Overview

Your previous homework iteration employed Django to create a dynamic dashboard. It notably lacked, however, any use of the database. This prevented customizing content on the site, without having to redeploy the site, making it inflexible and limited in features.

In this iteration, you will add database support using Django's ORM, and then launch the new version to Heroku. This requires starting the project using a "more standard" repo set-up than previous ones.

Example Solution

Click below for an example solution:[1]

- GitHub Dashboard - Intermediate (Part 1) `https://kcbe-hw2-sol-p1.herokuapp.com/`
  - This is the demonstration of having completed most of the requirements
- GitHub Dashboard - Complete (Part 2) `https://kcbe-hw2-sol-p2.herokuapp.com/`
  - All requirements completed, including all 3 of the "pick 1 of 3" extra requirements

1 Requirements

1.1 Hard requirements

- **Must be deployed and functioning to Heroku**
- Must have at least 1 Django ORM model in-use, allowing customization of the dashboard[2]
  - Must support use of the admin interface to modify this content
  - Must use the Django ORM, not SQL[3]
- The public site launched on Heroku must be have some example data demonstrating its functionality[4]
- Must have at least 1 URL parameter in use, the captured value must then be used to select and display a model from the database[5]
- *Pick 1 of 3:*[6]
  1. Move data or graph manipulation into a method (function) on your model, and you must also utilize Django caching in order to permit more data to be shown at once
  2. Log-in / sign-up, coupled with user-facing CRUD operations for customizing a per-user individual dashboard, associated with the user with One-to-Many relation

[1] Part 1 and Part 2 have separate example solutions, so you can see how the app "was gradually built up" at each stage.

[2] The suggested aspect of customization: `DashboardPanels`

[3] Note that it is not necessary or even typically that useful to use any SQL code for this HW assignment. That said, this should be "backed" by Postgres when launched to Heroku, but it's fine (and encouraged) to use SQLite locally during development.

[4] This is so that the person grading your homework can see how it works without having to create new model instance, or have admin access, etc. Don't have to be fancy, just use the admin interface to make (for example) 3 example `DashboardPanels`.

[5] That is to say, you must have a URL / view-function that supports viewing particular items from the database, linked from elsewhere.

[6] *At least* one of these requirements must be met. It's okay—and recommended, if you have the time—to attempt all 3.

3. A second model[7] that allows for collections of Dashboard Panels specified by either a One-to-Many or Many-to-Many relation

- Previous requirements apply:
  - Must have at least two pages, showing tabular and/or graph data
  - Must use templating for the pages
  - Must have a Pipfile generated by `pipenv`
  - Must use Django to serve up content

1.2 Soft requirements

- Use of Postgres is preferred, although use of SQLite is acceptable[8]
- Previous soft requirements apply:
  - **CSS Framework:** Use of a clean, attractive CSS framework is highly encouraged, but not required.
  - **Code cleanliness requirements apply:** 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 even more complex than this!

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)

2 Homework Steps

There is no one "required" sequence of steps. Feel free to work on features in any order.[9] However, if you are not sure where to begin, consider the following steps:

1. Part 1: Create a `DashboardPanel` model

[7] For example, "Dashboard" or "DashConfig" – the idea being a saved dashboard configuration of dashboard panels

[8] If you use Sqlite, keep in mind that every time you launch it will be "reset" to whatever DB file you have in your git repo. This is not ideal if you want a typical blog use-case where you can add articles online and see them on your page.

[9] There is also no need to do Part 1 and Part 2 separately, and certainly no need to submit them separately. However, if you don't have much time, consider focusing only on completing and launching the "Part 1" requirements, since they are worth the most points when grading, to at least get partial credit.

1. Create the new model & create migrations
2. Create test data via admin panel
3. Modify views and templates to loop through dashboard panels to show on homepage
4. Add or modify views to create a "details" page for each dashboard panel (optional, but likely)

2. Part 2: *Pick 1 of 3*
   - Move your graphing and API logic to a method in `DashboardPanel`, and add caching to allow many panels on one page without hitting API limits
   - Make or configure log-in / sign-up pages, and create user-facing (e.g. non-admin panel) CRUD operations for `DashboardPanel` model
   - Create a new `Dashboard` model (related to the `DashboardPanel`), and add accompanying views
3. Launch to Heroku
4. 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.

2.1 Part 1 Tips: Adding Database Support

Background: The main requirement of this homework is to allow database-driven customization of the dashboard style and/or content.

The requirement is that at least *something* on your site is from the database. The recommended model is `DashboardPanel`. This represents a single customizable aspect of the data being displayed, such as a chart or table of certain data that is pertinent to be compared.

2.1.1 Clue 1: Steps to create and use a model

1. Write the model code in `models.py`. Make sure you do all your work in a Django app that was properly added to your `settings.py`. For example:

```
class DashboardPanel(models.Model):
    # For the GitHub example:
    github_username = models.CharField(max_length=127)
    repo_name = models.CharField(max_length=127)

    # Customizable aspects of panel chart
    panel_type = models.CharField(max_length=127, choices=[
       ("piechart", "Pie-chart of languages used"),
       ("barchart", "Bar-chart of languages used"),
    ])

    # TODO: More fields might go here...
    # (Consider adding other customizable aspects, including style or theme,
    # different chart options, text descriptions.)
```

2. Register it so that it shows up in the `admin.py`. For example:

```
import admin
from .models import DashboardPanel
admin.site.register(DashboardPanel)
```

3. Run necessary shell commands to be able to start adding instances in the admin interface:

```
python manage.py makemigrations
python manage.py migrate
```

4. Check that your new model appears in the admin interface and you can edit and update it.

Note: If `makemigrations` doesn't pick up on your changes, make sure 1) you've added it to your `settings.py`, 2) also try specifying your app name explicitly, like `python manage.py makemigrations app_name_goes_here`.

2.1.2 Clue 2: Listing instances in a view

1. Use it in a view. For example:

```
from .models import DashboardPanel

def view_panels(request):
    dboard_panels = DashboardPanel.objects.all()
    context = {
        "all_panels": dboard_panels,
    }
    return render(request, "home_panels.html", context)
```

2. Reference it in a template. For example:

```
{% for panel in dboard_panels %}
    <div class="shadow border"> <!-- Example HTML / CSS -->
        <p>{{ panel.title }}</p>
        <a href="/details/{{ panel.id }}/">Click for more</a>
    </div>
{% endfor %}
```

2.1.3 Clue 3: Making a view for viewing a single instance

1. Create a view that shows only a single DashboardPanel based on ID. For example:[10]

```
def panel_details(request, panel_id):
    panel = DashboardPanel.objects.get(id=panel_id)

    chart = pygal.Pie()
    # TODO: Make aspects of the chart (such as Pie vs Bar, styling, etc)
    # customizable based on the data in the panel model

    # TODO: Get data from API, file, DB, or somewhere else, possibly based on
    # the panel model
    for repo_dict in repo_list:
        value = 42 # TODO: Replace this...
        label = repo_dict["name"]
        chart.add(label, value)

    context = {
```

[10] Note that if you choose to move this logic to a method (see "Part 2: Creating a method") you will be rewriting this again, so you might choose to work on that refactor first.

```
        "panel": panel,
        "rendered_chart_svg": chart.render(),
    }
    return render(request, "panel_details.html", context)
```

2. Ensure your DashboardPanel has a path in `urls.py` that uses the id. For example:

```
urlpatterns = [
    path("/details/<panel_id>/", views.panel_details),
    # ...
]
```

3. Reference it in a template. For example:

```
<h1>{{ panel.title }}</h1>
<p>{{ panel.description }}</p>
{{ rendered_chart_svg|safe }}
```

2.2 Part 2 Tips: Creating a *method* and using Django caching

Background: One of the "pick 1 of 3" requirements is to move API and/or graphing code to a method on your Model, and then utilize Django caching to reduce slow or rate-limited API calls.

This feature represents partially a refactor, and partially new logic to make things more efficient. We'll start by moving some data or API related code from the `views.py` to `models.py`.[11]

2.2.1 Clue 1: Adding a method to a model

Functions can be defined within models, and then are called *methods*.[12] The value they return can be used in templates. This can be used to better organize code between `views.py` and `models.py`, and also in our case make it easier to apply "caching".

See below for an example of how you might add a new method to the `DashboardPanel` model:

```
class DashboardPanel(models.Model):
    # ...snip...  (all the "model fields" go here)
    def get_api_and_render_chart(self):
        response = requests.get("https://api.github.com/users/") # etc
        repo_list = response.json()
        # TODO: Make aspects of the chart (such as Pie vs Bar, styling, etc)
        # customizable based on the data in the panel model
        chart = pygal.Pie()
        # ...snip... (for loops etc building the Pygal chart here...)
        rendered = chart.render()
        return rendered


<!-- In our template, to then show... -->
{{ panel.get_api_and_render_chart|safe }}
```

[11] In the real world, it could logically be put in either the model or the view and still correctly follow the MVC/MVT paradigm, whichever is the most convenient. At least in the case of the example solution, it's more convenient to put this code in `models.py` than in `views.py`

[12] "Method" can be defined simply as a function defined within a class, or a function bound to an object instance. This is OOP terminology.

2.2.2 Clue 2: Showing everything at once

One advantage of putting the data related logic in the model is that it can make it easier to show many panel graphs at once in the template. For example, wherever you are looping through your panels, you might do something like:

```
{% for panel in dboard_panels %}
    <div class="shadow border"> <!-- Example HTML / CSS -->
        <p>{{ panel.title }}</p>
        {{ panel.get_api_and_render_chart|safe }}
    </div>
{% endfor %}
```

Warning: This can very quickly use up your API limit, since it will do an API call for EVERY panel each time you visit the page! This is why this requires another piece of the puzzle to prevent re-computing the graph and API each time someone visits your page: Caching.

2.2.3 Clue 3: Adding in Django caching

Background: Django has a "caching" system that is useful to prevent slower or wasteful operations (such as API calls) from taking too much time by saving previous results into a *cache*. Under the hood, the cache is saved in a file, in the DB, in a dictionary, or a variety of other special, more advanced options. Caches are set to expire in a time in seconds, at which point they are said to "go stale", and will stop re-using the old results or HTML and re-generate the results.

There are a three built-in approaches for caching. Pick one:

1. Template fragment caching
    - `https://docs.djangoproject.com/en/3.1/topics/cache/#template-fragment-caching`
    - Any part of our template HTML can be cached based on an expiration date and a "key" (typically a database ID)
    - One of the easiest ways to add caching to your page
2. Low-level cache usage
    - `https://docs.djangoproject.com/en/3.1/topics/cache/#the-low-level-cache-api`
    - This requires writing an if-statement structure in your views or model methods, making it slightly harder[13]
3. Page cache[14]

2.2.4 Example code snippet

The easiest way to add caching is with the `{% cache %}` template tag (example below). This will "save" the rendered HTML the first time the page is visited, and re-use the old version until the expiration time specified.

[13] Slightly harder in our case, requiring more custom code. The pseudocode might look like: "If it's in the cache, return the cache value. Otherwise do an API call, construct the new chart, and return that."

[14] You can also very easily cache an entire pages or view, with the "cache decorator". This is probably less useful for our case, but is very useful to speed up the homepage of websites.

```
<!-- Caching around the method call to prevent extra calls -->
{% load cache %}

<!-- 86400 is expiry in seconds, panel_cache is the "cache name", and panel.id
is the unique "key" -->
{% cache 86400 panel_cache panel.id %}
<div class="blah blah blah">
    <p>Other stuff might go here...</p>
    {{ panel.get_api_and_render_chart|safe }}
</div>
{% endcache %}
```

2.3 Part 2 Tips: Log-in, sign-up, and per-User custom dashboard

Background: One of the "pick 1 of 3" requirements is to provide a log-in and sign-up system, allowing users of your site to create their own custom dashboards.

2.3.1 Clue 1: Adding log-in and sign-up

If you have started this project using the Django KC Project Starter, then this has already been done for you! You already have a log-in and sign-up system.

If you have not started your project this way, and you really want to add this feature, you might either want to try switching to this, or to look closely at the log-in and sign-up system that it implements for code clues:

`https://github.com/kickstartcoding/django-kcproject-starter/tree/master/apps/accounts`

2.3.2 Clue 2: Creating user-specific dashboards

Background: To allow users to create "private" dashboards for each user, you will have to 1) add a ForeignKey to relate the `DashboardPanel` (or equivalent) model to the user that created it, and 2) create user-facing CRUD views so that the users can customize their dashboards after logging in (as opposed to using the admin panel).

1. Modify your model to specify a "creator" `ForeignKey`, to create the "One to Many" relationship ("one user has many dashboard panels"):

```
from apps.accounts.models import User
# Alternatively: from django.contrib.auth.models import User

class DashboardPanel(models.Model):
    # ...snip...  (all the other "model fields" go here)
    creator = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
    )
```

2. Add "R" and "C" CRUD URL paths, view functions, and templates to permit users to READ (e.g. show all panels on their "user page") and CREATE Dashboard Panels. See activities for more examples of this (such as in `3.1-CRUD`). Incomplete code snippet example of the "R" in CRUD below:

```
# urls.py
urls = [
    path('/user-dashboards/<the_username>/', views.view_user_dash),
]

# views.py
def view_user_dash(request, the_username):
    # R in CRUD
    user_object = User.objects.filter(username=the_username)
    panels = DashboardPanel.objects.filter(user=user_object)
    context = {
        'panels': panels,
    }
    return # ...snip...
```

3. Add a "DELETE" route and view function
4. Optional: If you have the time, round it out with an UPDATE route[15]

2.4 Part 2 Tips: Adding a second model

Background: One of the "pick 1 of 3" requirements is to have another model representing an entire "set" of dashboard panels configured to provide a single dashboard.

Suggested names: `Dashboard`, `DashConfig`, or `PanelCollection`

Adding another model is a lot like adding the first. To fulfill the requirement, this model has to do something useful for your dashboard project, and must include either a ManyToMany or a ManyToOne (e.g. ForeignKey) relationship with another model (e.g. your `DashboardPanel` model). An example code snippet:

```
class DashboardPanel(models.Model):
    # ...snip...  (all the "model fields" go here)

class PanelCollection(models.Model):
    title = models.CharField(max_length=127)
    description = models.TextField()
    panels = models.ManyToMany(DashboardPanel)
```

In order to receive full credit, you must also create a new view-function that permits viewing this new model, along with its associated `DashboardPanel` objects. This view likely will *replace* your `panel_details` view (from 2.1.2 on page 4).

2.5 Tip: 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 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

[15] Not 100% necessary, since users can DELETE & then CREATE a new one for the same effect.

3 Bonus: Suggested extra features

Have more time, and want to make your dashboard even cooler while getting practice with other Django features? Consider some of these challenges:

3.1 Create a custom "templatetag" or template "filter"

- Django supports creating custom extensions to the Django templating language that do stuff unique to your project.
- You might consider partially replacing the model method (mentioned above) with a custom "template-tag"
- This would be your own template tag that renders given data as a Pygal chart.
- `https://docs.djangoproject.com/en/3.1/howto/custom-template-tags/`

3.2 Try out the new Django 3 TypeChoices instead of a list

Modern versions of Django introduced an alternative syntax for model fields with limited choices.

```
# This 'models.TextChoices' allows only certain options to be used by
# panel_type
class PanelTypes(models.TextChoices):
    LANG_PIE = 'lang_pie', 'Pie-chart of languages used'
    LANG_BAR = 'lang_bar', 'Bar-chart of languages used'

class DashboardPanel(models.Model):
    # ...snip...  (all the other "model fields" go here)
    panel_type = models.CharField(
        max_length=127,
        choices=PanelTypes.choices,
    )
```