))

@server_bp.route('/register', methods=['GET', 'POST'])
def register():
# ...
```

webapp.py hosted with ❤️ by **GitHub** **view raw**

Full code here

and define the app factory in `__init__.py` :

```python
from flask import Flask
from config import BaseConfig

def create_app():
    server = Flask(__name__)
    server.config.from_object(BaseConfig)

    register_extensions(server)
    register_blueprints(server)

    return server

def register_extensions(server):
    from app.extensions import db
    from app.extensions import login
    from app.extensions import migrate

    db.init_app(server)
    login.init_app(server)
    login.login_view = 'main.login'
    migrate.init_app(server, db)

def register_blueprints(server):
    from app.webapp import server_bp

server.register_blueprint(server_bp)
```

__init__.py hosted with ❤️ by **GitHub** **view raw**

Second, `dashapp.py` is the file which we run with `flask run`. That file creates the server Flask app which is re-used by the Dash app.

Moreover, the file is also used to define the layout and callbacks of the Dash app and to protect its routes/views (edited for clarity):

```python
# Other dash and not related imports
# ...

# Import factory method
from app import create_app

# Method to protect dash views/routes
def protect_dashviews(dashapp):
    for view_func in dashapp.server.view_functions:
        if view_func.startswith(dashapp.url_base_pathname):
            dashapp.server.view_functions[view_func] = login_required(dashapp.server.vi

# Create Flask server app
server = create_app()

# ============================
# Dash app
# ============================

# Create dash app passing our server
dashapp = dash.Dash(__name__, server=server, url_base_pathname='/dashboard/')
protect_dashviews(dashapp)

# Push an application context so we can use Flask's 'current_app'
with server.app_context():
    #  Layout definition
    dashapp.layout = html.Div([
    # ...
    ])

    # Callback
    @dashapp.callback(Output('my-graph', 'figure'), [Input('my-dropdown', 'value')])
    def update_graph(selected_dropdown_value):
    # ...


# ============================
# Another dash app
# ============================
# ...
```

dashapp.py hosted with ❤️ by **GitHub** **view raw**

Older code, see below for proper Dash app organization

Factor out the Dash app into its own subfolder

As you probably noticed, the file that instantiates the app (sometimes called `run.py`) is more **cluttered** than usual. For instance, the Dash app should really belong to its own folder.

So, if you are not using the application factory pattern, you are probably better off with the simpler structure described in this answer.

Following the workaround explained in this answer, we can put the Dash app code into its own subfolder under the `/app`:

```
.
├── app
│   ├── __init__.py
│   ├── dashapp1
│   │   ├── assets/
│   │   ├── callbacks.py
│   │   └── layout.py
│   ⋮
│   └── webapp.py
├── dashapp.py
⋮
```

where layout and callbacks are in their **separate** files (side by side editing is easier than having callbacks underneath layout):

```python
from datetime import datetime as dt

import pandas_datareader as pdr
from dash.dependencies import Input
from dash.dependencies import Output


def register_callbacks(dashapp):
    @dashapp.callback(Output('my-graph', 'figure'), [Input('my-dropdown', 'value')])
    def update_graph(selected_dropdown_value):
        df = pdr.get_data_yahoo(selected_dropdown_value, start=dt(2017, 1, 1), end=dt.n
        return {
            'data': [{
                'x': df.index,
                'y': df.Close
            }],
            'layout': {'margin': {'l': 40, 'r': 0, 't': 20, 'b': 30}}
        }
```

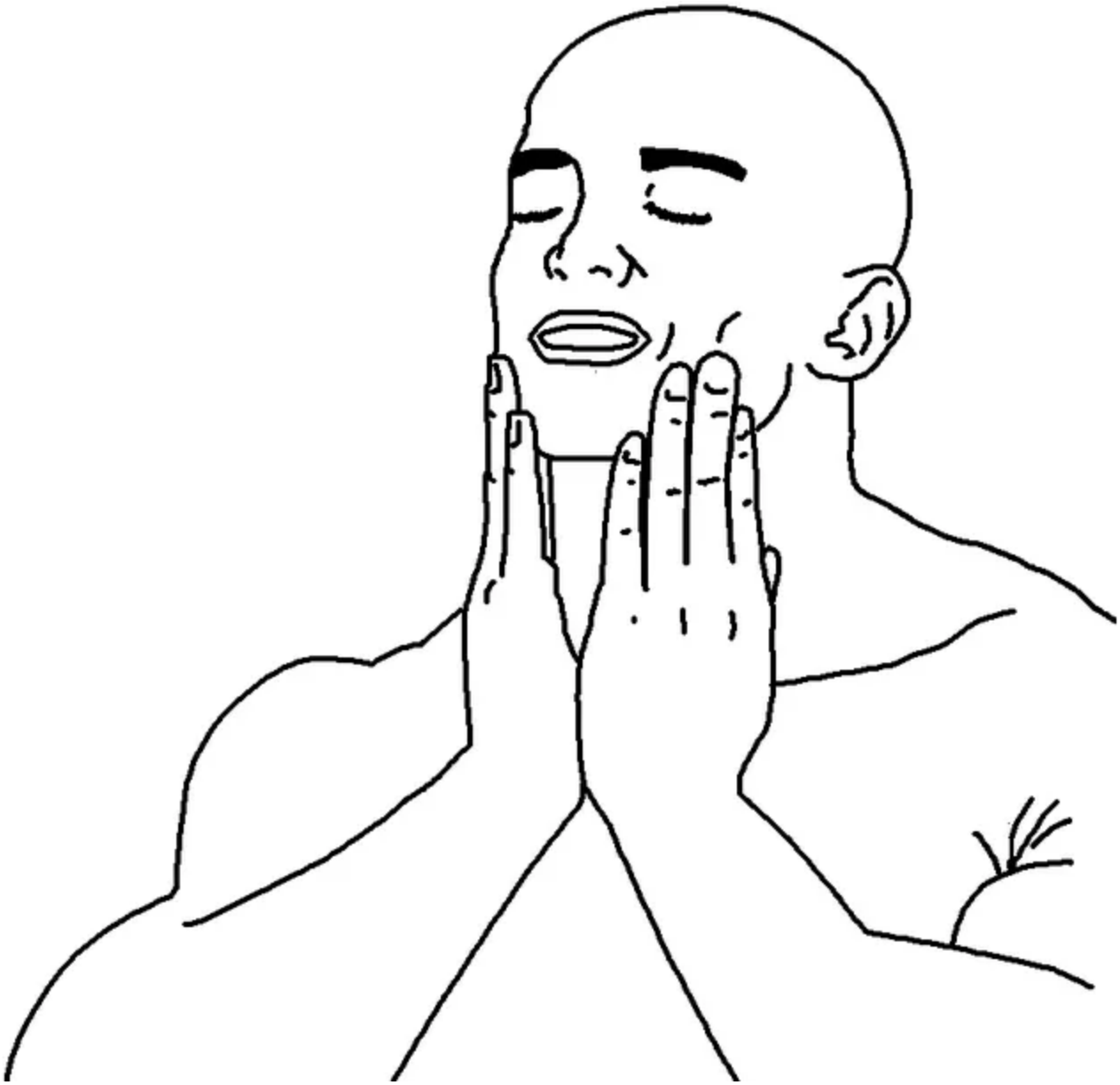
callbacks.py hosted with ❤️ by **GitHub** **view raw**

```python
import dash_core_components as dcc
import dash_html_components as html

layout = html.Div([
    html.H1('Stock Tickers'),
    dcc.Dropdown(
        id='my-dropdown',
        options=[
            {'label': 'Coke', 'value': 'COKE'},
            {'label': 'Tesla', 'value': 'TSLA'},
            {'label': 'Apple', 'value': 'AAPL'}
        ],
        value='COKE'
    ),
    dcc.Graph(id='my-graph')
], style={'width': '500'})
```

layout.py hosted with ❤️ by **GitHub** **view raw**

New layout and callbacks organization for Dash app



The Dash app instatiation is now all done inside `__init__.py` (for details check the github repository) and our main `dashapp.py` is just:

```
from app import create_app
server = create_app()
```

Feels good!

Multiple Dash apps on a single Flask server

Now that we know how to factor out our Dash app into its own subfolder we can instantiate multiple Dash apps under the same Flask server.

The full solution is on github under the branch feature/multiple_dash_apps.

As a side note, if you are not using the factory pattern to start the Flask server, skip this section and follow the official documention in how to structure a multi-page app.

Now, suppose you have another Dash app which you created under `/app/dashapp1` :

```
.
├── app
│   ├── __init__.py
│   ├── dashapp1
│   │   ├── assets/
│   │   ├── callbacks.py
│   │   └── layout.py
│   ├── dashapp2
│   │   ├── assets/
│   │   ├── callbacks.py
│   │   └── layout.py
│   ⋮
│   └── webapp.py
├── dashapp.py
⋮
```

We can adapt our `/app/__init__.py` to:

```python
def create_app():
    server = Flask(__name__)
    server.config.from_object(BaseConfig)

    from app.dashapp1.layout import layout as layout1
    from app.dashapp1.callbacks import register_callbacks as register_callbacks1
    register_dashapp(server, 'Dashapp 1', 'dashboard', layout1, register_callbacks1)

    from app.dashapp2.layout import layout as layout2
    from app.dashapp2.callbacks import register_callbacks as register_callbacks2
    register_dashapp(server, 'Dashapp 2', 'example', layout2, register_callbacks2)

    register_extensions(server)
    register_blueprints(server)

    return server


def register_dashapp(app, title, base_pathname, layout, register_callbacks_fun):
    # Meta tags for viewport responsiveness
    meta_viewport = {"name": "viewport", "content": "width=device-width, initial-scale=

    my_dashapp = dash.Dash(__name__,
                           server=app,
                           url_base_pathname=f'/{base_pathname}/',
                           assets_folder=get_root_path(__name__) + f'/{base_pathname}/a
                           meta_tags=[meta_viewport])
    # Push an application context so we can use Flask's 'current_app'
    with app.app_context():
        my_dashapp.title = title
        my_dashapp.layout = layout
        register_callbacks_fun(my_dashapp)
    _protect_dashviews(my_dashapp)
...
```

__init__.py hosted with ❤️ by **GitHub** **view raw**

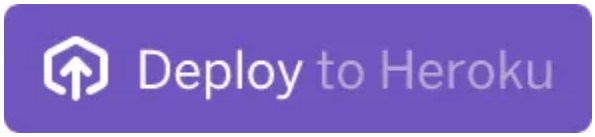
Instantiating multiple Dash app under the same flask server

We can now launch our Flask server and navigate to <*base_url*>/*dashboard* for the first Dash app or to <*base_url*>/*example* for the second app!

Deploy to Heroku for Free

1. Fork the Github repository
2. In the forked repository, edit the `app.json` and replace the value of the `"repository": "https://github.com/okomarov/dash_on_flask"` with the URL of the forked repository.
3. Click on the "Deploy to Heroku" button.

Python Dash Flask Web Development Business Intelligence



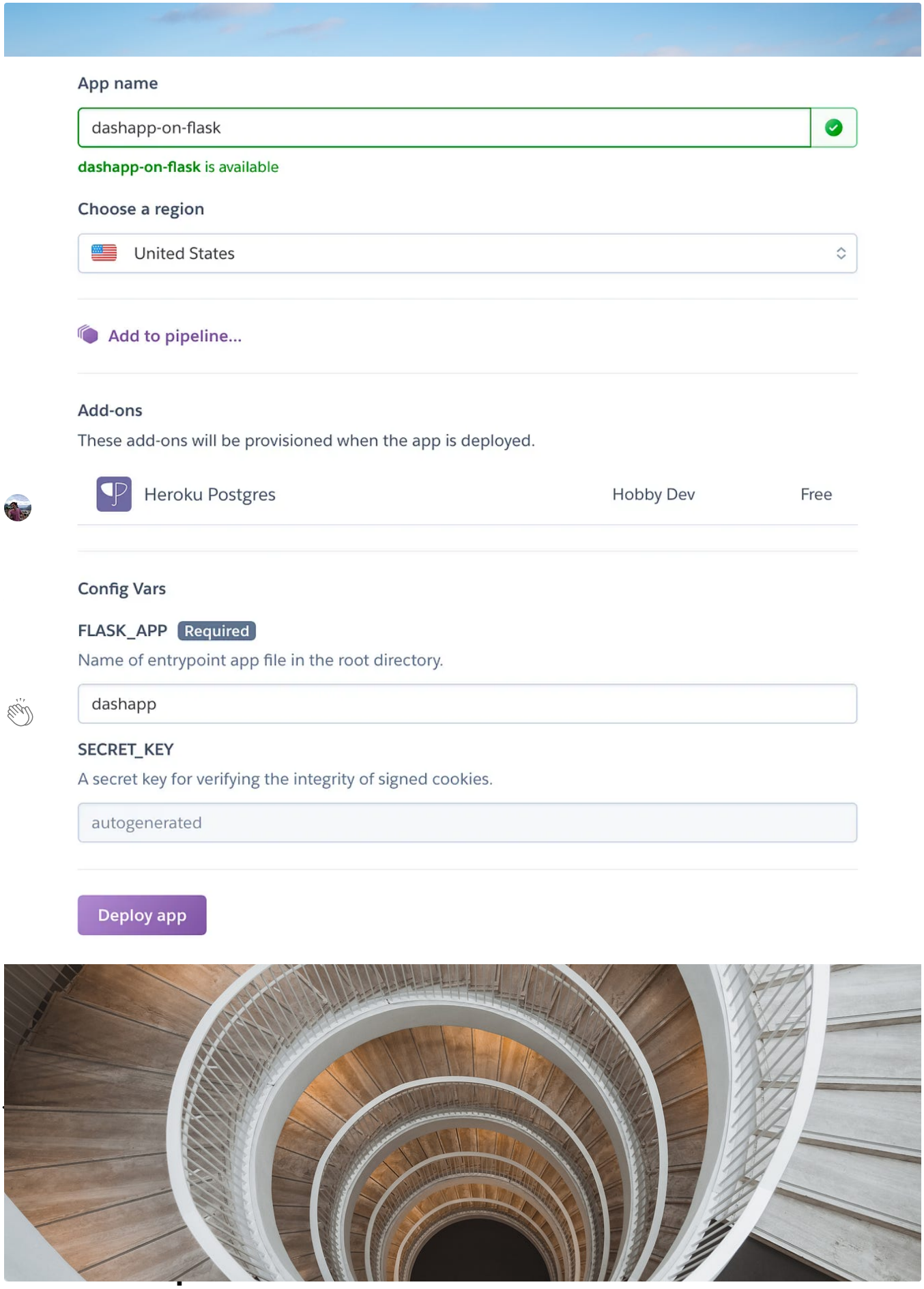
In the new screen, if you don't have a Heroku account create one. Then in the app creation screen, pick a name for the app and click on Deploy.

Follow

Written by Oleg Komarov

149 Followers

More from Oleg Komarov


App name
dashapp-on-flask
dashapp-on-flask is available
Choose a region
United States
Add to pipeline...
Add-ons
These add-ons will be provisioned when the app is deployed.
Heroku Postgres
Hobby Dev
Free
Config Vars
FLASK_APP
Required
Name of entrypoint app file in the root directory.
dashapp
SECRET_KEY
A secret key for verifying the integrity of signed cookies.
autogenerated
Deploy app

Oleg Komarov

direnv

Build your own viral queue in Flask

To run the example we need to create some environment variables, which are then automatically imported by the `config.py`. We place those variables into a file called `.env` and source them when required.

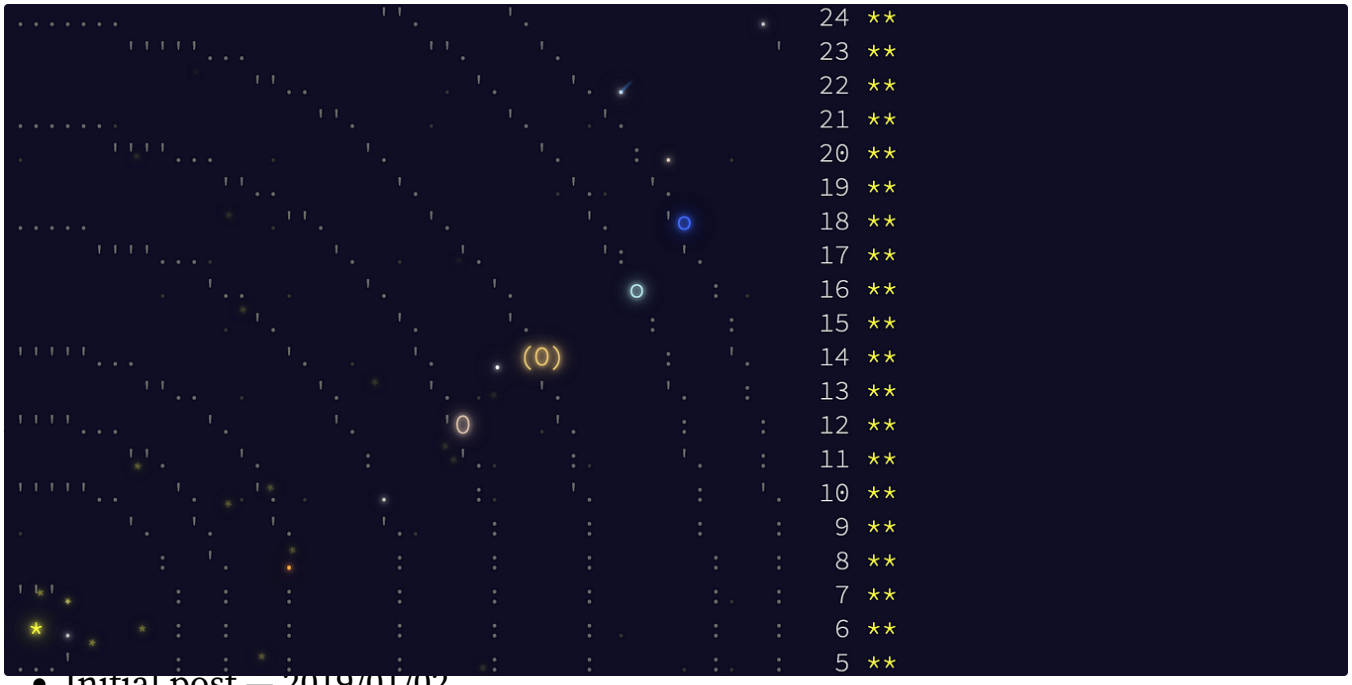
Validate your value proposition and build traction for your startup with a viral queuer.

17 min read · Nov 13, 2019

126 1

I let direnv auto-magically load project-specific environment variables defined in the `.env` file when I `cd` into the folder.

virtualenvwrapper



- Initial post — 2019/01/02

Oleg Komarov

- Better Dash app files organization — 2019/02/21

A Commented Journey Through the Advent of Code 2019 by an Average Joe

- Multiple Dash apps under a single Flask server — 2019/05/05

This is a diary and a post in progress as the challenge continues until December 25th.

- Added Flask context to Dash apps — 2019/07/09

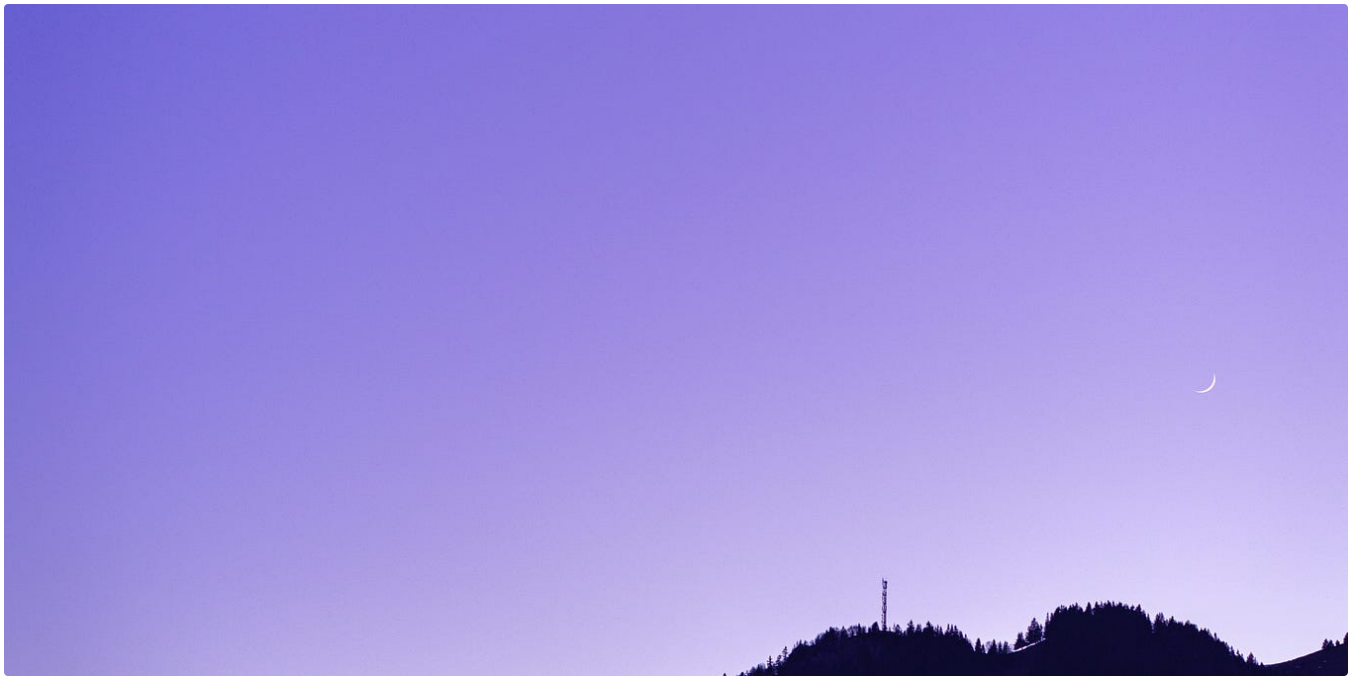
37 min read · Dec 4, 2019

- Deploy to Heroku — 2019/11/15

21 2

- Added Docker support — 2021/05/03

If you liked the story, leave a few claps 👏



Oleg Komarov in Seedsource

You don’t have much money? Invest anyway. Here is why

You would like to invest, but only have $50 per month to spare, and that’s not a lot. You think that to be an investor, you must have a...

5 min read · Dec 24, 2018

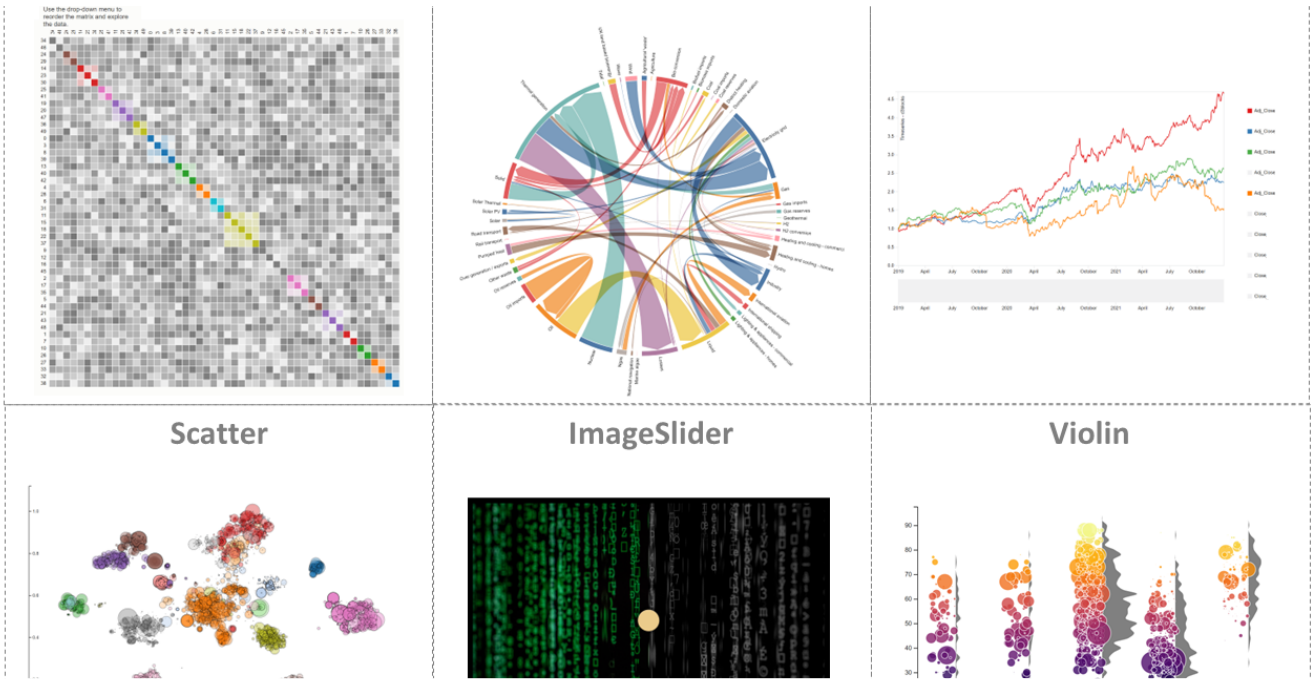
Jacob Ferus in Level Up Coding

How to Build a Dividend Investing Dashboard in Python and Streamlit

Dividend investing is a relatively safe and, in my opinion, fun investing strategy that focuses on companies that have paid out dividends...

· 8 min read · Feb 11

130



Erdogan Taskesen in Towards Data Science

D3Blocks: The Python Library to Create Interactive and Standalone D3js Charts.

Create interactive, and stand-alone charts that are built on the graphics of d3 javascript (d3js) but configurable with Python.

· 11 min read · Sep 22, 2022

801 6

Lists

Stories to Help You Grow as a Software Developer

19 stories · 102 saves

Staff Picks

342 stories · 97 saves

Lynn Kwong in Towards Data Science

Build a WebSocket Application with FastAPI and Angular

Learn to build a two-way interactive communication application with the WebSocket protocol

· 8 min read · Jan 30

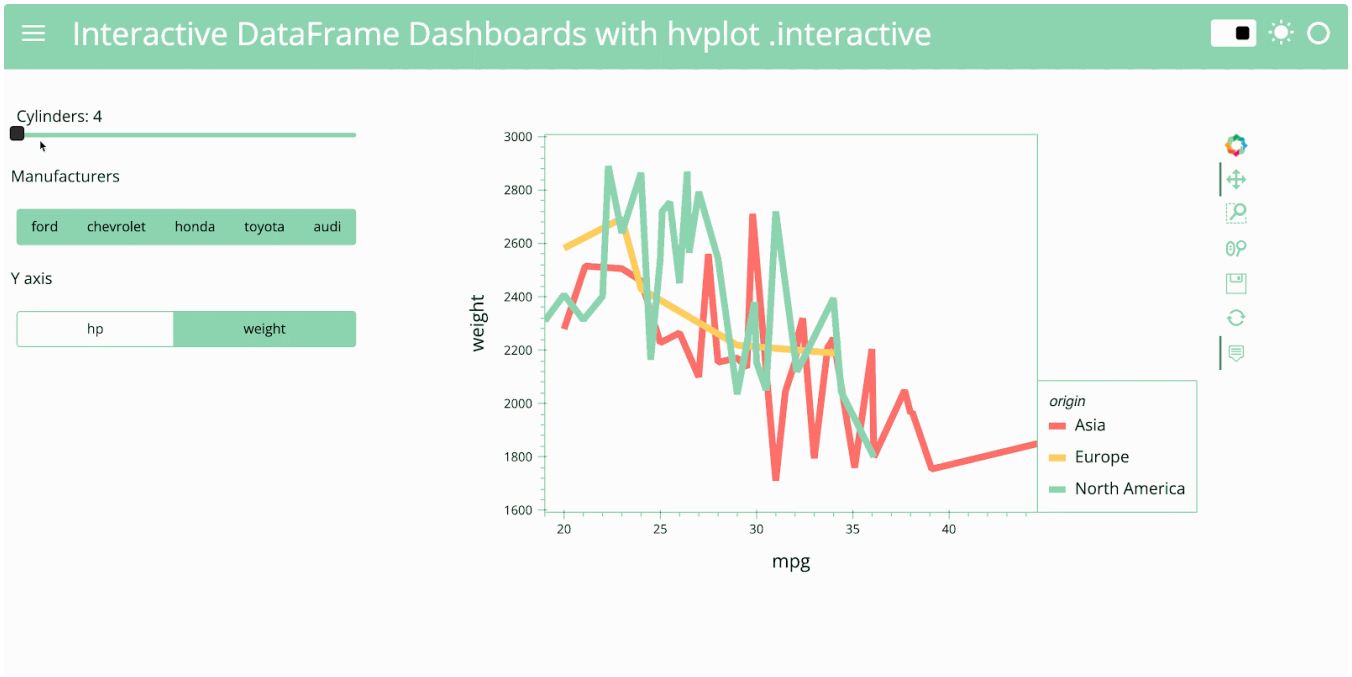
77

Gonçalo Chambel

Creating a Full Stack Python Application with Streamlit and Firebase

Creating a web-based Python Quiz game that gathers data on the answers and stores them in a database, using Firebase and Streamlit

· 18 min read · Jan 17



Sophia Yang, Ph.D. in Towards Data Science

3 ways to build a Panel visualization dashboard

hvPlot .interactive, Panel .bind, and Param .depends

✦ · 7 min read · Sep 20, 2022

231

Yujian Tang in Plain Simple Software

Create an API with User Management using FastAPI and Firebase

A quick start guide to creating an API with user authentication

✦ · 12 min read · Dec 13, 2022

103

See more recommendations