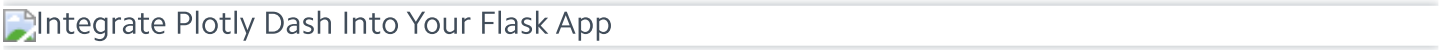
Integrate Plotly Dash Into Your Flask App

Todd Plotly 10 Min Read



Ahh, **Plotly.** Typing that name into a post headline triggers an emotional cocktail of both pride and embarrassment. Plotly has been at the core of some of the most influential products I've personally worked on over the years: a jumble of Fintech and humanitarian clients, all of which are still proudly waving their charts and dashboards around the world. Yet, my mind is boggled by a simple question: what the *hell* took us so long to write our first post about Plotly? We've been operating Hackers and Slackers for over a full year now... did I seriously write a post about JQuery in that time before reaching this point?

Much has changed in the last year or so for our friends in Montreal. Number 1 in my book is the price reduction of their core product: from ***300 dollars*** to ***zero***. I paid the 300 dollars. We really need to get a "donate" button around here.

A close second is undoubtedly the introduction of **Plot.ly Dash**. **Dash** tickles a sentiment which has danced through many young and helplessly naïve Pythonistas' minds: *what if we could write **only** in Python, like, **forever**?* As awful of an idea it is to start Googling Python-to-frontend code interpreters (they exist; I checked), Plotly's Dash does a shockingly good job of breathing life into that romantic fantasy of committing to Python forever.

But we're not here to deliver a recycled '*What is Plotly?*' synopsis. We're not even interested in the obligatory '*How to Get Started Using This Already-Well-Documented-Technology*' post. Plotly deserves better than that. Instead, we're coming hot out of the gate swinging: we're going to show you how to beat Plotly down, break it, and make it bend to your will. Welcome to a magical edition of Hacking Plotly. It must be Christmas, folks.

Let's Make a Plotly + Flask Lovechild from Hell

Like most advancements in Python-related architecture this year, Dash has a little secret: it's gotten here with a little help from Flask. In fact, Dash actually *extends* Flask: every time we make a Dash app, we're actually creating a Flask app with extra bells and whistles. It sounds sensible, and

perhaps even exciting: if you love Flask as I do, your mouth may be watering right now. The prospect of combing the power of Plotly with Flask is the equivalent to every crush you've ever had decided it be best to simply put their differences aside to start a group chat with you in the interest of making your sexual well-being an equal team effort out of sheer love. As you've already guessed, life doesn't work like that.

The moment Dash is initialized with `app = Dash(__name__)`, it spins up a Flask app to piggyback off of. In retrospect, this shouldn't be surprising because the syntax for starting a Dash app is precisely the same as starting a Flask app. Check out the recommended startup boilerplate:

A minimal Plotly Dash app.

```
from dash import Dash
import dash_core_components as dcc
import dash_html_components as html


app = Dash(
    __name__,
    external_stylesheets=['/static/dist/css/style.css'],
    external_scripts=external_scripts,
    routes_pathname_prefix='/dash/'
)

app.layout = html.Div(id='example-div-element')

if __name__ == '__main__':
    app.run_server(debug=True)
```

If you were to attempt to take this boilerplate and try to add core Flask logic, such as authentication with `Flask-Login`, generating assets with `Flask-Assets`, or just creating a global database, where would you start? Plotly cleverly suggests reserving the `app` namespace for your app- the very same that we would do with Flask. Yet if we attempt to modify the `app` object the same as we would with Flask, nothing will work. Plotly has (perhaps intentionally) created a sandbox for you with specific constraints. It's understandable: Plotly is a for-profit company, and this is a no-profit product. If it were too easy to bend Plotly Dash, would companies still need an enterprise license?

Dash excels at what it was intended to do: building dashboard-based applications. The issue is that applications which can *only display data* aren't always useful end products. What if we wanted to create a fully-featured app, where data visualization was simply a *feature* of said app?

Creating a Fully-Featured App (Where Data Vis is Simply a Feature of Said App)

A common workaround you'll find in the community is passing Flask to Dash as the underlying "server", something like this:

A lackluster solution.

```
from flask import Flask
from dash import Dash
import dash_core_components as dcc
import dash_html_components as html


server = Flask(__name__)
app = dash.Dash(
    __name__,
    server=server,
    url_base_pathname='/dash'
)

app.layout = html.Div(id='dash-container')

@server.route("/dash")
def my_dash_app():
    return app.index()
```

Make no mistake: this method *sucks.* Sure, you've regained the ability to create routes here and there, but let's not forget:

- Your app will always start on a Dash-served page: if anything, we'd want our start page to be something we have full control over to then dive into the Dash components.
- Access to globally available Flask plugins is still unavailable in this method. Notice how we never set an application context?
- Your ability to style your application with static assets and styles is entirely out of your hands.
- Container architecture built on Flask, such as Google App Engine, won't play nicely when we start something that isn't Flask. So there's a good chance that playing by the rules means losing the ability to deploy.

If we want to do these things, we cannot start our app as an instance of Dash and attempt to work around it. Instead, we must create a Flask app, and put Dash in its place as an app embedded in *our* app. This gives us full control over when users can enter the Dash interface, and even within

that interface, we can still manage database connections or user sessions as we see fit. Welcome to the big leagues.

Turning the Tables: Dash Inside Flask

So what does "Dash inside Flask" look like from a project structure perspective? If you're familiar with the Flask Application Factory pattern, it won't look different at all:

plotlydash-flask-tutorial

```
/plotlydash-flask-tutorial
├── /plotlyflask_tutorial
│    ├── __init__.py
│    ├── routes.py
│    ├── /static
│    ├── /templates
│    └── /plotlydash
│         └── dashboard.py
├── /data
├── README.md
├── config.py
├── requirements.txt
├── start.sh
└── wsgi.py
```

That's right folks: I'm going to shove proper app structure down your throats any chance I get: even in the midst of a tutorial about hacking things together.

Initializing Flask with Dash

As with any respectable Flask app, our entry point resides within **wsgi.py** as expected. Simply standard Flask thusfar:

wsgi.py

```
"""Application entry point."""
from plotlyflask_tutorial import init_app

app = init_app()

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

The initialization of our app happens within the `init_app()` function following the **Flask Application Factory** pattern. As a refresher, here's what a barebones function to initialize a Flask app looks like:

__init__.py

```
"""Initialize Flask app."""
from flask import Flask


def init_app():
    """Construct core Flask application."""
    app = Flask(__name__, instance_relative_config=False)
    app.config.from_object('config.Config')

    with app.app_context():
        # Import parts of our core Flask app
        from . import routes

        return app
```

This example is as simple as it gets. A Flask app is created, a config file is loaded, and some routes are imported. Pretty boring.

Now we're going to embed an app (Dash) within an app. Keep an eye on what changes:

__init__.py

```
"""Initialize Flask app."""
from flask import Flask


def init_app():
    """Construct core Flask application with embedded Dash app."""
    app = Flask(__name__, instance_relative_config=False)
    app.config.from_object('config.Config')

    with app.app_context():
        # Import parts of our core Flask app
        from . import routes

        # Import Dash application
        from .plotlydash.dashboard import create_dashboard
        app = create_dashboard(app)

        return app
```

All it takes is two lines! The first line imports a function called `init_dashboard()` which initializes a Dash application, not unlike what `init_app()` does itself. The second line registers our isolated Dash app onto our parent Flask app:

__init__.py

```
...

# Import Dash application
from .plotlydash.dashboard import init_dashboard
app = init_dashboard(app)

...
```

Let's turn our focus to `import init_dashboard` for a moment. We're importing a file called **dashboard.py** from a directory in our Flask app called /**plotlydash**. Inside **dashboard.py** is a single function which contains the entirety of a Plotly Dash app in itself:

dashboard.py

```
from dash import Dash


def init_dashboard(server):
    """Create a Plotly Dash dashboard."""
    dash_app = dash.Dash(
        server=server,
        routes_pathname_prefix='/dashapp/',
        external_stylesheets=[
            '/static/dist/css/styles.css',
        ]
    )

    # Create Dash Layout
    dash_app.layout = html.Div(id='dash-container')

    return dash_app.server
```

We pass our Flask instance to `create_dashboard()` as a parameter called `server`. Unlike the previous examples, its actually *`server`* running the show this time, with Dash piggybacking as a module. This brings us to our most important line of code:

dashboard.py

```
...
dash_app = dash.Dash(
        server=server,
        routes_pathname_prefix='/dashapp/',
        external_stylesheets=[
            '/static/dist/css/styles.css',
        ]
    )
...
```

Instead of creating our `dash_app` object as a global variable (as is suggested), we stuck in a function called `create_dashboard()`. This allows us to pass our top-level Flask app into Dash as `server`, hence `dash_app = Dash(server=server)`. This effectively spins up a Dash instance using *our* Flask app at its core, as opposed to its own!

Take note of how we pass a value to `routes_pathname_prefix` when creating `dash_app`. This is effectively our workaround for creating a route for Dash within a larger app: everything we build in this app will be preceded with the prefix we pass (of course, we could always pass `/` as our prefix). Dash has full control over anything we build *beneath* the hierarchy of our prefix, and our parent Flask app can control pretty much anything else. This means we can build a sprawling Flask app with hundreds of features and views, and if we want a Dash view, we can just create a module or subdirectory for that to chill in. It's the best collab since jeans and pockets.

Now you're thinking with portals™.

Subtle Differences

Because we create Dash in a function, we should be aware of how this will change the way we interact with the core dash object. The bad news is copy + pasting other people's code will almost certainly not work, because almost every keeps the `Dash()` object as a global variable named `app`. The good news is, it doesn't matter! We just need to structure things a bit more logically.

For example, consider callbacks. Dash enables us to create callback functions with a nifty `callback` decorator. The docs structure this as such:

Standard Plotly Dash file structure

```
import dash
from dash.dependencies import Input, Output
import dash_table
import dash_html_components as html

app = dash.Dash(  name  )
```

```
app.layout = html.Div([
    # Layout stuff
    ...
])


@app.callback(
    # Callback input/output
    ....
    )
def update_graph(rows):
    # ... Callback logic
```

Notice how everything is global; `app` is global, we set `app.layout` globally, and callbacks are defined globally. This won't work for us for a number of reasons. Namely, we don't create `Dash()` upon file load; we create it when our parent Flask app is ready. We need to structure our Dash file a bit more logically by using functions to ensure our app is loaded before defining things like callbacks:

Plotly Dash within a Python function

```
import dash
from dash.dependencies import Input, Output
import dash_table
import dash_html_components as html

def init_dashboard(server):
    app = dash.Dash(__name__)
    app.layout = html.Div([
        # ... Layout stuff
    ])

    # Initialize callbacks after our app is loaded
    # Pass dash_app as a parameter
    init_callbacks(dash_app)

    return dash_app.server

def init_callbacks(dash_app):
    @app.callback(
    # Callback input/output
    ....
    )
    def update_graph(rows):
        # Callback logic
        ...
```

See, not so bad!

Creating a Flask Homepage

Because the entry point to our app now comes through Flask, **routes.py** has the flexibility to serve up anything we want. We now have the freedom to build an app without restriction, jumping in or out of Plotly Dash on the views we see fit. I added a simple landing page to demonstrate this:

routes.py

```
"""Routes for parent Flask app."""
from flask import render_template
from flask import current_app as app


@app.route('/')
def home():
    """Landing page."""
    return render_template(
        'index.jinja2',
        title='Plotly Dash Flask Tutorial',
        description='Embed Plotly Dash into your Flask applications.',
        template='home-template',
        body="This is a homepage served with Flask."
    )
```

dashboard.py

`dashboard.py` is the Dash app we have living *within* our Flask app. But how does Flask know which route is associated to Dash? Wasn't it missing from **routes.py**? Indeed it was, good fellow! Because we set **routes_pathname_prefix** while creating the `dash_app` object, we *don't need* to create a route for Dash: it will always be served whenever we navigate to `127.0.01/dashapp`. Thus, we can link to our dashboard via a regular Flask template like so:

```
<a href="/dashapp/" class="dash-link">
    <span>Embdedded Plotly Dash</span>
    <i class="fas fa-arrow-right"></i>
</a>
```

Creating Something Useful

I threw together a working demo of Dash within Flask to demonstrate this in action. The example below shows the journey of a user navigating our app. The user lands on the homepage of our

Flask app which we defined in **routes.py**. From there, the user is able to click through to the Plotly Dash dashboard we define in **dashboard.py** seamlessly:

Demonstration of a Plotly Dash within Flask.

The user is none the wiser that they've jumped from a Flask application to a Dash application, which is what makes this practice so appealing: by combining the ease of Plotly Dash with a legitimate web app, we're empowered to create user experiences which go far beyond the sum of their parts.

If you're hungry for some source code to get started building your own Plotly Dash views, here's the source I used to create the page above:

dashboard.py

```
"""Instantiate a Dash app."""
import numpy as np
import pandas as pd
import dash
import dash_table
import dash_html_components as html
import dash_core_components as dcc
from .data import create_dataframe
from .layout import html_layout
```

```
from .layout import html_layout



def init_dashboard(server):
    """Create a Plotly Dash dashboard."""
    dash_app = dash.Dash(
        server=server,
        routes_pathname_prefix='/dashapp/',
        external_stylesheets=[
            '/static/dist/css/styles.css',
            'https://fonts.googleapis.com/css?family=Lato'
        ]
    )

    # Load DataFrame
    df = create_dataframe()

    # Custom HTML layout
    dash_app.index_string = html_layout

    # Create Layout
    dash_app.layout = html.Div(
        children=[dcc.Graph(
            id='histogram-graph',
            figure={
                'data': [{
                    'x': df['complaint_type'],
                    'text': df['complaint_type'],
                    'customdata': df['key'],
                    'name': '311 Calls by region.',
                    'type': 'histogram'
                }],
                'layout': {
                    'title': 'NYC 311 Calls category.',
                    'height': 500,
                    'padding': 150
                }
            }),
            create_data_table(df)
        ],
        id='dash-container'
    )
    return dash_app.server



def create_data_table(df):
    """Create Dash datatable from Pandas DataFrame."""
    table = dash_table.DataTable(
        id='database-table',
        columns=[{"name": i, "id": i} for i in df.columns],
        data=df.to_dict('records'),
        sort_action="native",
        sort_mode='native',
        page_size=300
    )
```

```
    return table
```

A working version of this demo is live here:

Plotly Dash Flask Tutorial - Hackers and Slackers

Embed Plotly Dash into your Flask applications.

Plotly Dash Flask Tutorial

I've uploaded the source code for this working example up on Github. Take it, use it, abuse it. It's all yours:

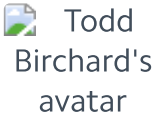
toddbirchard/plotlydash-flask-tutorial

Source code for the accompanying tutorial. Contribute to toddbirchard/plotlydash-flask-tutorial development by creating an…

toddbirchard • GitHub

Needless to say, there's way more cool shit we can accomplish with Plotly Dash. Stick around long enough and there's a good chance we'll cover all of them.

Plotly | Flask | Data Vis | Python | Software Development



Todd Birchard

Website Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.

lotly-chart-dio	**Create Cloud-hosted Charts with Plotly Chart Studio**	Plotly, Python
crape-metadata--ld	**Scrape Structured Data with Python and Extruct**	Python, Scraping
thon-poetry-package-ager	**Package Python Projects the Proper Way with Poetry**	Python, Software Development

Monthly Newsletter

Toss us your email and we'll promise to only give you the good stuff.

Your name

Your email address

Sign Up

Support us

We started sharing these tutorials to help and inspire new scientists and engineers around the world. If Hackers and Slackers has been helpful to you, feel free to buy us a coffee to keep us going :).

Buy me a coffee

Hackers and Slackers Logo

Community of hackers obsessed with data science, data engineering, and analysis. Openly pushing a pro-robot agenda.

Pages

About

Series

Join

Search

Donate

Series

Python Code Snippet Corner

Build Flask Apps

Data Analysis with Pandas

Rise of Google Cloud

Learning Apache Spark

Creating APIs in AWS

Working with MySQL

Authors

Todd Birchard

Max Mileaf

Graham Beckley

David Aquino

Matthew Alhonte

Ryan Rosado

Paul Armstrong

Dylan Castillo