Overview

Time to begin your Backend solo project! For this and the next homework assignment, you will be iteratively building a Django Dashboard.

A *dashboard* is category of GUI interface that's intended to show a "big picture" or "at-a-glance" perspective on some topic. The term comes from dashboards on vehicles, which provide an "at-a-glance" perspective on the speed, fuel, milage, RPM, and other important information for the vehicle. Dashboards are often used in analytics software (to show popularity of a website or app), content management software (for a birds-eye view on what's to be published), business software (to monitor performance indicators of an organization), to name a few.

Dashboards

For this homework, you are to pick an API, data-set, or another dashboard topic, and build an interactive dashboard leveraging your newly developed Django and Python API skills. This page will portray (at least) 1) a table-based chart, and 2) a chart that graphically reflects information found from this real data-set or API. For this first homework, you won't be doing any database work just yet. Instead, you'll be primarily writing Python "view functions", HTML Django templates, along with any CSS tweaks that might be needed.

Example Solution

Click below for an example solution:

Kickstart Coding GitHub Dashboard - `https://kcbe-hw1-sol.herokuapp.com/`

Solo project topic

- The recommended topic is a **GitHub Stats Dashboard**

If you have an idea that you think might fit for the solo project, let the instructors know. Subject to instructor approval, we can work to tweak the requirements for the page to be about this idea, as opposed to a GitHub repos dashboard (this default project). Keep in mind you will likely be working on the same project for the next homework assignment as well.

1 Requirements

1.1 Hard requirements

- **Must be deployed and functioning to Heroku**
- Must use Django to serve up each page of content[1]
- Must use Django Template Language to template the pages[2]
- Must have at least 3 pages that link to each other

[1] Do not simply create a "static site"

[2] If you strongly prefer it, Jinja is permissible as well, as long as it's properly configured in `settings.py`

- Must have a Pipfile generated by `pipenv`
- Must do one of the following:
  - **Recommended:** Must use the `requests` package to access and display data from the free, public GitHub Repos API for your user account
  - *Alternative 1:* Must use the `requests` package to access and display data from another web API that's at least as complicated
  - *Alternative 2:* Must read in a large, static data-set from an CSV, XLS, JSON, or equivalent file-format
- Must have at least two pages as follows:
  1. One page must show tabular data, with at least 3 columns and rows.
  2. Another page must show some sort of related graph or charted data.[3]
- All code must be publicly available via GitHub, in a new repository[4]

1.2 Soft requirements

- **CSS Framework:** Use of a clean, modern, attractive CSS framework such as Bootstrap, Bulma, or Tailwind is highly encouraged but not required. Adding more CSS tweaks in addition is also recommended.[5]
- **Code cleanliness requirements apply:** Extremely messy repos might be docked in grade. Keeping your code clean and clear will look the most professional. Remove unused files, add in commenting, and utilize descriptive function and variable names.

Remember, these requirements are a minimum, not a maximum. If you have the time to spare, feel free to make your dashboard more complex than the minimum specified here, or even more complex than the example.

1.3 Submission

Submission is similar to homework from other courses with Kickstart Coding.

Include the following

- A link to the GitHub repo page. It should look something like:
  `https://github.com/YOUR-USERNAME/YOUR-REPO-NAME/`
- A link to the deployed, functioning site on Heroku. It should look something like:
  `https://YOUR-APP-NAME.herokuapp.com`
- *Optional* - Screenshots of it working locally (in case the Heroku site is down)

[3] The recommended graphing library is Pygal, since it's versatile and relatively easy to integrate into a Django project, but anything that results in an attractive graph is permitted.

[4] Create a new repository - don't just continue an old *Fundamentals* repo, start fresh with this one!

[5] The goal of this homework is to hone the Django skills necessary for a functional dashboard, not to spend a lot of time designing it.

2 Homework Steps

Unlike the *Fundamentals* homework, we do not provide clearly delineated phases.[6] If you are not sure where to begin, considering the following steps:

1. Choose a topic
2. Set up a new repo & Django project
3. Create 3 different view functions in `views.py` and accompanying templates that link to each other
4. Create a table page, with data populated from the API
    - Update one of the view-functions to retrieve the data from the API
    - Ensure that data from the API is put into the "context" of the template rendering
    - Update the HTML template to use a for-loop (e.g. `{% for x in y %}`) to show the data in an HTML table format
5. Create a chart page, with data populated from the API
    - Update another view-function to retrieve the data from the API
    - Now, use the Pygal documentation, and/or example snippets found in "Tips" below to create a Pygal chart
    - Ensure that the "rendered" version of the chart gets included in the "context" of the template rendering (see tips)
    - Update the HTML template to embed the chart (see tips)
6. Bonus: Work on any additional features that you want

Below are a big assortment of tips, including code snippets, to get you going with each of these steps.

3 Tips

Your goal is to get your software working and meeting the requirements. The clues and code snippets offered may need to be modified or tweaked for your use. Focus on understanding and accomplishing each goal one at a time.

3.1 Deciding on a dashboard topic

For your Solo Project for *Backend*, you must pick a topic. You have two options: Either go with the example one (for which solutions and examples will be provided), or pick some other topic. You will continue to build your next homework assignment based on this topic.

3.1.1 Option 1: GitHub Dashboard

This is the recommended option, and the easiest, since there is already something like it in the *Lesson 1.3* minidjango project.

Retrieve in JSON format by visiting `https://api.github.com/users/janeqhacker/repos` — more information is available at GitHub's official documentation. `docs.github.com`. The GitHub API is one of the easiest to use, and is free without even having to sign up.

3.1.2 Option 2: Custom Dashboard

[6] This is because you are given an example project to imitate, making the need for more precise steps less important.

> **Note:** Keep in mind that the *Frontend* course also has a data-related project with a similarly flexible topic. Instead of a dashboard, you will focus on building a single data visualization method from scratch using JavaScript — just something to keep in mind if you are intending on taking *Frontend* later and considering a custom topic now!

Alternatively, can work on a custom topic for your Dashboard that is not about GitHub. If you go down this route, ensure first you have access to the data of your choice, either via an API, or a static CSV or JSON. A valid topic should have **real, useful information** available to explore. In order to be compatible with the next homework assignment, pick a data-set with **the potential for users to customize the aspects of the data be displayed** (e.g. for GitHub repos, customizing which repos or stats are visible for a "custom dashboard" showing off the work of different people). In general, it should be **at least as complicated or realistic as the GitHub repos example topic**.

Some example of valid topics:

- Explore popularity of music artists and/or "chart topping hits"
- Explore sports team match-ups, and athlete performance data
- Comparing countries and demographics or statistics (population, Gini coefficient, health, etc) [7]
- Comparing information on cars, computers, clothes, video games, or other consumer products in a table, and use the chart to highlight other aspects

Remember: Make sure you have enough data on whatever topic you pick, since you'll want to make an interesting dashboard!

3.2 Starting the new Django project

Start a new Django project. You have three options:

1. Our introductory Django project: `minidjango` *(explored in Lesson 1.3)*
   - Pick this if you think this minimalist approach makes the most sense to you, or you want to get going before you get to later lessons.[8]
   - To get going: Use the project with the Heroku Getting Started guide, or copy either the `minidjango` directory from the Lesson 1.3, or one of the other activities
   - `github.com/kickstartcoding/heroku-getting-started`.
2. Django default: `django-admin startproject` *(explored in Lesson 2.1)*
   - This is the default way to start a Django project.
   - To begin, follow the procedure we cover to start new projects in the first activities of Lesson 2.1.
3. Our full-featured, custom project: `django-kcproject-starter` *(explored in Lesson 2.3)*
   - This is a more "heavy-weight", "production-ready" approach, including an extendable user system, Postgres DB support, login / sign-up pages, and more
   - Learn how to start here: `github.com/kickstartcoding/django-kcproject-starter`.

[7] Example: A grim, but interesting, and recently extremely common side-project among web developers are dashboards that demonstrate the on-going pandemic's impact on countries or regions.

[8] Note: While simple to get going, it does present a downside in that it does not support databases as-is, meaning you'll need to convert your work into one of the other options below for the next iteration

3.3 GitHub: Creating a new repo

Since you'll likely be starting your project with existing code, it might be more convenient to take a different series of steps than you may be used to from previous courses to get your local directory "git-enabled" and linked with GitHub:

1. Create a new repo on GitHub, and *BE SURE TO NOT* check "initialize with README.md" or .gitignore or anything. You want to make a *truly blank repo* for these steps to work without a hiccup. Just fill in the name and description, and nothing else.
2. Now, initialize and push the existing code we already have, by running this in the command-line:

```
ls # Ensure you are "next to" the manage.py, etc
git init # Initalize this
git add -A # Add all your files
git commit -m "first" # Make a commit
```

4. Finally, link up your local repository with the GitHub one, and push. Follow the instructions provided on the github.com page. They will look a bit like this, except instead of `janeqhacker/mycooldash` it will be whatever your username is and your repo name is:

```
git remote add origin git@github.com:janeqhacker/mycooldash.git
git push -u origin master
```

3.4 Using Django template inheritance

Using template inheritence is industry best-practices for Django for sharing code across different templates. Using template inheritance allows us to have a "base" template, that other HTML code "inherits" from.[9] Check out the Django Girls tutorial on extending: `tutorial.djangogirls.org/en/template_extending/`

3.4.1 Clue

Examine the following templating example code.

base.html The "base" template.

```
<!DOCTYPE HTML>
<html>
<head> <title>{% block title %}{% endblock %}</title></head>
<body>

<!-- TODO: navigation goes here... -->
<a href="all-repos/">All repos</a>

<div class="container">
    {% block content %}
    {% endblock content %}
</div>
<footer>
    (C) 2020 Jane Q. Hacker (janeq@hacker.com)
</footer>
</body>
</html>
```

[9] This is a similar concept to Object Oriented Programming's inheritance, and why it this terminology is used.

allrepos.html

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

{% block title %}
All repos
{% endblock title %}

{% block content %}
<h1>All repos</h1>
<table> <!-- TODO: add django for loop here... --> </table>
{% endblock content %}
```

views.py The view itself, now will just render the "final" (child) HTML template, and the `extends` keyword will bring in the rest (the `base.html` template).

```
def allrepos(request):
    context = {
        # TODO: Need to include data from API here
    }
    return render(request, "allrepos.html", context)
```

3.5 Dashboard page 1: Data table

- **Model solution:** `https://kcbe-hw1-sol.herokuapp.com/all/`

To create the data table page, there are three primary steps:

1. Use `requests` to get the data from the API[10]
2. Include the relevant data in the context
3. Use Django Template for-loops to show the data as a table.

Clues for each step:

3.5.1 Clue 1: Using requests to get relevant data

```
response = requests.get("https://api.github.com/users/kickstartcoding/repos")
```

Note: Remember to install `requests` before importing it.

3.5.2 Clue 2: Including data in the context

```
repo_list = response.json()
context = {
    "github_repos": repo_list,
}
```

[10] If you are using a CSV or JSON file as the data source, then you'd need to read the file instead.

3.5.3 Clue 3: Showing data in the template

```
<table>
    <thead> <tr><th>Name</th> <th>etc...</th></tr> </thead>
    <tbody>
        {% for repo_dict in github_repos %}
            <tr>
                <td>{{ repo_dict.name }}</td>
                <td>...other stuff goes here...</td>
            </tr>
        {% endfor %}
    </tbody>
</table>
```

3.6 Dashboard page 2: Data chart

> Background: *Pygal*[a] is a free software charting library that turns data into SVG-powered charts. It is recommended for this assignment since it's comparatively quick to get going and fairly powerful. *SVG* is the most popular vector image format.[b] Pygal uses SVG to draw its complex and interactive graphs.
>
> [a]Note: You are free to use any equivalent charting library instead of Pygal that can fulfill a similar function, however the clues offered in this document will only be for Pygal.
>
> [b]Vector graphics differ from formats like JPG or PNG, in that they store shapes and colors as opposed to using pixels, and are popular for illustrations, graphs, and many other situations in graphic design. In both form and function, *SVG* closely resembles HTML and CSS, and is supported in all modern browsers to be embedded within HTML documents.

- **Model solution:** `http://kcbe-hw1-sol.herokuapp.com/by-size/`

To create the data chart page, there are three primary steps:

1. Use `requests` to get the data from the API
2. Use a for-loop or equivalent in the Python view to build a Pygal chart
3. Include the resulting SVG image in the template

Clues for each step:

3.6.1 Clue 1: Using requests to get relevant data

*(See previous **Clue 1**)*

3.6.2 Clue 2: Building a Pygal chart

- Documentation: `pygal.org/en/stable/documentation/types/index.html`

There are many, *many* different charts you can build with Pygal, and they all are okay for the homework assignment requirements, so browse through the options and pick whichever one you like and think is most appropriate for the dashboard you are creating!

```
chart = pygal.Pie()
for repo_dict in repo_list:
    value = 42 # TODO: Replace this...
    label = repo_dict["name"]
    chart.add(label, value)
```

3.6.3 Clue 3: Including chart on page

- Documentation: pygal.org/en/stable/documentation/web.html

There are three ways to embed the SVG from Pygal:

Method 1: Including the SVG code directly Include the SVG code directly in the template using a "safe" filter:[11]

```
# In our views.py...
chart_svg = chart.render()
context = {
    "rendered_chart_svg": chart_svg,
}

<!-- In our HTML template... -->
{{ rendered_chart_svg|safe }}
```

Method 2: Including the SVG code via an HTML embed tag This is a slightly "weirder" way to include it, but it is preferred because it enables the interactive capabilities (hover, clicking) that Pygal supports, as opposed to just being a "static image":

```
# In our views.py...
chart_svg_as_datauri = chart.render_data_uri()
context = {
    "rendered_chart_svg_as_datauri": chart_svg_as_datauri,
}

<!-- In our HTML template... -->
<embed type="image/svg+xml" src="{{ rendered_chart_svg_as_datauri|safe }}" />
```

Method 3: (Hardest) Create new route just for the chart Since SVGs are images, you can serve them from a separate view, which has only 1 job: Rendering the chart. Pygal even offers a `render_as_django_response` method to make this easier — instead of returning a rendered HTML template, you can just directly return the rendered chart. There is little reason to go down this route for this homework assignment, as it requires quite a bit more code and gives no advantages over the others. More information: `http://www.pygal.org/en/stable/documentation/output.html`

3.7 Optional: GitHub API improvements

3.7.1 Avoiding rate-limiting via authentication

This is a common "gotcha" with the GitHub REST API.

The GitHub REST API is "rate-limited". The anonymous API will only work 60 times per hour, unless you use their (free) authentication API. This is a little trickier to get set-up, so it's recommended you focus on getting things working with the anonymous API first.

More information on GitHub authentication:

- `https://docs.github.com/en/free-pro-team@latest/rest/overview/resources-in-the-rest-api`
- `https://docs.github.com/en/free-pro-team@latest/rest/overview/other-authentication-methods`

[11] The safe is needed to prevent the code from showing up due to HTML characters being escaped.

3.7.2 Using PyGitHub

Instead of using requests, you may prefer to use PyGitHub, a package specifically for interacting with GitHub from within Python. While it uses the same REST API under the hood, it does smooth over some of the aspects of authentication, so it might be easier than requests if you are attempting to use the authenticated version of the API mentioned above.

Read about PyGitHub here:

- `https://github.com/PyGithub/PyGithub`

3.8 Launching on Heroku

Dynamic sites can't be launched on GitHub pages. You must use a host that supports running arbitrary HTTP server processes, such as Heroku.

- If you used minidjango starter, use the Heroku Getting Started guide for steps on launching your page to Heroku. It's found at `github.com/kickstartcoding/heroku-getting-started`.
- If you used the default Django configuration, then use this version of the Heroku Getting Started guide: `djangoadmin_startproject.md (github.com/kickstartcoding/heroku-getting-started)`.
- If you used the `django-kcproject-starter`, follow the guide that came with it in the README.md

3.8.1 Optional

Consider logging into the Heroku online interface and changing the app name from the randomly-generated nonsense names to be something more appropriate, such as your name or website name.

4 Bonus features

Have more time? Here are a list of extra features to consider:

- Allow other repos to be looked up based on a form:
  - See the "Languages used" page: `https://kcbe-hw1-sol.herokuapp.com/by-language/`
- Show a logo for computer languages:
  - See this project: `https://github.com/abranhe/programming-languages-logos`
  - See the "All" page in the example: `https://kcbe-hw1-sol.herokuapp.com/all/`
- (Currently not in example) - Sortable table. Click on a table header to cause it to "sort" by that header.

If you are using an alternative data set that has to do with countries, consider using the Country Flags "API" for getting flag logos: `https://www.countryflags.io/`