pyxley Documentation

Release 0.0.7

Nicholas Kridler

Jul 07, 2017

Contents

The Pyxley python package makes it easier to deploy Flask-powered dashboards using a collection of common JavaScript charting libraries. UI components are powered by PyxleyJS.

CHAPTER 1

Why Pyxley?

Pyxley was born out of the desire to combine the ease of data manipulation in pandas with the beautiful visualizations available in JavaScript. It was largely inspired by Real Python. The goal was to create a library of reusable visualization components that allow the quick development of web-based data products. For data scientists with limited JavaScript exposure, this package is meant to provide generic visualizations along with the ability to customize as needed.

React

Pyxley utilizes Facebook's React. React is only concerned with the UI, so we can use it for only the front-end portion of our web-applications.

Flask

Flask is a great micro web-framework. It allows us to very easily stand up web applications and services.

PyxleyJS

Pyxley relies on a JavaScript library that heavily leverages React and existing visualization libraries. Wrappers for common libraries have been created so that the user only needs to specify the type of chart they want, the filters they wish to include, and the parameters specific to that visualization.

To download or contribute, visit PyxleyJS.

Insane Templates

A lot of other projects rely on a set of complicated templates that attempt to cover as many use cases as possible. The wonderful thing about React is the ability to create compositions of components. This means that we only need a single template: a parent component that manages all of the child components. With this layout, we can use factories within JavaScript to create the components using the supplied parameters. Organizing the code in this way allows us to create a common framework through which we can create a variety of different components.

For example, the underlying interface for a Dropdown Button and a Line Chart are the same. The only difference is the options supplied by the user. This provides a really easy way to integrate with Python. We can use Python for organizing the types and options. PyReact can then be used to transform to JavaScript.

CHAPTER 2

Core Components

In Pyxley, the core component is the `UILayout`. This component is composed of a list of `charts` and `filters`, a single React component from a JavaScript file, and the Flask app.

```
# Make a UI
from pyxley import UILayout
ui = UILayout(
    "FilterChart",
    "./static/bower_components/pyxley/build/pyxley.js",
    "component_id")
```

This will create a UI object that's based on the `FilterChart` React component in `pyxley.js`. It will be bound to an html `div` element called `component_id`.

If we wanted to add a filter and a chart we could do so with the following

```
# Make a Button
cols = [c for c in df.columns if c != "Date"]
btn = SelectButton("Data", cols, "Data", "Steps")

# Make a FilterFrame and add the button to the UI
ui.add_filter(btn)

# Make a Figure, add some settings, make a line plot
fig = Figure("/mgchart/", "mychart")
fig.graphics.transition_on_update(True)
fig.graphics.animate_on_load()
fig.layout.set_size(width=450, height=200)
fig.layout.set_margin(left=40, right=40)
lc = LineChart(sf, fig, "Date", ["value"], init_params={"Data": "Steps"},␣
↪timeseries=True)
ui.add_chart(lc)
```

Calling the `ui.add_chart` and `ui.add_filter` methods simply adds the components we've created to the layout.

```
app = Flask(__name__)
sb = ui.render_layout(app, "./static/layout.js")
```

Calling `ui.render_layout` builds the JavaScript file containing everything we've created.

Charts

Charts are meant to span any visualization of data we wish to construct. This includes line plots, histograms, tables, etc. Several wrappers have been introduced and more will be added over time.

Implementation

All `charts` are `UIComponents` that have the following attributes and methods

- An endpoint route method. The user may specify one to override the default.
- A `url` attribute that the route function is assigned to by the flask app.
- A `chart_id` attribute that specifies the element id.
- A `to_json` method that formats the json response.

Filters

Filters are implemented in nearly the same way that `charts` are implemented. The only difference is the lack of the `to_json` method.

CHAPTER 3

A Basic Pyxley App

Here we will go through building a basic web-application using Pyxley and Flask.

I recommend visiting the Real Python blog for a great intro to a basic app.

```
App
|   package.json
|   .bowerrc
|   bower.json
|
---project
    |   app.py
    |   templates
    |
    ---static
        |   css
        |   js
```

Some notes about the above structure

- This assumes that you are running the app from the `project` folder.
- Any JavaScript created by the app should go in the `js` folder.

JavaScript!

Node & NPM

At the highest level, Node is our biggest JavaScript dependency. All of the JavaScript dependencies are managed via NPM. This document won't show you how to get Node or NPM, but for Mac OS X users, you can get it through homebrew.

Note: Prior to version 0.0.9, Bower was used to manage dependencies. In an effort to simplify the massive amount of dependencies, NPM will be used as the primary package manager and Bower is completely optional. In addition, PyReact has been deprecated and will no longer be used to transpile the jsx code.

HTML & CSS

HTML templates used by flask are stored in the `templates` folder. For our purposes, we only need a really basic template that has a single `div` element.

```
<div id="component_id"></div>
```

Everything in our app will be tied to this single component.

CSS

We store any additional CSS we need in the `static\CSS` folder.

Flask

Courtesy of the Flask website, “Hello, World!” in Flask looks like the code below.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

We simply need to build upon this.

Adding Some Pyxley

Let’s start by importing some simple things and building upon the example above. We will import some `pyxley` components, create a UI, and load a data frame.

```
# Import flask and pandas
from flask import Flask, render_template
import pandas as pd

# Import pyxley stuff
from pyxley import UILayout
from pyxley.filters import SelectButton
from pyxley.charts.mg import LineChart, Figure

# Read in the data and stack it, so that we can filter on columns
df = pd.read_csv("fitbit_data.csv")
sf = df.set_index("Date").stack().reset_index()
sf = sf.rename(columns={"level_1": "Data", 0: "value"})
```

```
# Make a UI
ui = UILayout(
    "FilterChart",
    "./static/bower_components/pyxley/build/pyxley.js",
    "component_id")

# Create the flask app
app = Flask(__name__)
```

At this point we now have some data and a layout to build upon. Adding the code below will add a dropdown select button and a line plot.

```
# Make a Button
cols = [c for c in df.columns if c != "Date"]
btn = SelectButton("Data", cols, "Data", "Steps")

# Make a FilterFrame and add the button to the UI
ui.add_filter(btn)

# Make a Figure, add some settings, make a line plot
fig = Figure("/mgchart/", "mychart")
fig.graphics.transition_on_update(True)
fig.graphics.animate_on_load()
fig.layout.set_size(width=450, height=200)
fig.layout.set_margin(left=40, right=40)
lc = LineChart(sf, fig, "Date", ["value"], init_params={"Data": "Steps"},␣
↪timeseries=True)
ui.add_chart(lc)
```

Now that our `ui` object is full of filters and charts, we need to write out the JavaScript and transpile the jsx code. Previously, we used PyReact, but unfortunately that has been deprecated. Instead, we rely on `webpack`. We have written a wrapper for `webpack` that does the bundling for us.

:: sb = ui.render_layout(app, "./project/static/layout.js")

  # Create a webpack file and bundle our javascript from pyxley.utils import Webpack wp = Webpack(".") wp.create_webpack_config(

    "layout.js", "./project/static/", "bundle", "./project/static/"

) wp.run()

  @app.route('/', methods=["GET"]) @app.route('/index', methods=["GET"]) def index():

    _scripts = ["./bundle.js"] css = ["./css/main.css"] return render_template('index.html',

      title=TITLE, base_scripts=[], page_scripts=_scripts, css=css)

  if __name__ == '__main__': app.run()

`wp.run()` will transpile "./project/static/layout.js" with the necessary dependencies and produce "bundle.js". If you had further dependencies not managed by NPM, you could include them in the `base_scripts` keyword argument.

Now when you run `app.py` from the `project` folder, accessing your localhost on port 5000 will lead to a simple plot. This example was adapted from the metricsgraphics example in the Pyxley repository.

CHAPTER 4

Pyxley In Production

Now that we've built an app, how do we deploy it? Because Pyxley is built on top of Flask, the process for deploying our app is no different than deploying any other Flask app. Rather than cover the basics of deploying web apps, this guide will cover some patterns and address some of Pyxley's capabilities.

Data and DataFrames

Pyxley was developed specifically with Pandas in mind. Each widget has methods for transforming dataframes into JSON objects that the different JavaScript libraries can interpret. It's not entirely obvious how to deal with issues such as reloading data.

Updating Data

Let's revisit our MetricsGraphics example, specifically the `LineChart` object. Recall when we created the object, we passed it a dataframe and it looked something like the snippet below.

:: # sf is our dataframe # fig is our figure object lc = LineChart(sf, fig, "Date", ["value"], **kwargs)

When the dataframe, `sf`, is passed into the constructor it is passed into a function that Flask will use any time it needs the chart data.

In the __`init`__ method you will find the following snippet.

::

if not route_func:

def get_data(): args = { } for c in init_params:

if request.args.get(c): args[c] = request.args[c]

else: args[c] = init_params[c]

return jsonify(LineChart.to_json(

self.apply_filters(df, args), x, y, timeseries=timeseries

))

route_func = get_data

The default behavior for `LineChart` is to use this basic route function. Let's focus on the data that gets returned for a moment. Notice we call `jsonify` which is a Flask function that turns a python `dict` into a JSON object. The input `dict` is returned by the `LineChart.to_json` method. `to_json` takes a dataframe as the input as well as a few other variables. `self.apply_filters` is a method belonging to the `Chart` base class that takes a dataframe and a `dict` containing our filtering conditions.

As far as default behavior, this is perfectly reasonable, but it makes a pretty rigid assumption that the underlying data will not change. This default route function is created when the `LineChart` object is initialized and changing the data would require us to recreate the object. In practice, it's not a good idea to have a dependency that requires an app to be restarted.

Now notice the first line of our snippet: `if not route_func`. `route_func` is a keyword argument in the `__init__` method. If a developer provides their own route function, they are no longer restricted by the default behavior.

Reloading Dataframes

Let's assume that we are perfectly comfortable with using Pandas as our data source, but we would like to be able to refresh the data on some cadence.

Returning to our `LineChart` example, what if created it in the following way:

:: # sf is our dataframe # fig is our figure object lc = LineChart(sf, fig, "Date", ["value"],

route_func=some_other_route, **kwargs)

Now that we have passed in `some_other_route` it will be used instead and `sf` will be ignored. So let's start building what the function looks like.

::

class MyDataWrapper(object): """ We are building a simple wrapper for our data """ def __init__(df, x, y, timeseries=True):

self.df = df self.x = x self.y = y self.timeseries = timeseries

def my_route_function(self): # put args check here return jsonify(LineChart.to_json(

Chart.apply_filters(self.df, args), self.x, self.y, timeseries=self.timeseries

))

So the first thing you should notice is that this looks pretty similar to our default `get_data` function from above. The most obvious difference is that now most of the variables are now member variables of our `MyDataWrapper` class. The main benefit from this is that now we are calling a method in our `MyDataWrapper` object that also manages the state of our dataframe `self.df`. Now that `LineChart` no longer has anything to do with our dataframe and how to parse it, we are free to reload our data. You could imagine adding a `set_data` method to our class that has logic about how and when to reload data. The snippet below shows how to modify our example. We will still pass in all of the necessary chart options, but the important logic is now handled by our new object.

:: # sf is our dataframe myData = MyDataWrapper(sf, "Date", ["value"]) # fig is our figure object lc = LineChart(sf, fig, "Date", ["value"],

route_func=myData.my_route_function, **kwargs)

Databases

What if your data is too big. Say, for example, you would like to use `guincorn` and the idea of replicating your data across multiple processes just isn't feasible. If you are using a relational database that works with `SQLAlchemy` there's not much to change. The snippet below shows one possible change to our data wrapper from above. It's also important to remember that we just need to format our data for the JavaScript libraries. We can freely customize the methods to accommodate any data source.

::

class MyDataWrapper(object): """ We are building a simple wrapper for our data """ def __init__(sql_engine, x, y, timeseries=True):

self.engine = sql_engine self.x = x self.y = y self.timeseries = timeseries

def get_data(self, args): # make some sql query or use SQLAlchemy functions return pd.read_sql('select * from table', self.engine)

def my_route_function(self): # put args check here df = get_data(args) return jsonify(LineChart.to_json(

Chart.apply_filters(df, args), self.x, self.y, timeseries=self.timeseries

))

REST APIs

What if you want to hit some other API? Use `requests`! Then it's the same game of just leveraging the other functions to get the data in the format the chart needs.

CHAPTER 5

About

Read more about Pyxley at the Multithreaded blog.

CHAPTER 6

Features

- Simple web-applications using Flask, PyReact, and Pandas
- Integration with d3.js based libraries such as MetricsGraphics
- Ability to integrate custom React UIs

CHAPTER 7

Installation

Install Pyxley by running:

```
pip install pyxley
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search