**Building an Open Source, Multitiered Application**

In software engineering, multitier architecture (or multilayered architecture) is an architecture in which presentation, application processing and data management are physically separated.

Often you will see architectural diagrams depicting the three “tiers” as:

* **The Data Tier** (where information is stored in a database or file system)
* **The Logical Tier** (where processing commands are executed, logical decisions are made, and calculations may be performed)
* **The Presentation Tier** (which is the “topmost” tier and is where results are transformed and presented into something the user can understand)

For illustration, let us suppose we have some data residing in multiple source systems and we want to “gather” that data and then present it in some sort of visual dashboard.

**The Data Tier**

In this example, the data resides in both an IBM Planning Analytics cube as well as Microsoft SQL server tables. I will not spend too much time here – just note that we have a cube defined in Planning Analytics named “Shipments” that is updated with the number of shipments for each day in the prior week and in SQL Server, there are tables named “Brands” which holds (among other things) the brand names of the products that were shipped, and a table named order\_items that holds quantities (shipped).

**The Logical Tier**

We will use Python to build our logical tier which will connect to and query both of our named data sources and then format and make the results available to the presentation tier. To build our middle tier solution, will use [PyCharm](https://www.jetbrains.com/pycharm/) starting with a new Python project. The “script” will be pretty simple, divided into several distinct “parts”:

* Python Package Imports
* Function Definitions
* Main Processing

**Python Packages**

In this example we are going to leverage the following Python modules:

import configparser # -- a Python class used to read configuration .ini files

import pyodbc # --- an open source Python module for accessing ODBC databases.

import pandas as pd # --- python data analysis library

from TM1py.Services import TM1Service # --- python library of functions that wraps TM1 REST API

**Function Definitions**

Following best practices, we will create a series of simple Python functions. The first two will handle connecting to our data sources and the next two will retrieve the data that we are interested in:

def **connect\_to\_sql\_server**(l\_sql\_server):

# --- connect to sql server and return active connection

sql\_server\_connection = pyodbc.connect('Driver={SQL Server};'

'Server=' + l\_sql\_server + ';'

'Database=JimSample;'

'Trusted\_Connection=yes;')

return sql\_server\_connection

def **connect\_to\_planning\_analytics**(pa\_server):

# --- connect to planning analytics server and return active connection

config = configparser.ConfigParser()

config.read(r'config.ini')

tm1\_source\_server = pa\_server

planning\_analytics\_connection = TM1Service(\*\*config[tm1\_source\_server], timeout=3600)

return planning\_analytics\_connection

def **get\_brands\_from\_sql\_Server**(l\_my\_sql\_connection):

# --- use connection and query sql server table and return results in a pandas data frame

sql\_stmt = 'SELECT production.brands.brand\_name, sum(sales.order\_items.quantity)' + \

' FROM production.brands INNER JOIN ' + \

' production.products ON production.brands.brand\_id = production.products.brand\_id INNER JOIN ' + \

' sales.order\_items ON production.products.product\_id = sales.order\_items.product\_id AND ' + \

' production.products.product\_id = sales.order\_items.product\_id ' + \

' group by production.brands.brand\_name'

sql\_query\_results = pd.read\_sql\_query(sql\_stmt,l\_my\_sl\_connection)

return sql\_query\_results

def **get\_daily\_shipments**(l\_my\_pa\_connection):

# --- build MDX to query Shipment’s cube

mdx = f"""

SELECT

TM1SORT({{TM1SUBSETALL([Days])}},ASC) ON ROWS,

TM1SORT({{TM1SUBSETALL([Daily Shipments])}}, ASC) ON COLUMNS

FROM [Shipments]

"""

# --- use connection to pass mdx to TM1py function to return list of cell values

data = l\_my\_pa\_connection.cells.execute\_mdx\_values(mdx=mdx)

# --- convert cell values to a python list object and return it

daily\_shipment\_list = list(data)

return daily\_shipment\_list

Next, we need some functions to support the presentation of the data. The first is the simplest and will simply generate an HTML page (rows\_and\_columns.html) from the data retrieved from planning analytics:

def **create\_rows\_and\_columns\_html**(my\_cell\_set):

f = open('rows\_and\_columns.html', 'w')

message = """<html>

<head><link rel="stylesheet" type="text/css" href="gridtable.css"></head>

<font size="2" face="verdana"><h2><b><center>Daily Shipments</h1>

<table align="center" class="gridtable" width="200">

<TR><th>Day of Week</th><th>Shipments Total</th></TR>

<TR><td >Monday</td><td align="right">""" + str(my\_cell\_set[0]) + """</td></TR>

<TR><td>Tuesday</td><td align="right">""" + str(my\_cell\_set[1]) + """</td></TR>

<TR><td>Wednesday</td><td align="right">""" + str(my\_cell\_set[2]) + """</td></TR>

<TR><td>Thursday</td><td align="right">""" + str(my\_cell\_set[3]) + """</td></TR>

<TR><td>Friday</td><td align="right">""" + str(my\_cell\_set[4]) + """</td></TR>

<TR><td>Saturday</td><td align="right">""" + str(my\_cell\_set[5]) + """</td></TR>

<TR><td>Sunday</td><td align="right">""" + str(my\_cell\_set[6]) + """</td></TR>

</table></html>"""

f.write(message)

f.close()

The next function, will be even easier; and it will simply generate a JavaScript variable definition file (pie\_data.js) based upon the data:

def **create\_data\_for\_pie\_chart**(my\_cell\_set):

f = open('pie\_data.js', 'w')

message = """ var pie\_data = {Mon:""" + str(my\_cell\_set[0]) + """

, Tue: """ + str(my\_cell\_set[1]) + """, Wed: """ + str(my\_cell\_set[2]) + """

, Thu: """ + str(my\_cell\_set[3]) + """, Fri: """ + str(my\_cell\_set[4]) + """

, Sat: """ + str(my\_cell\_set[5]) + """, Sun: """ + str(my\_cell\_set[6]) + """} """

f.write(message)

f.close()

And for our last function (similar to the one above), it will create another JavaScript variable definition file (bar\_data.js):

def **create\_data\_for\_bar\_chart**(sql\_query\_results):

f = open('bar\_data.js', 'w')

message = """ data = [

{label: """ + """" """ + str(sql\_query.iat[0,0]) + """", value: """ + str(sql\_query.iat[0,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[1,0]) + """", value: """ + str(sql\_query.iat[1,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[2,0]) + """", value: """ + str(sql\_query.iat[2,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[3,0]) + """", value: """ + str(sql\_query.iat[3,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[4,0]) + """", value: """ + str(sql\_query.iat[4,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[5,0]) + """", value: """ + str(sql\_query.iat[5,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[6,0]) + """", value: """ + str(sql\_query.iat[6,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[7,0]) + """", value: """ + str(sql\_query.iat[7,1]) + """},

{label: """ + """" """ + str(sql\_query.iat[8,0]) + """", value: """ + str(sql\_query.iat[8,1]) + """}

];"""

f.write(message)

f.close()

Lastly, we need a “main” section to coordinate the use of the above functions:

# --- establish a connection to IBM planning analytics

my\_pa\_connection = connect\_to\_planning\_analytics('PA\_Sample')

# --- if connection successful then query PA cube to create a pandas DataFrame

if my\_pa\_connection:

my\_cell\_set = get\_daily\_shipments(my\_pa\_connection)

# --- generate simple HTML page

create\_rows\_and\_columns\_html(my\_cell\_set)

# --- generate the data for the pie chart

create\_data\_for\_pie\_chart(my\_cell\_set)

# --- establish a connection to Microsoft sql server

my\_sl\_connection = connect\_to\_sql\_server('QBIT-5P481N2\SQLEXPRESS')

# --- if connection successful then query sql server table to create a pandas DataFrame

if my\_sl\_connection:

sql\_query = get\_brands\_from\_sql\_Server(my\_sl\_connection)

# --- generate the data for the bar chart

create\_data\_for\_bar\_chart(sql\_query)

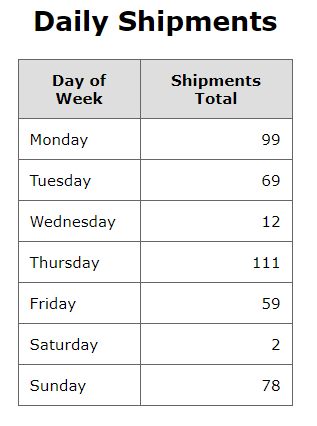
**The Presentation Tier**

For the presentation tier, we will follow the same strategy – to do as little programming as possible – by using simple HTML pages and leveraging some open source JavaScript.

Since this example assumes that the data always “comes back” from the middle or “logical” tier in the same format, we do not have to do any programming in this tier and can just “let” the code in the HTML page present the data.

**Rows and Columns**

For the first visualization, we can simply open the page created by the create\_rows\_and\_columns\_html function (rows\_and\_columns.html) in a web browser:



**The Pie Chart**

Rather than just “boring” rows and columns, to present the data in a slightly nicer format, we can use [D3.js](https://d3js.org/), which is an open source JavaScript library for manipulating documents based on data. D3 helps you “bring data to life” using HTML, SVG, and CSS. D3’s emphasis on web standards gives you the full capabilities offered in web browsers without having to produce custom code or tying yourself to a proprietary framework, combining powerful visualization components in a data-driven approach.

What all that means is that we can simply add some <script> tags to an HTML page to specify the URLs of some external script (the D3 libraries) as well as the data generated by the create\_data\_for\_pie\_chart function and voilà!

Below is the HTML for the pie chart page:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>PA Pie Chart</title>

<!-- Load d3.js -->

<script src="https://d3js.org/d3.v4.js"></script>

<!-- Create a div where the graph will be shown -->

<div id="my\_dataviz"></div>

<!-- Add the Color scale -->

<script src="https://d3js.org/d3-scale-chromatic.v1.min.js"></script>

<!-- Planning Analytics data -->

<script src="pie\_data.js"></script>

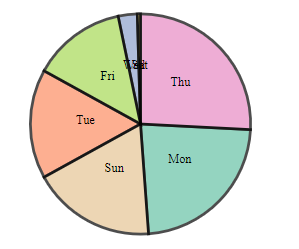
<!-- Create the pie chart -->

<script src="pie\_chart.js"></script>

</head>

</html>

And if we open our HTML pie chart page we see:



**A Bar Chart**

Again, without any presentation tier programming, lets now present the data we retrieved from SQL Server. The following shows the HTML page that presents the data in a bar chart:

<!DOCTYPE html>

<HTML>

<meta charset="utf-8">

<head>

<link rel="stylesheet" href="mystylesheet.css">

</head>

<body>

<font size="2" face="verdana"><h2><b><center>Bar Chart Example</h1>

<script src="https://d3js.org/d3.v3.min.js"></script>

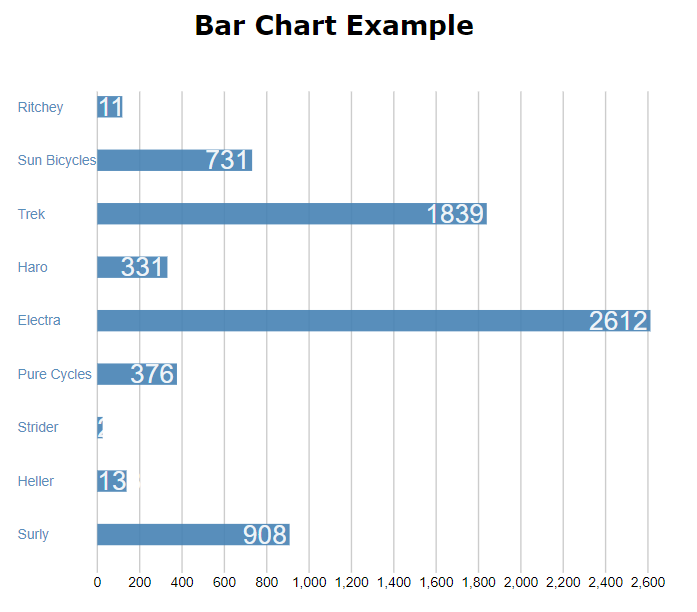
<script src="bar\_data.js"></script>

<script src="create\_bar\_chart.js"></script>

</body>

</HTML>

Which looks like this (opened in a Web browser):



**Wrap Up**

Okay, first some “disclaimers”.

For example, I have not addressed security anywhere and the code assumes that all of the files are in the same location (to save time, I have also referenced some of the D3 files directly from https://d3js.org). Additionally, I took the liberty of assuming what data is retrieved from the source systems, in that daily shipments are always just seven numbers (one quantity for each day of the prior week) and I have also limited the SQL Server data to seven brands. In a “real” scenario, you would have to add some additional logic to deal with any unexpected data situations. Finally, to be fair, some of the Python code would require some “cleaning up” before it could be considered productized.

What is exciting is that by using simple HTML and D3 library functions, we have done very little actual programming – all in the middle or logical tier – and have created a working multitier solution that demonstrates how an open source (Python) application can connect to backend source systems such as Planning Analytics and SQL Server and then “push” data to the presentation tier where it is displayed by standard HTML pages.