Callbacks, Layouts, & Bootstrap: How to Create Dashboards in Plotly Dash

Mitchell Krieger Feb 27 · 7 min read ★

Photo by Luke Chesser on Unsplash

display what you are discovering in your data exploration and machine learning modeling. Plotly offers a robust open-source version of their library for building dashboards called Dash. In addition, there are multiple additional open-source libraries that make building a dashboard in Dash easy. This tutorial is an overview of one way to implement your first multi-page Dashboard app using Dash.

What is Dash?

Dash is a framework that allows you to create highly customizable and interactive data visualization apps that are rendered in your web browser using only Python. Dash is also available in R and Julia but this tutorial will focus on Python. Built on top of Plotly, Flask, and React, Dashboards in Dash have the capability to be deployed to the internet and displayed cross-platform and on mobile devices. Essentially Dash is able to create web applications without the use direct of HTML or CSS. We'll focus on the free open source version here but Plotly offers an enterprise version for Dash for use in managing enterprise dashboards.

Key Elements of Dash

Before we dive into specifics, let's go over some key ideas that we'll use in building our first multi-page dash app:

- **Layouts & Components:**
  Layouts describe what your dashboard will look like. This includes style choices like fonts and colors generated by CSS scripts and what components will populate the app. The `dash_core_components` and `dash_html_components` libraries provide a variety of frequently used components of dash apps. HTML components are python versions of HTML tags that can be used in your dash app, allowing you to create Divs, display text, structure paragraphs/headers, and so on. You can also assign them a style, a class, or an id just like in HTML. The core components are various useful elements to place on your dashboard just as dropdown menus, graphs, sliders, buttons, and so on. There are many additional Dash component libraries that you can find in Dash's documentation.

- **Callbacks:**
  Callbacks are python decorators that control the interactivity of your dash app. A

property to look for changes made by the user to your dashboard app's components and executes any associated changes you define. For example, you could set up a callback that takes in information from a slider component on your dashboard and then executes changes to a graph based on the input from the slider.

- **Bootstrap:**
  If you are already familiar with CSS, you can style your Dash app by writing your own CSS script. But, if you are not familiar with CSS or don't want to take the time to write your own, Bootstrap is the way to go. Bootstrap is a popular CSS Framework for creating interactive and mobile-ready applications. The `dash_bootstrap_components` library by faculty.ai provides CSS styling as well as additional components like app navigation tools, collapsable content, dialog boxes, and more. Bootstrap makes it easy to organize your content by giving you the ability to create columns and rows that contain components. Here's a good rundown of how it operates and check out the documentation for how to translate it into python.

Getting started with your dashboard

First, you need to install Dash and Dash Bootstrap Components (core and HTML components should come pre-installed with Dash)

```
$ pip install dash==1.19.0
$ pip install dash-bootstrap-components
```

Once you have dash installed you'll want to create a directory to keep all of your python scripts that build your app. Note that there are multiple ways to structure this directory and the python files in it, the following is only one way to structure it. I like to structure it this way because I find it helps me keep my code organized. In this structure you'll have **four** files:

PyFile #1: The App

`app.py` : This script is short and simple. It creates your dash app and connects it to a server. If you choose to use Bootstrap CSS styling (or another pre-made CSS stylesheet),

```
import dash
import dash_bootstrap_components as dbc

#Instantiates the Dash app and identify the server
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])
server = app.server
```

app.py hosted with ♡ by GitHub view raw

PyFile #2: Index

`index.py` : This will be your main point of entry into the dashboard. Meaning when you want to run the dashboard app, in your command line you'll type `python index.py` . Here's you'll also define which layouts to display on what pages of your app. The `app.layout` will be your overarching structure. In this example, I have a location component that ties the structure to a URL, instantiates a horizontal navigation bar, and the page content in an HTML div. This structure will display the same on all pages of the app. The callback below that is a special callback that sends the page layout to the `page-content` HTML div in the structure when you click on the links in the navigation bar to a new page. Be sure to import the layouts, components, and styles you want to display from `layouts.py` .

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

from app import app, server
#import your navigation, styles and layouts from layouts.py here
from layouts import nav_bar, layout1, layout2, CONTENT_STYLE
import callbacks

# Define basic structure of app:
# A horizontal navigation bar on the left side with page content on the right.
app.layout = html.Div([
    dcc.Location(id='url', refresh=False), #this locates this structure to the url
    nav_bar(),
    html.Div(id='page-content',style=CONTENT_STYLE) #we'll use a callback to change the layout o
])

```

```
@app.callback(Output('page-content', 'children'), #this changes the content
              [Input('url', 'pathname')]) #this listens for the url in use
def display_page(pathname):
    if pathname == '/':
        return layout1
    elif pathname == '/page1':
        return layout1
    elif pathname == '/page2':
         return layout2
    else:
        return '404' #If page not found return 404

#Runs the server at http://127.0.0.1:5000/
if __name__ == '__main__':
    app.run_server(port=5000, host= '127.0.01',debug=True)
```

index.py hosted with ♡ by GitHub view raw

PyFile #3: Layouts

`layouts.py` : Here's where you'll design your dashboard app. You'll want to set each layout, component, or style equal to a variable or store it in a function so you can access it in `index.py` and in other places in this layouts script. Remember, `layouts.py` is just defining the layouts of each page and various additional components such as a navigation bar. **We'll discuss how to implement interactivity in `callbacks.py` after.**

In this example, I've imported the classic iris dataset to generate two Plotly Express graphs to display. I've also created a navigation bar, two styles, and two layouts. Each layout consists of a 1 row by 2 column bootstrap grid. In the first layout, the column on the left has two tabs that will display different graphs. The column on the right displays text associated with a click on the graph from the user. For the second layout, it demonstrates two dash core components that are great for interactivity: the dropdown menu and radio buttons. They don't do much right now except display the selected value, but that's where you can get creative and figure out how to use the selected value to interact with other components and graphs.

`layouts.py` is likely to be your longest py script.

```
from app import app
import plotly.express as px

#####################################
# Add your data
#####################################

#example iris dataset
df = px.data.iris()

#####################################
# Styles & Colors
#####################################

NAVBAR_STYLE = {
    "position": "fixed",
    "top": 0,
    "left": 0,
    "bottom": 0,
    "width": "16rem",
    "padding": "2rem 1rem",
    "background-color": "#f8f9fa",
}

CONTENT_STYLE = {
    "top":0,
    "margin-top":'2rem',
    "margin-left": "18rem",
    "margin-right": "2rem",
}

#####################################
# Create Auxiliary Components Here
#####################################

def nav_bar():
    """
    Creates Navigation bar
    """
    navbar = html.Div(
    [
        html.H4("System Performance Dashboard", className="display-10", style={'textAlign':'cent
```

```
            [
                dbc.NavLink("Page 1", href="/page1",active="exact", external_link=True),
                dbc.NavLink("Page 2", href="/page2", active="exact", external_link=True)
            ],
            pills=True,
            vertical=True
        ),
    ],
    style=NAVBAR_STYLE,
    )
    return navbar

#graph 1
example_graph1 = px.scatter(df, x="sepal_length",y="sepal_width",color="species")

#graph 2
example_graph2 = px.histogram(df, x="sepal_length", color = "species",nbins=20)
example_graph2.update_layout(barmode='overlay')
example_graph2.update_traces(opacity=0.55)


#####################################
# Create Page Layouts Here
#####################################

### Layout 1
layout1 = html.Div([
    html.H2("Page 1"),
    html.Hr(),
    # create bootstrap grid 1Row x 2 cols
    dbc.Container([
        dbc.Row(
            [
                dbc.Col(
                    [
                        html.Div(
                            [
                            html.H4('Example Graph Page'),
                            #create tabs
                            dbc.Tabs(
                                [
                                    #graphs will go here eventually using callbacks
```

```
                                id="tabs",
                                active_tab='graph1',
                                ),
                            html.Div(id="tab-content",className="p-4")
                            ]
                        ),
                    ],
                    width=6 #half page
                ),

                dbc.Col(
                    [
                        html.H4('Additional Components here'),
                        html.P('Click on graph to display text', id='graph-text')
                    ],
                    width=6 #half page
                )

            ],
        ),
    ]),
])


### Layout 2
layout2 = html.Div(
    [
        html.H2('Page 2'),
        html.Hr(),
        dbc.Container(
            [
                dbc.Row(
                    [
                        dbc.Col(
                            [
                                html.H4('Country'),
                                html.Hr(),
                                dcc.Dropdown(
                                    id='page2-dropdown',
                                    options=[
                                        {'label': '{}'.format(i), 'value': i} for i in [
                                        'United States', 'Canada', 'Mexico'
```

```
                                ),
                                html.Div(id='selected-dropdown')
                            ],
                            width=6
                        ),
                        dbc.Col(
                            [
                                html.H4('Fruit'),
                                html.Hr(),
                                dcc.RadioItems(
                                    id='page2-buttons',
                                    options = [
                                        {'label':'{}'.format(i), 'value': i} for i in [
                                        'Yes ', 'No ', 'Maybe '
                                        ]
                                    ]
                                ),
                                html.Div(id='selected-button')
                            ],
                        )
                    ]
                ),
            ]
        )
    ])
```

layouts.py hosted with ♡ by GitHub view raw

PyFile #4: Callbacks

`callbacks.py` : Any interactivity you want to create should go here. You'll notice that many of these elements in the above `layouts.py` script have an `id=` argument passed through them. These id's are how the callbacks in `callbacks.py` will identify which components to listen to for user input and where to display the output of the callback. The structure of a callback is the following:

```
)
def callback_name(valuesIn):
    #code for callback to execute
    return output_to_children
```

Each callback listens to a component and takes in values from it. This is the `Input()` piece. Then the callback function is executed using these values. Once the function returns a value or object, it is sent to `Output()`. `"children"` is a common output for every callback. Children are sub-elements of a component. We've actually already used children a lot. Each time we nested components inside another in the scripts above we've added a child component. So essentially, when `children` is in the output of the callback, it tells Dash to place the returned content inside the component with the id that matches the id listed in the output. For example, the first callback in `callbacks.py` takes in the value `active_tabs` from the component with the id `tabs` and returns the output of the callback to the component with the id `tab-content`. That returned content is now a child of the `tab-content` component.

I've set up four callbacks in the following script:

1. Listens to which tab is currently clicked & returns the associated graph
2. Listens to clicks on the graph and returns text associated with that click
3. Listens to the dropdown menu and returns the selected value as in a string
4. Listens to the radio buttons and returns the selected value as in a string

Most of these callbacks aren't that interesting, but it helps to understand simple callbacks before starting to build more complex ones.

Once you've got all four files set up, edit them to your data, visualizations from Plotly, and more. Also, try altering the structure/layouts and creating your own callbacks. The best way to learn Dash is to play around and try to experiment with different layouts and

```
$  python index.py

Dash is running on http://127.0.01:5000/

* Serving Flask app "app" (lazy loading)

* Environment: production

WARNING: This is a development server. Do not use it in a production
deployment.

Use a production WSGI server instead.

* Debug mode: on
```

Type the URL where Dash is running into any web browser to see what you've created! If debug mode is on, you should be able to update your scripts and your app will change live. You can turn debug mode off in `index.py` by setting `debug=False`.

Dash is a vast, flexible, and powerful library and there's so much to learn it can be a bit overwhelming at first. But it's a great tool to have in your toolbelt and worth it once you dive in.

Find a GitHub Repository of this project here.