Creating an HTML/Javascript UI for a Data Science Module

Contents

Introduction

This document is intended for software developers who will assist Data Scientists in building a complete Data Science web application. This document demonstrates how an HTML/Javascript User Interface (UI) can be used as a front end for a Data Science module. The UI will allow the user to interact with the Data Science module to build, test and use the model for predictions. The projects associated with this document serve as a proof of concept that once a Data Scientist writes a model with the appropriate APIs, a software developer can create a UI that allows the user access to the model, and this model together with the UI form a complete web application.

Why a Web Application?

If we wish to make an application available to users, we have three basic choices:

1. Install the application on the users' machines (Desktop app).
2. Install a UI on the users' machines and make REST calls to services hosted on a server.
3. Make the application available as a web app.

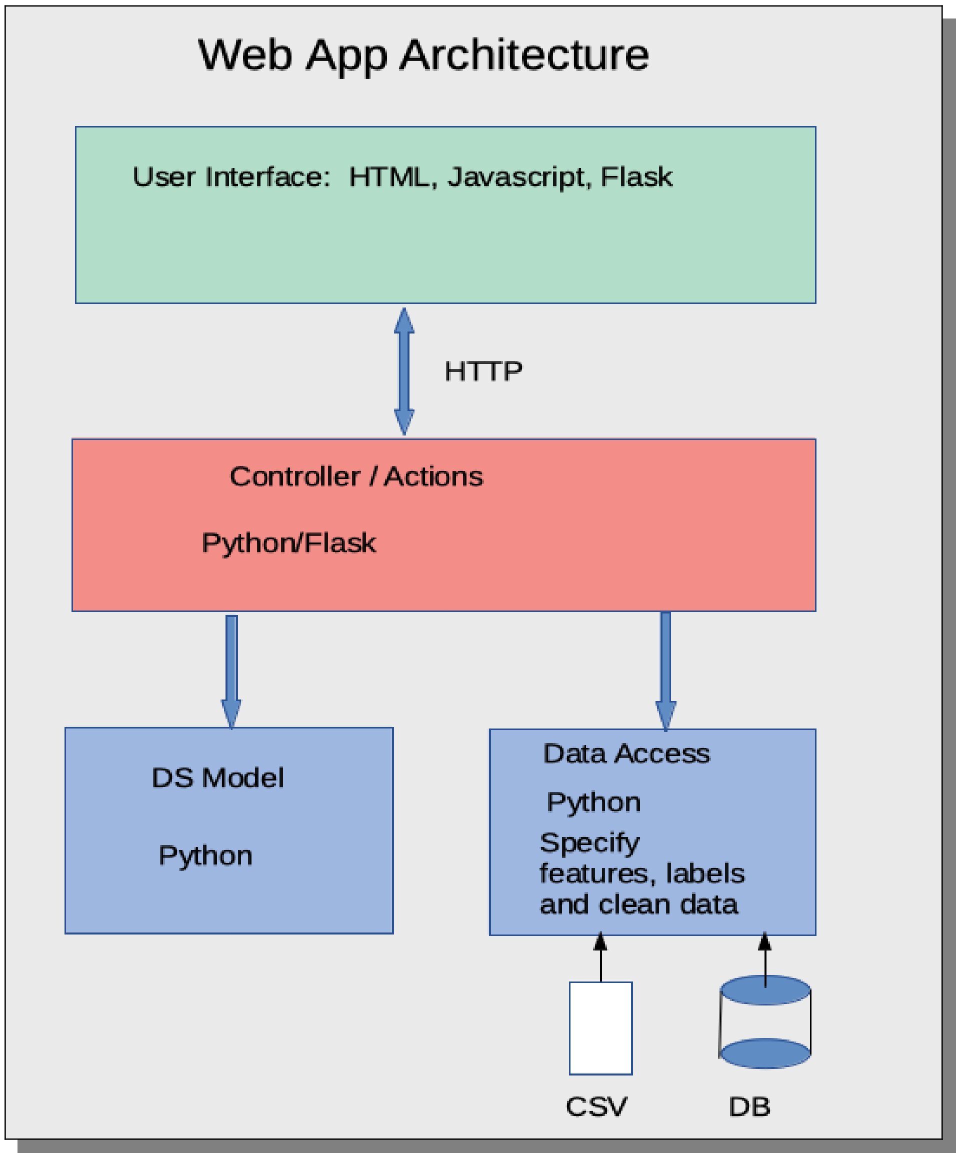
Choice (1) involves configuration of every user's machine. Experience has shown that software developers would like to avoid configuration whenever possible. The more users, the more configuration. Any changes to the UI or business logic requires reinstallation of the application on the user's machine. This choice is just not acceptable when you have a large number of apps to deploy. Most importantly, this choice is not able to make sufficient use of the accepted standard of a CI/CD pipeline for development and deployment.

Choice (2) still brings configuration issues, but to a much lesser extent since the dependencies to support a UI are less numerous than dependencies to support a whole application. The backend REST services can make use of a CI/CD pipeline, but the front end UI can not make use of the CD part of the pipeline since it is installed on the user's machine.

Choice (3) is the most attractive approach because it allows user access to the application simply through a web browser. There is no client side configuration because all browsers contain support for HTML and Javascript. The Data Science and Data Access modules can be written as either REST services, or as API's that can be called directly by object instantiation. Of course, development and deployment of both the UI and backend modules can make use of the CI/CD pipeline concept.

NOTE: If we were to install an application in a container, the only reasonable way that we can offer a common user interface is to make the application a web application.

The Application Architecture



It is strongly recommended that web applications follow the architecture displayed in the image on the right. This architecture has been around for a number of decades, and it has served web developers well. (Actually, this architecture also works for other non web applications)

There are four components or modules that make up the application. The two blue modules represent the code that the Data Scientist would write. The DS module contains the ML or AI model. The blue modules are ***completely independent from the context in which they are used.*** That is they have no dependence upon the web framework in which they are used, and they have no dependence upon the Controller/Actions as well as the UI. Simply stated, the blue modules can be used in any application context. They could be used locally in a desktop application, they could be used in a cloud application, they could be used in any web framework, they could be used in combination with any UI.

The implementation of this architecture is executed on at least two separate computers. The user interface (green) is presented in a browser which runs on the user's computer. A browser is a powerful application that:

- Renders HTML into a Graphical User Interface (GUI).
- Optionally executes Javascript in order to give functionality to the GUI.
- Generates HTTP requests that are sent to the server specified in the URL of the request
- Receives and decodes HTTP responses that can be displayed in the GUI

The URL that is contained in the HTTP request identifies the particular server where the request will be fulfilled. The request often contains an action name together with optional parameters that the server executes. The execution of this server action can generate an HTTP response that is sent back to the browser where the HTTP request came from. The browser then decodes the response and updates the GUI.

As is shown in the Web App Architecture, the code that forms the GUI is written in HTML and Javascript. This code that defines the GUI is commonly called "The Front End". The code that is executed on the server is commonly called "The Back End". (If a person has both front and back end code skills together with database skills, they are known as a "Full Stack Developer".)

The code on the Back End typically accomplishes the following tasks:

- Receives and decodes the HTTP request.
- Executes the action that is mapped to the URL specified in the HTTP request. The module that maps the URLs to actions is called the Controller.
- In a Data Science Web Application, the actions would be found in the blue DS Model and Data Access modules
- If needed, builds and returns an HTTP response to the browser that made the request.

The handling of the HTTP request and response protocol involves a great deal of code. For example, HTTP relies on the TCP standard which is connection based. Fortunately, all this code has been encapsulated into products called Web Frameworks, and by virtue of Object Oriented Design, all the details are hidden and out of view by users of the Web Frameworks. As expected with an Object Oriented package, the Frameworks provide useful API's.

Web Application Frameworks

All web applications must make use of a web framework that offers support for the following:

- Handling HTTP requests and responses

- Handling of URL/HTML page mappings
- Handling of URL/Controller Action mappings

In all web applications, an action is triggered by the URL that appears in the HTTP request. The web application framework maps URLs to functions or class methods that carry out the actions.

For Python web applications there are many web frameworks that are available. To name a few, we have: Django, Pyramid, TurboGears, Web2Py and Flask. In this document we will use Flask. Flask is a lightweight web framework that is simple to use for development, but when put into an OpenShift container together with a web server, can also be used in production.

NOTE: For convenience, we can develop and run our examples on our developer machine instead of using a separate server deployment (see Note on Running Web App Examples).

The Flask Web Application Framework

We will start with some simple examples of how to create a web application using the Flask frameworks and then incrementally add more usable features.

First a note about the Flask application object that we create in all Flask applications. Flask, like all other web frameworks has a set of default values. Of importance to us are the default directories that Flask uses. If we use the defaults, files must be stored in specific directories:

1. **HTML files must be placed in a directory named ~/templates.**
2. **The Javascript and CSS files are stored in a directory named ~/static**
3. **The application Controller, application.py must be placed in the project root: ~/**

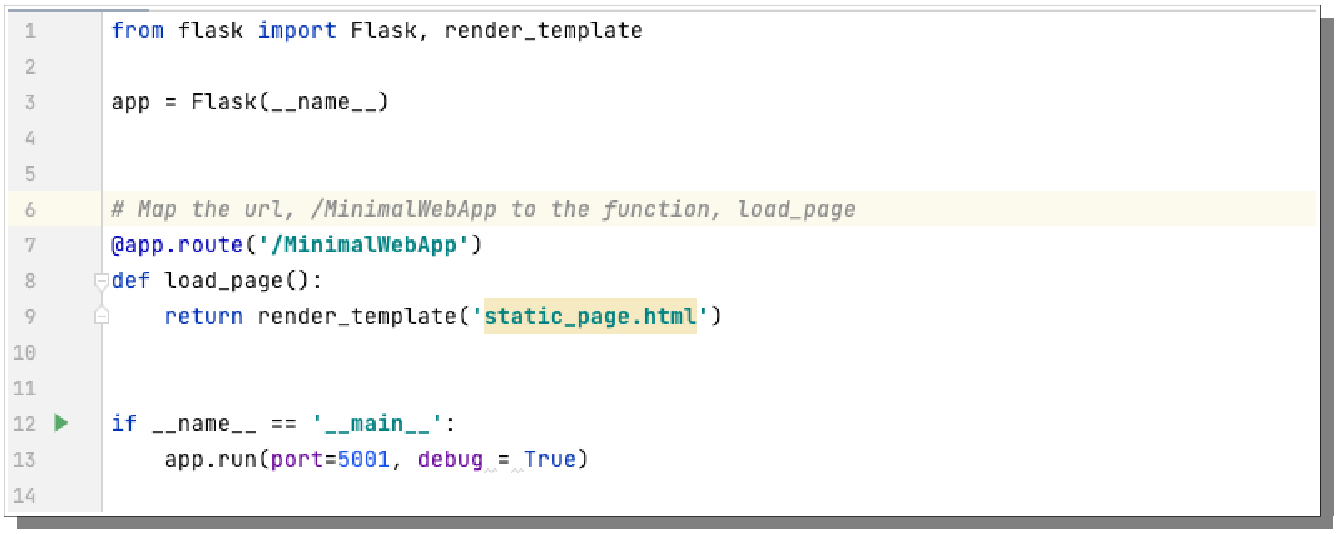
You can find out more about Flask's defaults and how to change them here.

The code for the examples that follow in this document can be obtained from a Git Repository. Please see the Appendix at the end of this document.

Example: Minimal Web App With Static Page

As a first example, we will build a minimal web app. The code for this example is found in the Git Repository with the project name of **Minimal**. This web app simply generates a static web page. By static we mean that the content of the web page returned from the server remains the same whenever the page is served.

Since Flask is a simple framework, we can put to work most of the Flask support mentioned above into one Python file, usually named application.py, or sometimes just app.py. This Python file forms the Controller module in our architecture. An example of a Flask application that serves a static web page looks like:



```
1   from flask import Flask, render_template
2
3   app = Flask(__name__)
4
5
6   # Map the url, /MinimalWebApp to the function, load_page
7   @app.route('/MinimalWebApp')
8   def load_page():
9       return render_template('static_page.html')
10
11
12  if __name__ == '__main__':
13      app.run(port=5001, debug = True)
14
```

Assuming that we have installed the **flask** package, we can import the Flask class and instantiate a Flask object that we call **app** on line 3. The global variable, **__name__** is the usual Python variable that holds the name of the Python application when it is run. When we run app.py from the command line with: **python app.py**, the variable **__name__** gets assigned the value '**__main__**'. This is a Python characteristic, not a Flask one.

Line 12 checks to see if the Python file is being run with the Python interpreter. If so (ie. __name__ == '__main__'), then we call on Flask's method, run(). The two parameters, **port** and **debug** are set to 5001 and True respectively. The newly started Flask server will listen on port 5001 for HTTP requests, and debug = True allows us to use the runtime debugger on the application. (NOTE: The port and debug designations are used for development only. In production, the port is set by the server's config files.)

Line 6 uses a Python decorator on the function **load_page()**. Recall that a Python decorator is itself a function that calls upon the function that it decorates. In addition, our decorator called **route** receives one parameter whose value is the URL '**/MinimalWebApp**'. The value of this parameter, '**/MinimalWebApp**' is called an **endpoint**. The decorator maps the endpoint to the code (**load_page()** function) that will be executed when the endpoint is recognized. A web application will have many endpoints, each one representing some action that will be carried out by the application.

In our simple hello app, the decorator @app.route('/') is equivalent to the code:

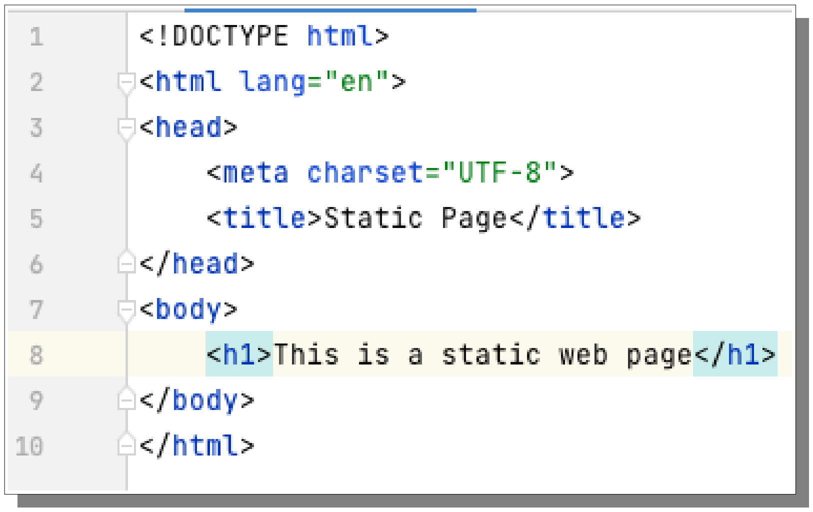
app.add_url_rule('/MinimalWebApp', 'load_page', load_page)

So the decorator just adds a mapping(Python dictionary) to the Flask object. The key used in the mapping is the string name of the function that gets decorated (**load_page**). For a deeper explanation of the decorators used in Flask, please use this link.

The function, load_page() has one line of code:

```
return render_template('static_page.html')
```

The function, **render_template** returns the content of the static_page.html file:

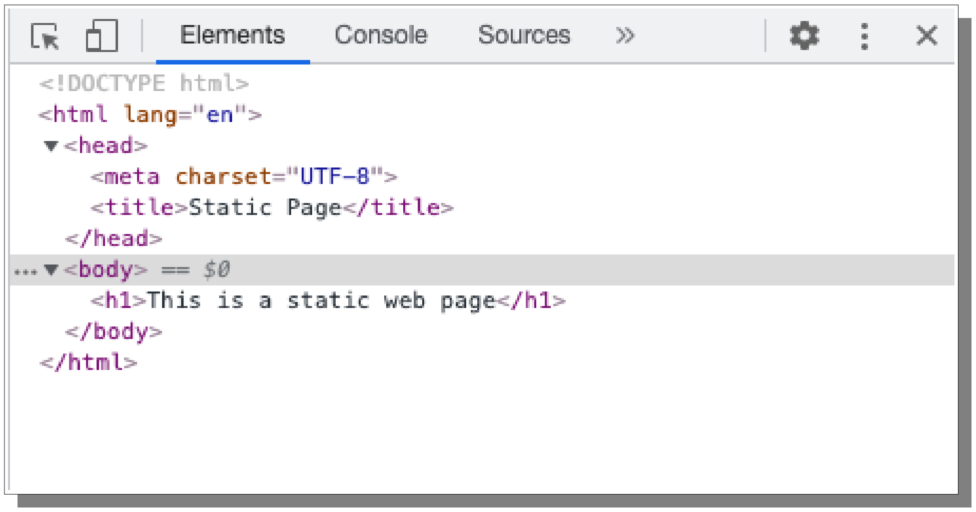


```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Static Page</title>
</head>
<body>
    <h1>This is a static web page</h1>
</body>
</html>
```

If we were to run the above code from the command line with: **python app.py**, the Flask server starts and listens on port 5001. We can view the simple results of our app by opening a browser and typing the URL: **localhost:5001/MinimalWebApp**

You will see the output text: “This is a static web page” in the browser.

You can also view the raw HTML that the browser received. For example, in Chrome, open the Chrome menu and choose “More Tools | Developer Tools”. Click the “Elements” tab and view the HTML:



```html
<!DOCTYPE html>
<html lang="en">
▼<head>
    <meta charset="UTF-8">
    <title>Static Page</title>
  </head>
…▼<body> == $0
    <h1>This is a static web page</h1>
  </body>
</html>
```

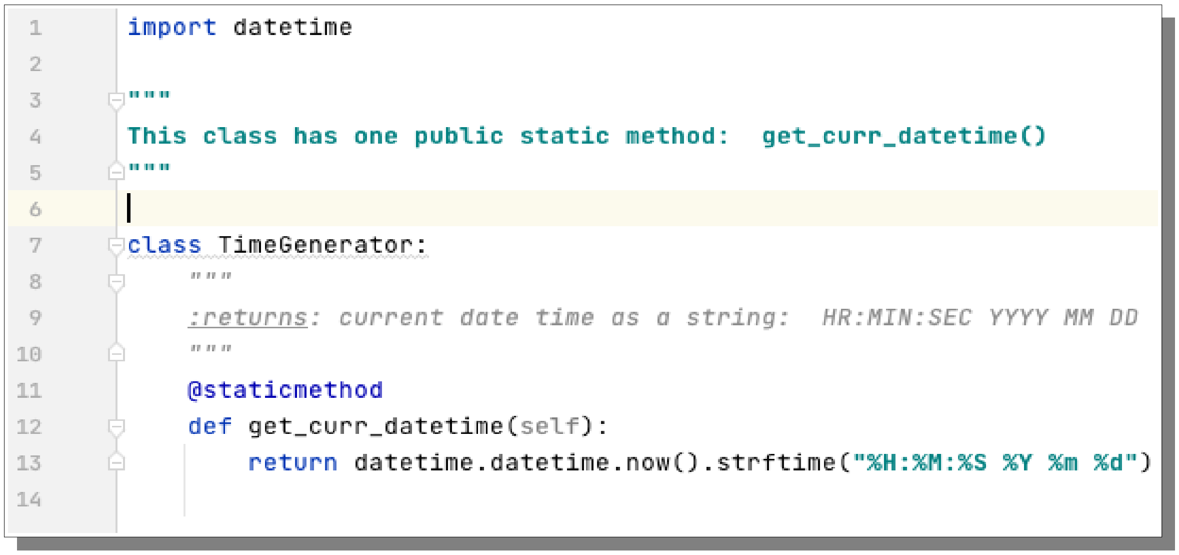
<u>Example: Minimal Web App With Date Time</u>

Now that we see how a minimal static web page is served, let’s increase our insight by adding a new feature. This time we will display the current server time in the web page. Each time the web page is loaded, it will display the time on the server *when the page was rendered*. This example is found in the Git Repository with project name, **MinimalWithTime**.

There are two new concepts that we will introduce:

1. Use a separate class to generate the date time string.
2. Add a named parameter to the render_template function that passes the date time string to the html file.

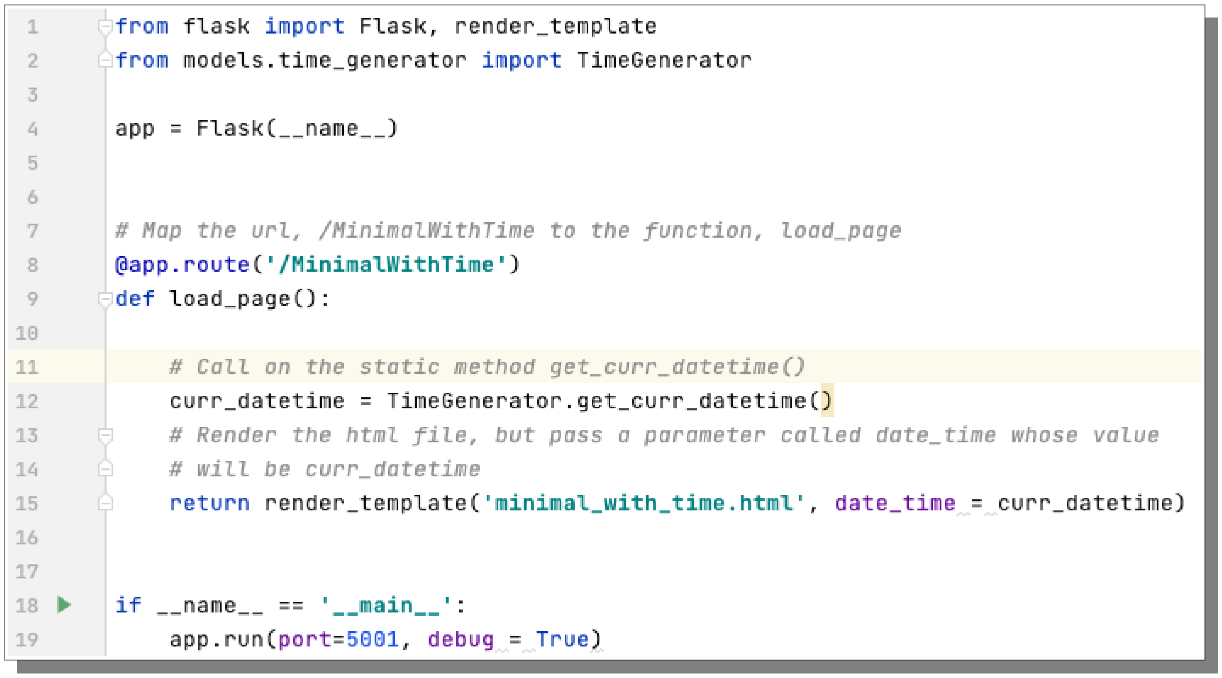
First let's create a class called TimeGenerator that will simply get the current datetime and format it into a string:



```python
import datetime

"""
This class has one public static method:  get_curr_datetime()
"""

class TimeGenerator:
    """
    :returns: current date time as a string:  HR:MIN:SEC YYYY MM DD
    """
    @staticmethod
    def get_curr_datetime(self):
        return datetime.datetime.now().strftime("%H:%M:%S %Y %m %d")

```

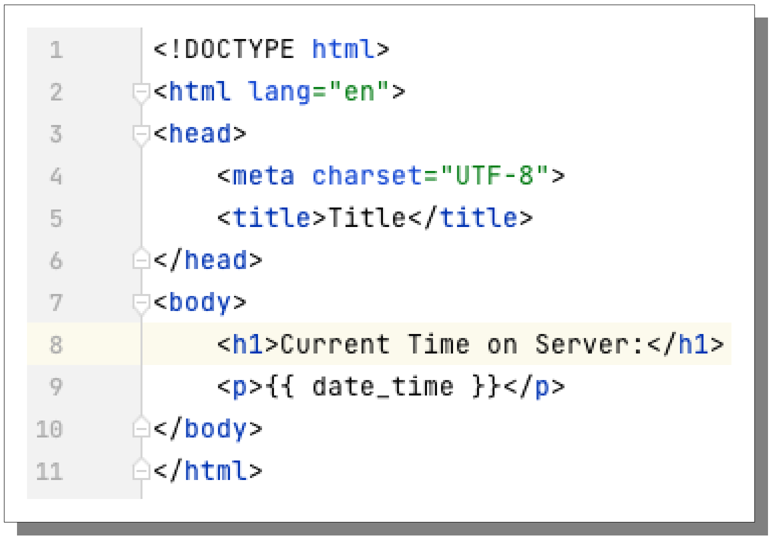
Notice that the method get_curr_datetime() is a static method, which means that it can be called directly without having to create a TimeGenerator instance.

Then in the controller's load_page() function we make the call to generate the formatted datetime:



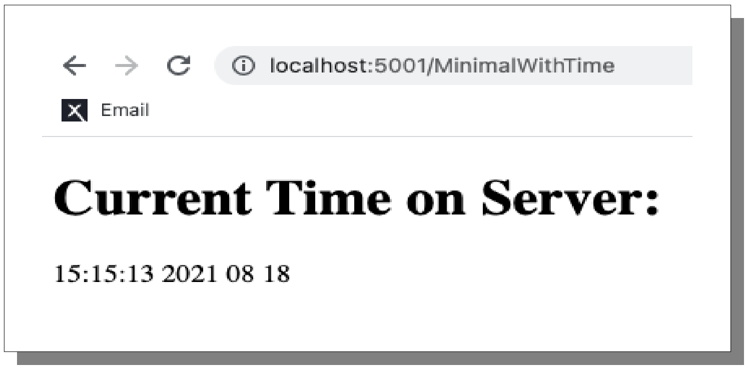
```python
from flask import Flask, render_template
from models.time_generator import TimeGenerator

app = Flask(__name__)


# Map the url, /MinimalWithTime to the function, load_page
@app.route('/MinimalWithTime')
def load_page():

    # Call on the static method get_curr_datetime()
    curr_datetime = TimeGenerator.get_curr_datetime()
    # Render the html file, but pass a parameter called date_time whose value
    # will be curr_datetime
    return render_template('minimal_with_time.html', date_time = curr_datetime)


if __name__ == '__main__':
    app.run(port=5001, debug = True)
```

Notice that in line 15 we have added a new parameter named date_time in the call to render_template and assign to it the value of curr_datetime that was obtained from the TimeGenerator class. With the addition of parameters, render_template() will render the html page AND insert the values of the passed parameters into the html. This insertion of values into the html page is done by using placeholder symbols in the html page and inserting the parameter named date_time:



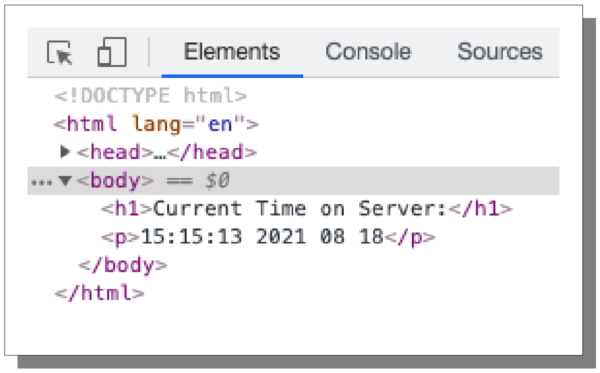
```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <title>Title</title>
6   </head>
7   <body>
8       <h1>Current Time on Server:</h1>
9       <p>{{ date_time }}</p>
10  </body>
11  </html>
```

In line 9 we have added a placeholder and have specified its value to be that of the passed parameter, date_time. When the page is rendered, we see the date time:



Current Time on Server:

15:15:13 2021 08 18



```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
…▼<body> == $0
    <h1>Current Time on Server:</h1>
    <p>15:15:13 2021 08 18</p>
  </body>
</html>
```

The image above right shows the elements view in the browser. Notice how the date time has been inserted into the <p> tag.

Before we move on to the next example, let’s return to the code of app.py and focus on the load_page() that is the action carried out at the “/MinimalWithTime” endpoint. You may be wondering why go through the trouble of creating a separate class, TimeGenerator to generate the formatted datetime? Why not just put the datetime generation code directly in load_page() and forget about the class? That code change would certainly work, but would at the same time break our Web App Architecture. Remember that app.py serves as the Controller in our Architecture.

The job of the Controller is primarily to map incoming URL HTTP requests to functions that carry out some task. In order to carry out some tasks, the Controller may have to extract

HTML form values out of the request and pass those values to a class that does the desired work. In addition, the Controller may then receive and return the results of the task. *But the code that does the task itself should not be placed in the Controller.* If we did place that task code in the Controller, it is possible that the code would have Flask dependencies. From an architectural point of view, the code that carries out the task should not have any dependencies on Flask. Instead, that code should be decoupled from the context in which it is used. Another advantage of decoupling is that the TimeGenerator class can be used in other contexts and other applications.

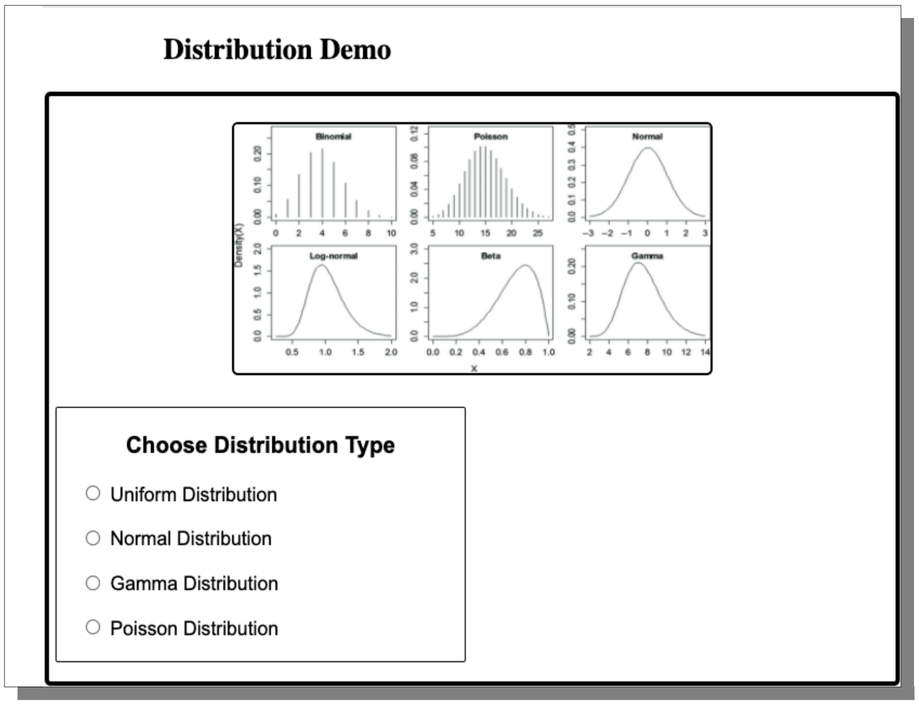
Software developers know that a delivered product most likely will have requested modifications—changes to existing features, new features, even changes in the web framework that is used. For example, if we had to change from Flask to Django, a properly designed architecture would only necessitate a change in the Controller, since all the classes that it uses are decoupled from the Controller. In fact, all of the web examples in this paper can easily be changed to Django just by modifying the Controller, app.py.

Example: Web App With User Interaction

Let's build on the skills we developed in the previous examples by adding two new concepts:

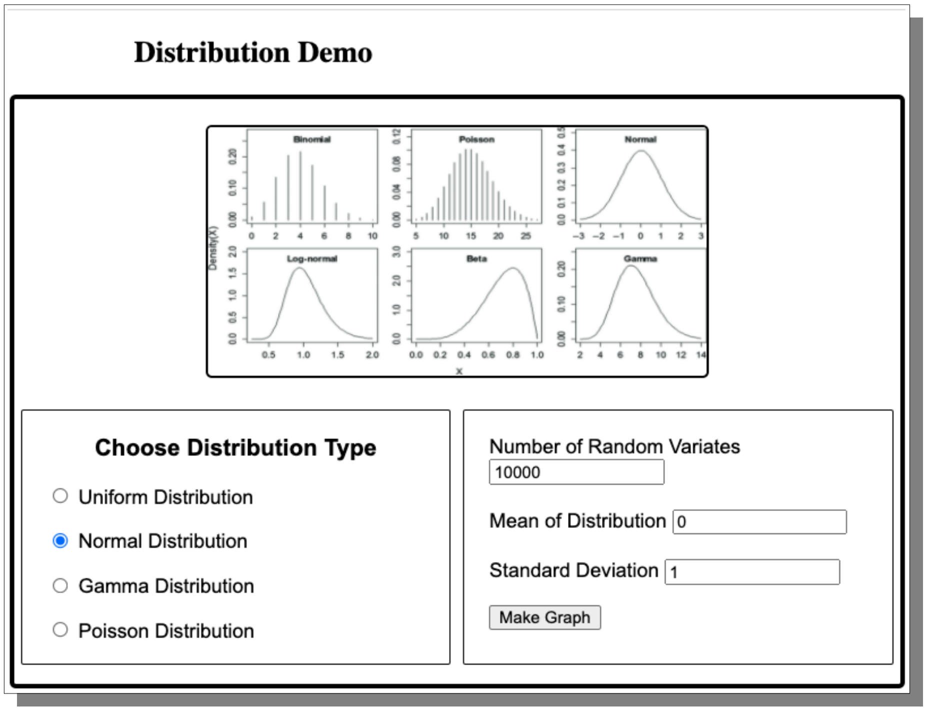
1. Create a user interface that allows the user to input some values that affect the outcome.
2. Make a graph image and send it back asynchronously to the browser.

This example is found in the Git Repository with project name, **Distribution**. In this example we will make a web app that displays one of four types of graphs out of four different types of statistical distributions: Uniform, Normal, Gamma, and Poisson distributions. The user will be able to select which distribution type to graph. Based on the distribution type, the user will be able to provide input values that affect the selected graph. The Initial UI is shown in the figure on the right.



The image at the top of the UI with the six mini graphs is just static and just serves as a decoration for the page. The user interaction starts with the box (div)in the lower left with the title "Choose Distribution Type". So the user is able to select one of the radio buttons that determines the graph type.

For example if the user selects “Normal Distribution”, we must then display a div to the right that offers text input tags in order to collect necessary values to generate the Normal Distribution graph:



Choose Distribution Type

- ○ Uniform Distribution
- ● Normal Distribution
- ○ Gamma Distribution
- ○ Poisson Distribution

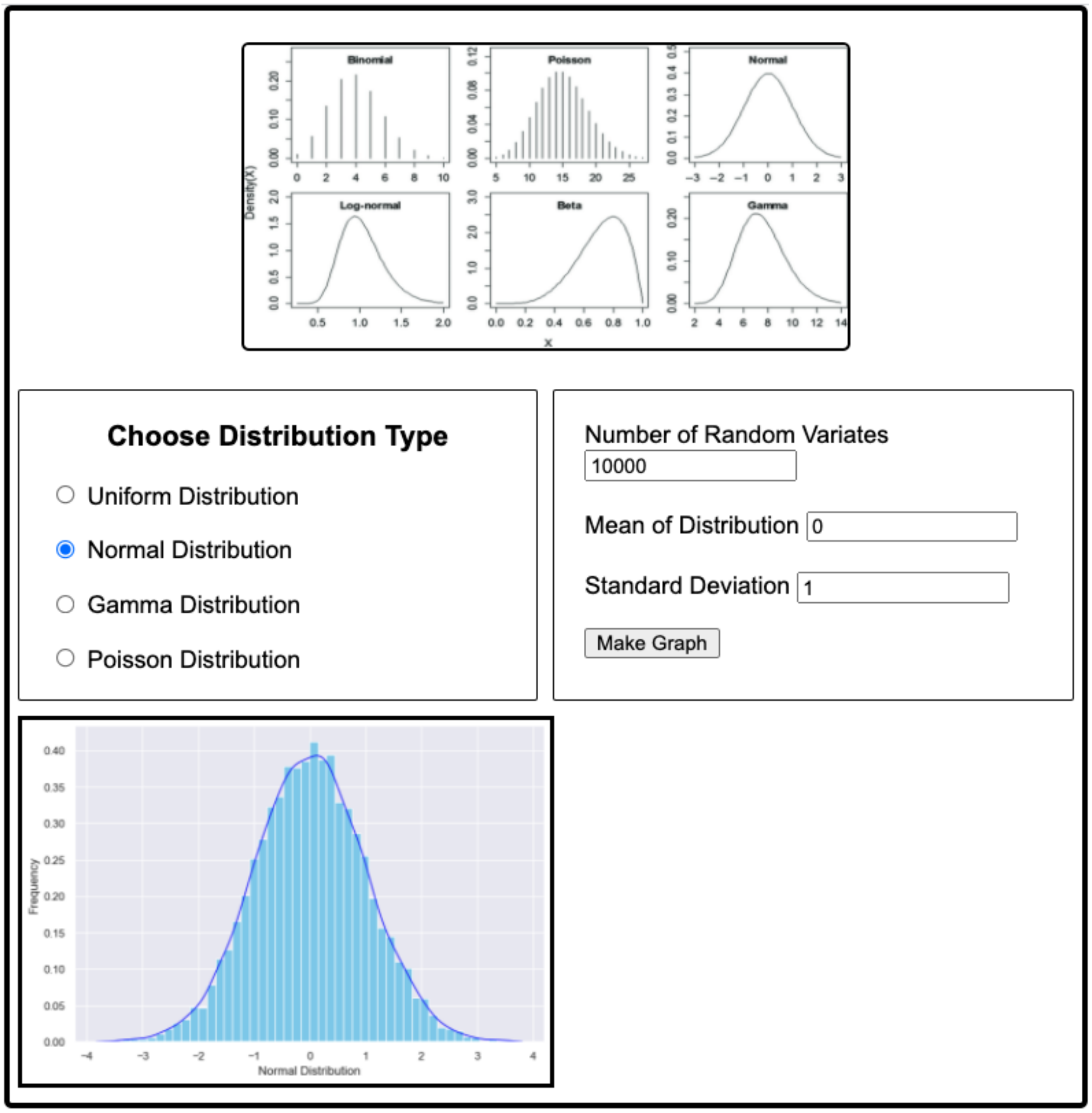
Number of Random Variates 10000

Mean of Distribution 0

Standard Deviation 1

Make Graph

When the user clicks on the button labeled “Make Graph”, a Normal Distribution graph will be displayed at the bottom of the UI:



Choose Distribution Type

- Uniform Distribution
- [✓ Normal Distribution]
- Gamma Distribution
- Poisson Distribution

Number of Random Variates: 10000

Mean of Distribution: 0

Standard Deviation: 1

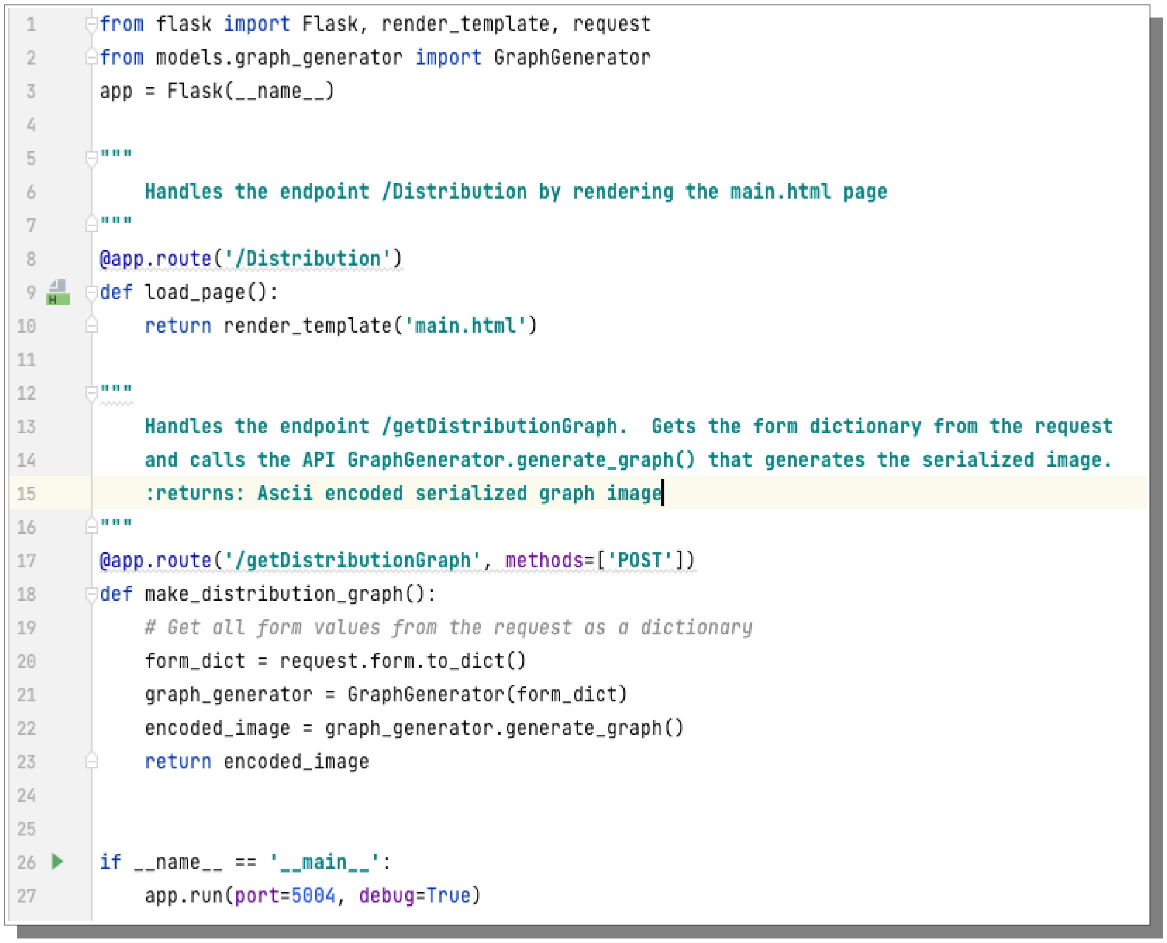
Make Graph

Even though this example does not involve a Data Science model, it does illustrate some of the concepts that we need to master in order to actually implement a DS model. The example shows how to collect user input and then use a class that generates the graph. *It will also show how to asynchronously insert the graph image into the UI without having to reload the whole page.*

All these features will be done using only Python, HTML5 (with CSS for layout) and Javascript. There are no other third party dependencies except for Flask, the web framework.

As per our Architecture, the graph generation code (A class named GraphGenerator) will be placed in a Python package called models.graph_generator.

Even though the UI looks complicated, the Controller, app.py is simple since there are only two endpoints to handle:



```
1  from flask import Flask, render_template, request
2  from models.graph_generator import GraphGenerator
3  app = Flask(__name__)
4
5  """
6      Handles the endpoint /Distribution by rendering the main.html page
7  """
8  @app.route('/Distribution')
9  def load_page():
10     return render_template('main.html')
11
12 """
13     Handles the endpoint /getDistributionGraph.  Gets the form dictionary from the request
14     and calls the API GraphGenerator.generate_graph() that generates the serialized image.
15     :returns: Ascii encoded serialized graph image
16 """
17 @app.route('/getDistributionGraph', methods=['POST'])
18 def make_distribution_graph():
19     # Get all form values from the request as a dictionary
20     form_dict = request.form.to_dict()
21     graph_generator = GraphGenerator(form_dict)
22     encoded_image = graph_generator.generate_graph()
23     return encoded_image
24
25
26 if __name__ == '__main__':
27     app.run(port=5004, debug=True)
```

We now have two endpoints for our web app: /Distribution, and /getDistributionGraph. The function load_page() is associated with the URL: /Distribution and it does the usual job of loading the page, main.html. The function, make_distribution_graph() is associated with the URL: /getDistributionGraph and it calls upon GraphGenerator's generate_graph(). it generates the graph identified by its parameter, graph_type and returns the serialized graph image.

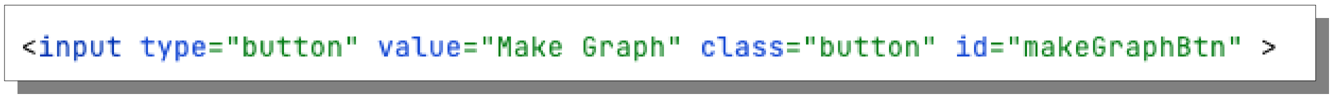
In keeping with our Web App Architecture, the Controller function, make_distribution_graph() simply retrieves the values of the HTML form's input tags in line 20 and extracts the form data as a dictionary. Line 21 creates a GraphGenerator object, passing the form dictionary, and line 22 calls upon GraphGenerator's method, which returns the serialized graph image.

Notice that make_distribution_graph() does not return a call to render_template() as does load_page(). *If we were to return a call to render_template(), then the whole page, main.html would be reloaded. This would be contrary to the requirement of inserting the generated graph into the page without reloading the whole page.*

So far we have examined the endpoint (/getDistributionGraph) in the Controller and we know that GraphGenerator.make_distribution_graph() returns a serialized graph image. So an important question is, how does the graph image get inserted into the main.html page without reloading the whole page? Let's follow the code.

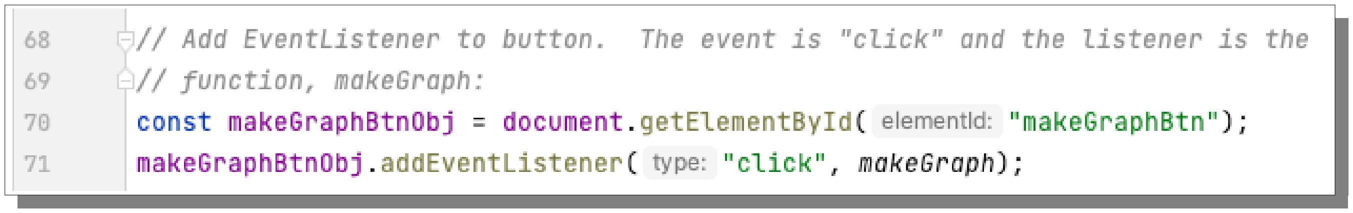
It is important to be aware that the generation and posting of an HTTP request, the receipt of the HTTP response, and the insertion of the returned graph image into the HTML page is all accomplished within the browser. That is, this is done with Javascript. So in the next couple of pages we will be looking at the browser side Javascript(mainFunctionality.js) and HTML(main.html).

The user action that starts the process is a click of the button labeled “Make Graph”. If we look at the HTML in **main.html**, we will find an input tag of type “button” with an id of “makeBraphBtn”:



```html
<input type="button" value="Make Graph" class="button" id="makeGraphBtn" >
```

All browser side actions are handled by Javascript. In order to give an action to the above button, we add a Javascript EventListener to the button object:



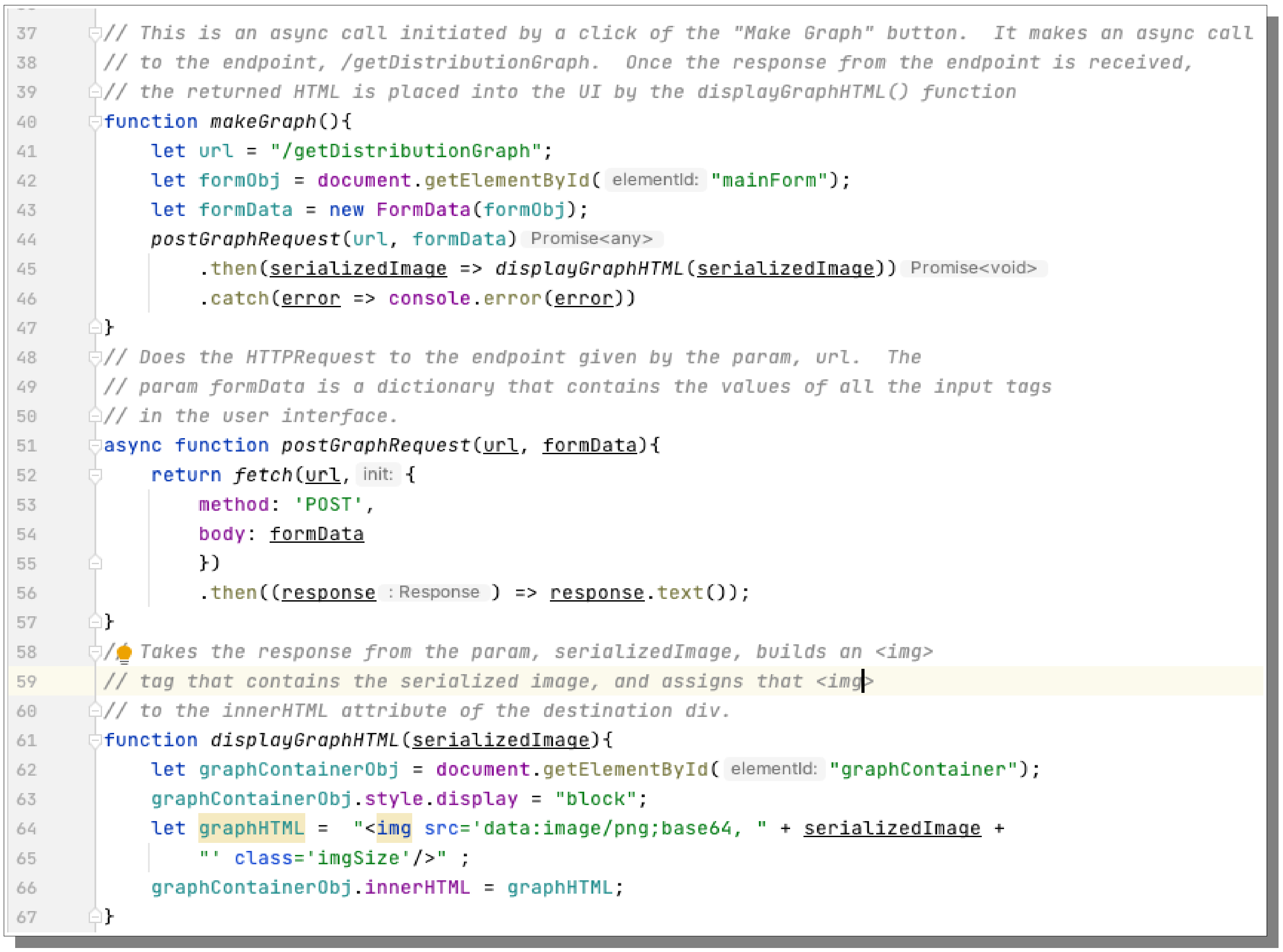
```javascript
68  // Add EventListener to button.  The event is "click" and the listener is the
69  // function, makeGraph:
70  const makeGraphBtnObj = document.getElementById( elementId: "makeGraphBtn");
71  makeGraphBtnObj.addEventListener( type: "click", makeGraph);
```

In line 70 we get the graph button as an object whose id is “makeGraphBtn”. In line 71 we add an EventListener to the button object. The event type is “click” and the listener is the Javascript function named makeGraph.

Before we look at the Javascript code to handle the graph image, let’s first just list all the jobs that need to be done by our Javascript code after the “Make Graph” button is clicked:

1. Create a Javascript **FormData** dictionary from all the values contained in the HTML form. (NOTE: a **FormData** object can be put into the body of a POST or GET request where it will be encoded and sent out with Content-Type: multipart/form-data.)
2. Specify the URL of the endpoint that will generate the graph image.
3. POST a request to the endpoint URL, while the body of the request contains the **FormData** object.
4. Accept the response of the POST. The response is simply the serialized graph image which is generated on the server side by GraphGenerator.generate_graph().
5. Get the **<div>** from **main.html** where the graph image will be inserted.
6. Build an **<img>** tag whose **src** parameter takes the serialized image. Then insert the response **<img>** tag into the **<div>** in main.html.

The code that accomplishes all 6 jobs is broken down into 3 functions for readability (NOTE these three functions can be combined into one large chain of calls that shortens the code, but makes it more difficult to read):



```
37  // This is an async call initiated by a click of the "Make Graph" button.  It makes an async call
38  // to the endpoint, /getDistributionGraph.  Once the response from the endpoint is received,
39  // the returned HTML is placed into the UI by the displayGraphHTML() function
40  function makeGraph(){
41      let url = "/getDistributionGraph";
42      let formObj = document.getElementById( elementId: "mainForm");
43      let formData = new FormData(formObj);
44      postGraphRequest(url, formData) Promise<any>
45          .then(serializedImage => displayGraphHTML(serializedImage)) Promise<void>
46          .catch(error => console.error(error))
47  }
48  // Does the HTTPRequest to the endpoint given by the param, url.  The
49  // param formData is a dictionary that contains the values of all the input tags
50  // in the user interface.
51  async function postGraphRequest(url, formData){
52      return fetch(url, init: {
53          method: 'POST',
54          body: formData
55          })
56          .then((response : Response ) => response.text());
57  }
58  // Takes the response from the param, serializedImage, builds an <img>
59  // tag that contains the serialized image, and assigns that <img>
60  // to the innerHTML attribute of the destination div.
61  function displayGraphHTML(serializedImage){
62      let graphContainerObj = document.getElementById( elementId: "graphContainer");
63      graphContainerObj.style.display = "block";
64      let graphHTML =  "<img src='data:image/png;base64, " + serializedImage +
65          "' class='imgSize'/>" ;
66      graphContainerObj.innerHTML = graphHTML;
67  }
```

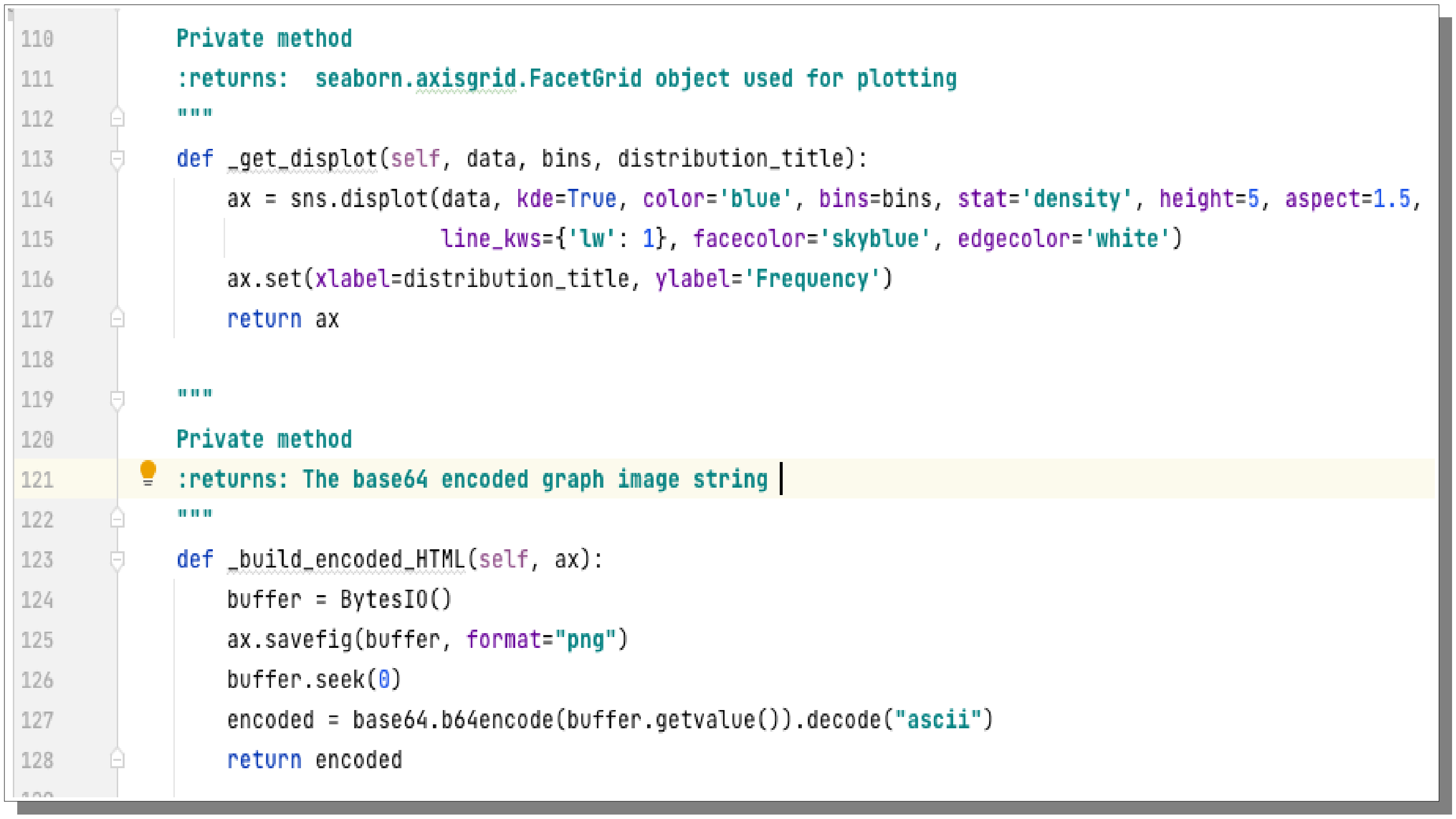
Recall that the function, makeGraph() was added as an EventListener to the “Make Graph“ button. The function makeGraph() sets the URL of the endpoint in line 41. In line 43 the input data from the **<form>** is put into a **FormData** object.

Line 44 calls the utility function, postGraphRequest(), which takes the endpoint URL together with the **FormData** object. Notice that in the body of the postGraphRequest() on line 52 we call the Javascript function, **fetch().** The **fetch()** function is given two parameters: (1) endpoint URL, and (2) a dictionary of post parameters. In our case we only need to specify the **method** (‘POST’) and **body** (**FormData**) parameters. The **fetch()** function returns a response where we then use its **text()** method to decode the response into text.

Then on line 45 we see that when f**etch()** returns a response, we call our custom function, displayGraphHTML() that first on line 64 builds an **<img>** tag that contains the serialized image and then in line 66 puts the HTML response into the **<div>** that has the id "graphContainer".

*Note that the Javascript **fetch()** is asynchronous. That is, it runs in its own thread and does not block while it is doing its job. If we had used some other non asynchronous way to retrieve the graph image, then we would have to re-render the whole page in order to insert the image into the page. This re-rendering blocks the page from user interaction. So imagine a web page with several resource intense tables or graphs and we want to display yet another graph or table. It would be a waste of resources and would limit functionality if we had to re-construct those other tables and graphs. By using an asynchronous call, the other graphs and tables stay in view and the user can still interact with the page no matter how long it takes for the new graph or table to be rendered.*

We have seen the full cycle of button press to image insertion. There is one more bit of technology that is necessary to make this all work. It is hidden in two private methods of the GraphGenerator class. The private methods are _get_displot() and _build_encoded_HTML(). The former makes an image of the plot of the data, and the latter serializes the image so that it can be streamed in an HTTP response. Let's look at the code:



```
    Private method
    :returns:  seaborn.axisgrid.FacetGrid object used for plotting
    """
    def _get_displot(self, data, bins, distribution_title):
        ax = sns.displot(data, kde=True, color='blue', bins=bins, stat='density', height=5, aspect=1.5,
                         line_kws={'lw': 1}, facecolor='skyblue', edgecolor='white')
        ax.set(xlabel=distribution_title, ylabel='Frequency')
        return ax

    """
    Private method
    :returns: The base64 encoded graph image string
    """
    def _build_encoded_HTML(self, ax):
        buffer = BytesIO()
        ax.savefig(buffer, format="png")
        buffer.seek(0)
        encoded = base64.b64encode(buffer.getvalue()).decode("ascii")
        return encoded
```

The method _get_displot() takes parameters specified in the User Interface and uses the Python package, **seaborn** (imported as sns) to do the plotting. In our example we use seaborn.displot() because it is based on matplotlib and also uses KDE (Kernel Density Estimation). The KDE part does the estimation of the probability density function.

In the method _build_encoded_HTML(), instead of saving the image to a file, we write the image to a buffer. Line 124 creates a **BytesIO** buffer and in the next line writes the image to that buffer. A this point, buffer contains binary data. So in line 127 we then use **base64.b64encode().decode("ascii")** to encode an ascii version of the buffer. More on base64 here. The method then returns the encoded graph image as a string.

NOTE: Why do we designate the methods, _get_displot() and build_encoded_HTML() as private? The jobs of graph creation and image serialization should not be exposed to a user of this class. We only provide one public method, make_history_graph() that calls the private methods and returns the serialized image.

This example shows the basic design of:

- how to collect user input (Javascript)
- how to use the input to generate the serialized graph image (Python class, GraphGenerator)
- How to insert and display a graph in the same UI asynchronously (Javascript)

Example: ML Test and Training Graphs

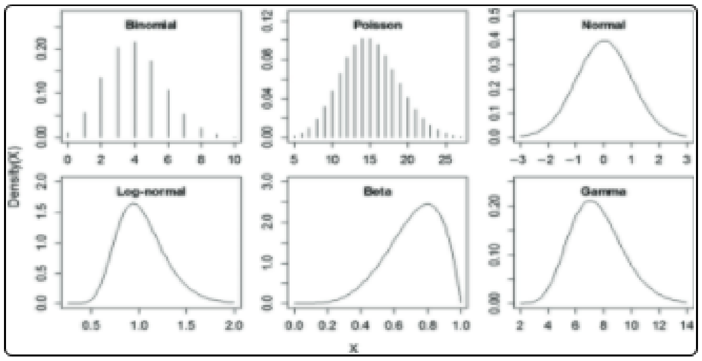
This example is found in the Git Repository with project name, **LearningRate**.

The design presented in the above example (Project Distribution)can be used to create a Data Science web application simply by:

1. **replacing the GraphGenerator with another class that builds an ML model and returns a serialized graph image.**
2. **Modify the input section of the UI to allow the user to enter relevant values.**

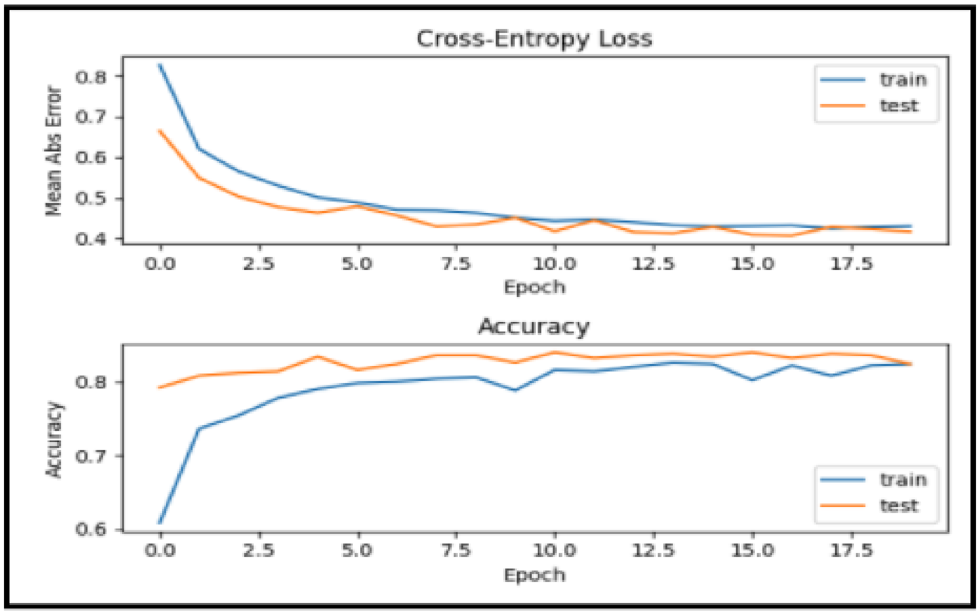
The UI will look familiar:

Learning Rate Demo



Sample Size	1000
Learning Rate	0.001
Number of Nodes	8
Number of Epochs	20

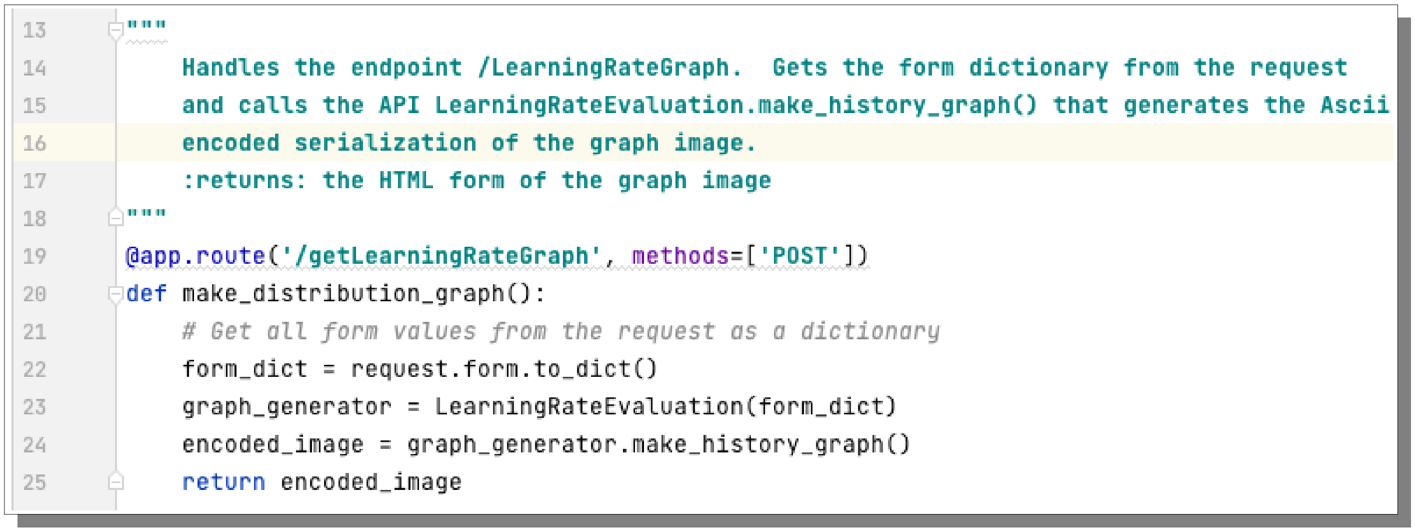
Make Graph



Not only does the UI look familiar, but this project uses the same main.html, app.py, main.css, and mainFunctionality.js with only slight modifications. We do completely replace the GraphGenerator class from the last example with a new one, LearningRateEvaluation.

The LearningRateEvaluation class has one public method: make_history_graph(). This method returns an Ascii encoded serialization of the two graph images.

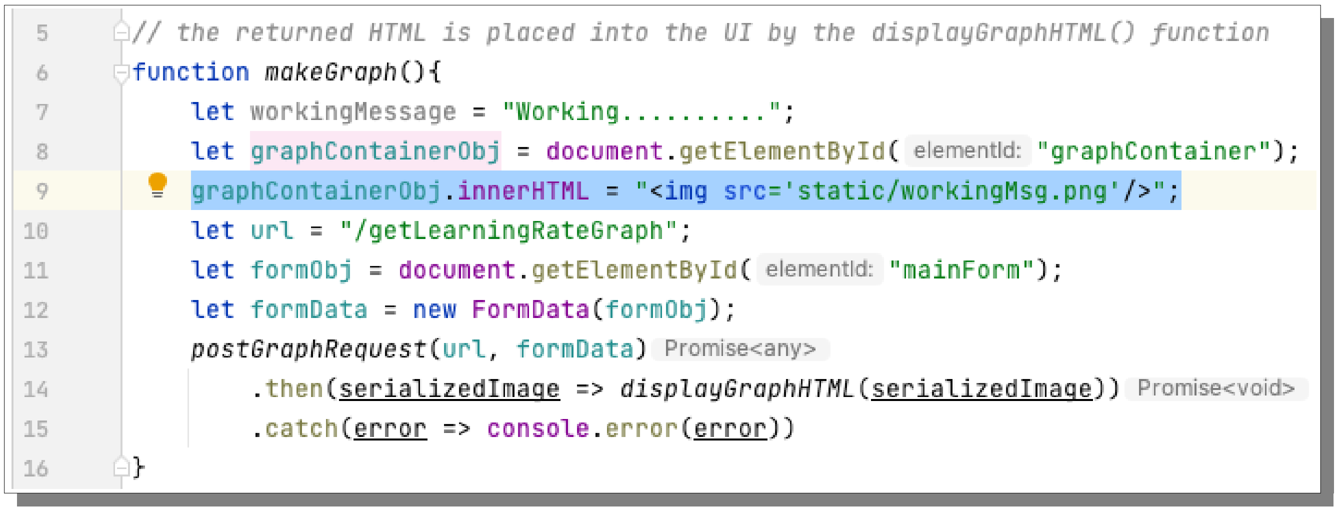
So in the main application, app.py, we replace the old route mapping with a new one called "/getLearningRateGraph".



```
"""
    Handles the endpoint /LearningRateGraph.  Gets the form dictionary from the request
    and calls the API LearningRateEvaluation.make_history_graph() that generates the Ascii
    encoded serialization of the graph image.
    :returns: the HTML form of the graph image
"""
@app.route('/getLearningRateGraph', methods=['POST'])
def make_distribution_graph():
    # Get all form values from the request as a dictionary
    form_dict = request.form.to_dict()
    graph_generator = LearningRateEvaluation(form_dict)
    encoded_image = graph_generator.make_history_graph()
    return encoded_image
```

Notice that this is the same code as in the last example, with only the URL, function name, and the new LearningRateEvaluation class.

The only other changes are in the Javascript makeGraph() function. We need to change the URL for the POST to the "/getLearningRateGraph" endpoint and we also added a display of the text: "Working…." that is displayed in the UI while the graph is being generated:



```
// the returned HTML is placed into the UI by the displayGraphHTML() function
function makeGraph(){
    let workingMessage = "Working..........";
    let graphContainerObj = document.getElementById( elementId: "graphContainer");
    graphContainerObj.innerHTML = "<img src='static/workingMsg.png'/>";
    let url = "/getLearningRateGraph";
    let formObj = document.getElementById( elementId: "mainForm");
    let formData = new FormData(formObj);
    postGraphRequest(url, formData) Promise<any>
        .then(serializedImage => displayGraphHTML(serializedImage)) Promise<void>
        .catch(error => console.error(error))
}
```

The rest of the Javascript code in mainFunctionality.js is unchanged. The reader can refer to the main.html file to see slight changes to the input tags in the UI.

This example can serve as a template for a more ambitious web app. We could add more parameters in the UI that would have values for batch size and number of layers . This would allow the user to tune his/her model so that loss and accuracy are optimized.

Conclusion

The point of this paper was to introduce the reader to the concept of putting a User Interface on top of a Data Science module, thereby making a complete web application. The author's experience is that while the real value of an application is in the back end calculations, the UI is what impresses most customers. It is no longer acceptable to deliver to a customer a Data Science computation in the form of a Jupyter Notebook, where there is no UI. The user should not have to modify code just to give parameters new values.

This paper demonstrated how a Data Science module can be constructed so that it has public API's and how this module can easily be joined to a UI in a full web app by using the proper Architecture. This demonstration implies that a Data Scientist can still write code in Notebooks, but an experienced web developer can take that Notebook code and put it into a form with public and private methods that can be used in a more presentable web application.

Finally, please familiarize yourself with the concepts shown in the first image of this paper: Web App Architecture. The most important concept that you can take away from this Architecture is that the App is built in modules that attempt to be as decoupled as possible. For example the code in the UI makes no reference to any of the other modules. All that appears is the URL of an endpoint.

For example, if our examples depended on a database, direct access to that database would be done in the blue colored module labeled Data Access. The database access classes would have public API's that could be called by other classes. More specifically, there would never be code in the UI that makes a query to a database even though, unfortunately, there is technology to do this.

Adherence to this Architecture saves software developers time in the long run when changes or additions are required. Changes and additions always bring the possibility of breaking the code. If we reduce dependencies, then broken code is more isolated and the cause is easier to find.

Appendix

Git Repository

All of the example projects in this document are located in a Git Repository. Currently the public repository is: https://github.com/adrezni/tutorials

Creating PyCharm New Project from Git Download

Each example stored in the Git Repository contains a file called **requirements.txt**. This file lists all the Python dependencies of each example project. This file will help you to set up a Virtual Environment for a project. A Virtual Environment will include all the Python packages that are specified in the **requirements.txt** file.

The strategy of creating a PyCharm project that includes dependencies is to

1. Create an empty project that will include a new empty Virtual Environment
2. Copy our project tree with code into this new empty project.

Here are the steps that need to be taken in PyCharm in order to create a new project for the first example called Minimal:

1. Create a new directory where you want to store all of your example projects. For example, create a directory called ~/PycharmProjects/examples/. The plan is to store all our example projects there.
2. In PyCharm, choose File | New Project. This launches the New Project window.
3. In the left hand column, select **Pure Python** as the project type.
4. In the text field labeled: **Location,** enter the path to the new project, including its name, Minimal: ~/PycharmProjects/examples/Minimal
5. Click on the radio button labeled: New environment using Virtualenv
6. For the location of the new environment, enter the same path as the new project: ~/PycharmProjects/examples/Minimal.
7. For the Base interpreter, use the path to your Python interpreter, in this case it is /usr/local/bin/python3.8
8. Click **Create** button

The above 8 steps will create a virtual environment for our new project, called Minimal. Now clone the project, Minimal from Git and copy all the directories and code to ~/PycharmProjects/examples.

Once the code and requirements.txt are copied into the new empty project, PyCharm will update the Minimal environment to include the dependencies specified in requirements.txt.

You can now run app.py to start the server and then open in a browser the URL: localhost:5001/Minimal

Creating new Project from Command Line

We can also create a Virtual Environment for our **Minimal** project by following the steps below. We assume that you have already created a directory that will hold all our example projects (~/PycharmProjects/examples), and that you have copied the cloned project, Minimal from Git into ~/PycharmProjects/examples/Minimal.   Assume that you have a PATH set to your Python interpreter.

Next, we create a Virtual Environment in our Minimal project. From the command line, first cd to our Minimal directory. *All remaining commands in this section assume that Minimal is the working directory.* So create the Virtual Environment for our Minimal project:

```
python -m venv .venv
```

This command will create a directory named **.venv** off the root, Minimal. The **.venv** directory will eventually contain all the dependencies that our project, Minimal needs to run.

There is a shell script in **.venv/bin** that we use to activate the environment:

```
source Minimal/bin/activate
```

By activating the environment, the shell's prompt will show what Virtual Environment we are in.

Next we want to complete the Virtual Environment by adding all the project's dependencies from **requirements.txt**. We do so with the command:

```
python -m pip install -r requirements.txt
```

This last command installs the dependencies listed in the txt file as well as their dependencies into the **.venv** directory.

While still in the Minimal directory, you can run the Flask server with the command:

```
python app.py
```

All the dependencies, including python3.8 are bundled into the **.venv** directory inside the project. This makes our project self-contained.

Note on Running the Web App Examples

When you run a Flask app from either and IDE environment or from the command line, you will see a message that states: *WARNING: This is a development server. Do not use it in a production deployment.* This statement gives you an indication that Flask is not a complete web server. It is neither stable nor secure. If you run a Flask web app from the command line, you will be subject to instabilities. In our examples, the first two projects, Minimal and MinimalWithTime are trivial enough that Flask can handle their requirements, However, the last two projects, Distribution and LearningRate make use of asynchronous calls to the web server, and Flask will have threading issues when the graphs are built.
So you have two choices in running the last two examples: (1) run from IDE (Professional Edition Recommended), or (2) deploy the application on a web server such as Apache that makes use of the Web Server Gateway Interface (WSGI).