

Criterion C: Product Development

Techniques Used

- a) Django Layout**
- b) Views**
- c) Models/Forms**
- d) Templates**
- e) Plotly graphing library(algorithmic)**
- f) Routing**
- g) Git / Heroku**
- h) CSS**

A. Django layout

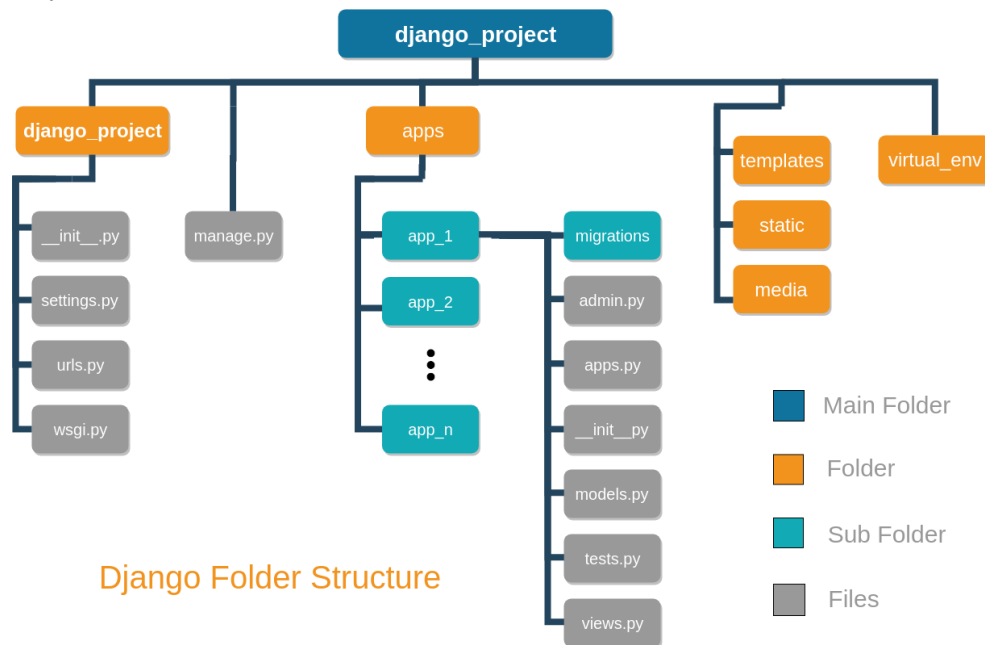


Figure 1. Sample layout of Django Project.

Samyak. “Django Project Structure Best Practice 2021.” *StudyGyaan*,

<https://studygyaan.com/django/best-practice-to-structure-django-project-directories-and-files>.

Accessed 10 March 2021.

I briefly covered the overall file structure in **Part B**.

This Django project has a database, startup files, and files for running. The base unit for the project is called an app; each app contains **models.py**, **view.py**, and **template files**.

A model is a data structure(Ex: a text field), and user added data is stored in the database in this structure. Users add data through the view, elaborated in **Section B**, which serves as a backend for the template, elaborated in **Section D**.

A specific url request leads to the view sending data to the template so it can display whatever the page needs.

A **forms.py** file allows the view a method to capture data.

My project follows this general premise. I have a high level “**entry**” data-type(what I refer to as an activity) that contains text and a relation to a user.

Each entry has multiple “**data**” attributes that contain a start time, end time, day, month, year, and description(what I refer to as a session). Each datapoint is connected to its entry through a many-to-one relationship; this is done through a “**foreign-key**” attribute.

B. Views

```
@login_required
def graph(request, entry_id):
    bob = ""
    graph_div = plotly.offline.plot(grapher(), auto_open=False, output_type="div")
    entry = Entry.objects.get(id=entry_id)
    data = entry.info_set.order_by('-date_added')
    day_list, month_list, year_list, da = [], [], [], []
    for c in data:
        if(c.entry.owner == request.user):
            day_list.append(c.day)
            month_list.append(c.month)
            year_list.append(c.year)
            da.append(c.date_added)
    year_graph_scatter = scatter_year_amount(year_list)
    scatter_ma = scatter_month_amount(month_list, day_list)
    scatter_mac = scatter_month_amount_color(month_list, day_list)

    context = {'eid':entry_id, 'graph':year_graph_scatter, 'graph2':scatter_ma, 'graph3':scatter_mac}
    return render(request, 'DA/graph.html', context)
```

Figure 1. A view function that receives parameters.

Figure 1 illustrates a sample view function. The request parameter is what is sent when the user demands the relevant webpage; the entry_id parameter is a value in the url(Ex: 1) that relates to a specific high level entry.

This view first sends relevant information from the database in the list “**data**” into different graph helper functions to create graph objects; these are stored in the “**context**” variable.

The render function finally sends the figure data to the ‘**DA/graph.html**’ template, which displays it.

```
@login_required
def delete_data(request, data_id):
    data = Info.objects.get(id=data_id)
    if request.method != 'POST':
        form = DeleteForm()
    else:
        #there is some data
        form = DeleteForm(data=request.POST)
        if form.is_valid():
            d = form.cleaned_data['del_choice']
            if(d == 'yes'):
                Info.objects.filter(id=data_id).delete()
            else:
                pass
            return HttpResponseRedirect(reverse('DA:entries'))
    context = {'data':data, 'form': form}
    return render(request, 'DA/delete_data.html', context)
```

Figure 2. Another view that receives data and then returns a render

Figure 2 is a simple view function that collects information from a form. It deletes a session before returning the user to the url 'DA:entries' through the 'HttpResponseRedirect' function.

Views in essence receive a request from a url, and potentially optional data, before updating the database if needed and then returning info from the database to the template which displays it for the user.

C. Models / Forms

```
class IntegerRangeField(models.IntegerField):
    def __init__(self, verbose_name=None, name=None, min_value=None, max_value=None, **kwargs):
        self.min_value, self.max_value = min_value, max_value
        models.IntegerField.__init__(self, verbose_name, name, **kwargs)
    def formfield(self, **kwargs):
        defaults = {'min_value':self.min_value, 'max_value':self.max_value}
        defaults.update(**kwargs)
        return super(IntegerRangeField, self).formfield(**defaults)

def default_start_time():
    now = datetime.now()
    start = now.replace(hour=22, minute=0, second=0, microsecond=0)
    return start if start > now else start + timedelta(days=1)

class Entry(models.Model):
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        return self.text

class Info(models.Model):
    entry = models.ForeignKey(Entry, on_delete=models.DO_NOTHING)
    text = models.TextField()
    date_added = models.DateTimeField(auto_now_add=True)
    start = models.TimeField(blank=True, default=default_start_time)
    end = models.TimeField(blank=True, default=default_start_time)
    day = IntegerRangeField(min_value=0, max_value=31)
    month = IntegerRangeField(min_value=0, max_value=12)
    year = models.IntegerField()

    class Meta:
        verbose_name_plural = 'data'

    def __str__(self):
        return self.text
```

Figure 1. Models used within the project.

Each model serves as a data type; each datum within the data type is restricted(Ex: model.TimeField() indicates that the **start** value must be a time value)

```

class InfoForm(forms.ModelForm):
    start = forms.TimeField(widget=forms.TimeInput(format='%H:%M'))
    end = forms.TimeField(widget=forms.TimeInput(format='%H:%M'))
    class Meta:
        model = Info
        fields = ['text', 'month', 'day', 'year', 'start', 'end']
        labels = {'text': 'Thoughts on the event', 'month': 'Enter month', 'day': 'Enter in day', 'year': 'Enter year', 'start': 'Starting time(in military time)', 'end': 'Ending time(in military time)'}
        widgets = {'text': forms.Textarea(attrs={'cols': 80})}

Day_Choices = [
    ('day', 'Day'),
    ('week', 'Week'),
    ('year', 'Year'),
    ('all', 'All'),
]

class DataForm(forms.Form):
    days_view = forms.CharField(label='How many days of data to view?', widget=forms.Select(choices=Day_Choices))

class EntryForm(forms.ModelForm):
    class Meta:
        model = Entry
        fields = ['text']
        labels = {'text': ''}

Delete_Choices = [
    ('yes', 'yes'),
    ('no', 'no')
]

class DeleteForm(forms.Form):
    del_choice = forms.ChoiceField(choices=Delete_Choices, widget=forms.RadioSelect)

```

Figure 2. A few forms within the project.

Each form has places to enter in data(which must conform to any limitations); Django handles the look on the template.

Add Data:

[League](#)

Thoughts on the event

Thoughts on the event

Enter month

Enter month

Enter in day

Enter in day

Enter year

Enter year

Start

Start

End

End

add entry

Figure 3. Form look on a loaded webpage to the user.

D. Templates

```
{% load bootstrap3 %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Learning Log</title>

    {% bootstrap_css %}
    {% bootstrap_javascript %}
  </head>
  <body>
    <!--Static navbar-->
    <nav class="navbar navbar-default navbar-static-top">
      <div class="container">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed"
            data-toggle="collapse" data-target="#navbar"
            aria-expanded="false" aria-controls="navbar">
          </button>
          <a class="navbar-brand" href="{% url 'DA:index' %}">Home</a>
        </div>

        <div id="navbar" class="navbar-collapse collapse">
          <ul class="nav navbar-nav">
            <li><a href="{% url 'DA:entries' %}">Topics</a></li>
          </ul>
          <ul class="nav navbar-nav">
            <li><a href="{% url 'DA:data' %}">All data</a></li>
          </ul>

          <ul class="nav navbar-nav navbar-right">
            {% if user.is_authenticated %}
              <li><a>Hello, {{ user.username }} </a></li>
              <li><a href="{% url 'Users:logout' %}">log out</a></li>
            {% else %}
              <li><a href="{% url 'Users:register' %}">register</a></li>
              <li><a href="{% url 'Users:login' %}">log in</a></li>
            {% endif %}
          </ul>
        </div><!--/.nav-collapse-->
      </div><!-- /container -->
    </nav>

    <div class="container">
      <div class="page-header">
        {% block header %}{% endblock header %}
      </div>
      <div>
        {% block content %}{% endblock content %}
      </div>
    </div><!-- /container -->
  </body>
</html>
```

Figure 1. The base.html template.

Figure 1 shows the base.html template. All templates inherit the div “**navbar-header**”, which contains links/elements that should be accessible throughout the website such as logging out. The

div “**page-header**” contains Django block elements to allow other templates to add unique elements.

```
{% extends "DA/base.html" %}

{% block header %}
<div style="display:flex;justify-content:space-between">
  <h1 style="display:inline-block;">{{ entry }}</h1>
  <h1 class="text-right" style="display:inline-block "><a href= "{% url 'DA:entries' %}">Back</a></h1>
</div>
<div style="display:flex;justify-content: space-between;">
  <h1 style="display:inline-block;font-size:medium"><a href="{% url 'DA:entry_data' entry.id %}">View Specific Data</a></h1>
  <h1 style="display:inline-block;font-size:medium"><a href="{% url 'DA:graph' entry.id %}">View Graphs</a></h1>
</div>
{% endblock header %}
{% block content %}

<p>Entries:</p>
<p><a href="{% url 'DA:new_data' entry.id %}">Add a new entry</a></p>

{% for d in data %}
  <div class="panel panel-default">
    <div class="panel-heading">
      <h3>
        <strong>{{ d.month }}</strong> - <strong>{{ d.day }}</strong> - <strong>{{ d.year }}</strong>
        <strong>|</strong>
        <strong>{{d.start}}</strong> - <strong>{{ d.end }}</strong>
        <strong>|</strong>
        <strong></strong>
        <strong>|</strong>
        <small>
          <a href="{% url 'DA:edit_data' d.id %}">Edit Entry</a>
        </small>
        <strong>|</strong>
        <small>
          <a href="{% url 'DA:delete_data' d.id %}">Delete Entry</a>
        </small>
      </h3>
    </div>
    <div class="panel-body">

```

Figure 2. A sample template that extends from the base.html template.

Django allows for values from the database to be included in the template through `{% %}` blocks. For example, one can conduct a for loop as shown in `{% for d in data %}`. Data is a list that is returned to the template in the context variable. So, for each element in data, certain aspects are shown(such as their month in `{{ d.month }}`).

Links are the last part of the template that I used which is not standard html. Links are still defined through the `<a>` tag’s href attribute. The basic template is `href="{% url 'APP:template_name template_variables %}'`. The `app:template_name` is merely a link to another template that is defined. In **Figure 2**, the example is `‘DA:entry_data’`. Additionally, if that view associated with that template requires another variable(ex: `entry_id`), then that is also sent to it.

The rest of the template just contains normal html.

E. Plotly Graphing

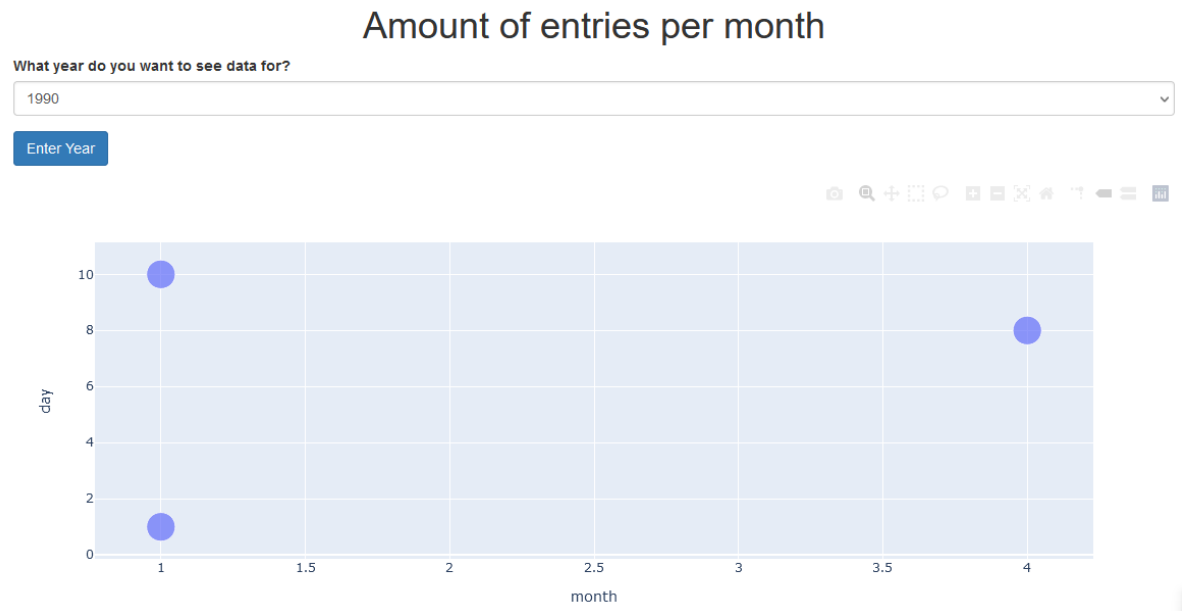


Figure 1. A graph available on the website.

This graph prompts the user for a specific year, then plots all sessions for an individual activity that occurred that year.

```
def grapher():
    fig = go.Figure(
        data = [go.Bar(y=[2,1,3])],
        layout_title_text="A figure displayed"
    )
    return fig

def scatter_year_amount(year_list):
    coc = []
    bob = {}
    for year in year_list:
        if year in bob:
            bob[year] += 1
        else:
            bob[year] = 1
    for year, amount in bob.items():
        coc.append([year, amount])
    df = pd.DataFrame(coc, columns=['year', 'amount'])
    print(df)
    fig = px.bar(df, x='year', y='amount')
```

Figure 2. Algorithmic data collection to create graphs.

Figure 2 illustrates the typical graphing methodology. I create a **pandas** dataframe object with a column containing the year, and the amount of sessions in that year. I use **plotly** to create a bar graph.

All graphs currently available follow this same process to generate meaningful representations of data.

F. Routing / File System

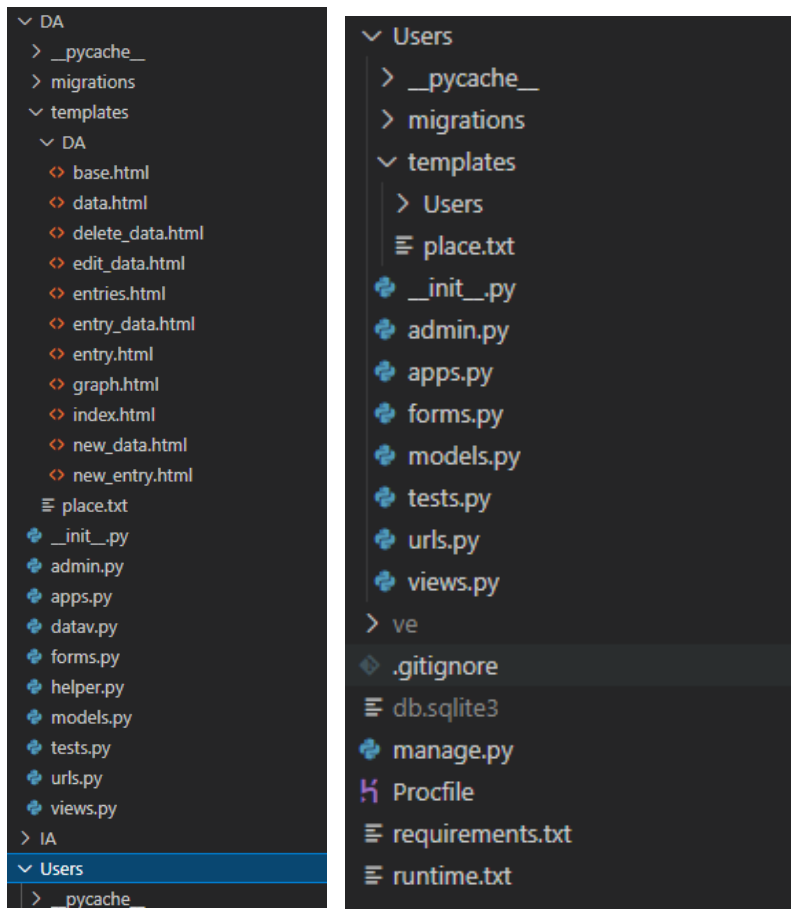


Figure 1 and 2. Show the file system for the project.

```

1  from django.urls import path
2
3  from . import views
4
5  app_name = 'DA'
6
7  urlpatterns=[
8      #Home Page
9      path('', views.index, name='index'),
10
11     #View all Entries
12     path('entries/', views.entries, name='entries'),
13
14     #New entry(data)
15     path('new_data/<int:entry_id>', views.add_data, name='new_data'),
16
17     #View data
18     path('data/', views.view_all_data, name='data'),
19
20     #Individual Entry
21     path('entries/<int:entry_id>', views.entry, name="entry"),
22
23     #New Topic
24     path('new_entry/', views.new_entry, name="new_entry"),
25
26     #Edit Entry
27     path('edit_data/<int:data_id>', views.edit_data, name='edit_data'),
28
29     #Delete data
30     path('delete_data/<int:data_id>', views.delete_data, name='delete_data'),
31
32     #View unique post data
33     path('entries/<int:entry_id>/data', views.view_some_data, name="entry_data"),
34
35     #View graphs??? One for now
36     path('entries/<int:entry_id>/graph', views.graph, name="graph"),
37 ]

```

Figure 3. Shows the routing for urls to pull up individual pages. Each path has the url that is sent to the program, the view program to call, and what the url goes by.

The main IA folder has a main url hub. There, I declared two namespaces so I could declare urls individually for the DA/User apps. The Users app handles user creation, logging in, logging out, etc.

The DA folder(standing for “data add”) handles all of the other functionality of the website.

Figure 3 shows how routing is done. A url is sent to the server(Ex: /DA/entries), and the server checks the initial value(DA) to see which app handles the request. Then, it checks the secondary attributes(/entries/) to see which view function needs to be called. Certain parameters such as <int:data_id> receive an integer value from the url(Ex: /DA/entries/1), and it sends that data_id value to the view as well.

G. Git / Heroku

Figure 2 in **section H** details the files needed to create a heroku project.

.gitignore leaves out files that heroku doesn't need to run the app

Runtime.txt and **Requirements.txt** detail the packages that heroku needs to create the app.

Settings.py, in the IA folder, as well as the **Procfile** then tell Heroku how to connect to the database/website as well as where to store data.

```
EMAIL_USE_TLS = True
EMAIL_USE_SSL = False

cwd = os.getcwd()
if cwd == '/app' or cwd[:4] == '/tmp':
    import dj_database_url
    DATABASES = {
        'default': dj_database_url.config(default='postgres://localhost')
    }

    SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

    ALLOWED_HOSTS = ['ia-time-noter.herokuapp.com']
    DEBUG = False

    BASE_DIR = os.path.dirname(os.path.abspath(__file__))
    STATIC_ROOT = 'staticfiles'
    STATICFILES_DIRS = (
        os.path.join(BASE_DIR, 'static'),
    )
```

Figure 1. Some of the settings in settings.py to run the app in general and on heroku

This file ensures heroku is running, the database is migrated to a postgres database appropriately, and more.

Git allows easy updating of the source files. Git is a “version control system”.

By creating a git repository for this project, it is simple to add changes and push them to heroku.

One can simply run “**git add .**” to add new files, “**git commit -am “message”**” to commit changes to heroku, and “**git push heroku master**” to push those changes to the heroku server.

There are many other uses for git, but these 3 commands help make changing the website very easy.


 <https://ia-time-noter.herokuapp.com>

Figure 2. Heroku url

Figure 2 simply displays that the web server can be connected to by a browser.

h) CSS

```
{% load bootstrap3 %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Learning Log</title>

    {% bootstrap_css %}
    {% bootstrap_javascript %}
  </head>
  <body>
    <!--Static navbar-->
    <nav class="navbar navbar-default navbar-static-top">
      <div class="container">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed"
            data-toggle="collapse" data-target="#navbar"
            aria-expanded="false" aria-controls="navbar">
          </button>
          <a class="navbar-brand" href="{% url 'DA:index' %}">Home</a>
        </div>

        <div id="navbar" class="navbar-collapse collapse">
          <ul class="nav navbar-nav">
            <li><a href="{% url 'DA:entries' %}">Topics</a></li>
          </ul>
          <ul class="nav navbar-nav">
            <li><a href="{% url 'DA:data' %}">All data</a></li>
          </ul>

          <ul class="nav navbar-nav navbar-right">
            {% if user.is_authenticated %}
              <li><a>Hello, {{ user.username }} </a></li>
              <li><a href="{% url 'Users:logout' %}">log out</a></li>
            {% else %}
              <li><a href="{% url 'Users:register' %}">register</a></li>
              <li><a href="{% url 'Users:login' %}">log in</a></li>
            {% endif %}
          </ul>
        </div><!--/.nav-collapse-->
      </div><!-- /container -->
    </nav>
    <div class="container">
      <div class="page-header">
        {% block header %}{% endblock header %}
      </div>
      <div>
        {% block content %}{% endblock content %}
      </div>
    </div><!-- /container -->
  </body>
</html>
```

Figure 1. A template that contains examples of the bootstrap css used.

Add Data:

League

Thoughts on the event

Thoughts on the event

Enter month

Enter month

Enter in day

Enter in day

Enter year

Enter year

Start

Start

End

End

add entry

Figure 2. What bootstrap looks like

Django-bootstrap essentially allows the user to create nice looking web pages with css by only defining class values. For example, in base.html, classes such as “**nav**” and “**container**” are used to import in styles already defined in bootstrap. This is illustrated in **figure 2**, where by only using a `{% bootstrap_form form %}` tag, the form displays itself nicely.

Word Count: 950 words