Overview

Take your site to the next level. With your static site generator finished and functioning, it's time to convert it to a server-based application.

This homework represents a fundamental departure from your previous homework. We are no longer building a static site, but instead transforming a static site into a dynamic web app powered by Django and hosted on Heroku.[1]

1 Requirements

1.1 Hard requirements

- **Must be deployed and functioning to Heroku**
- Must use Django to serve up each page of content[2]
- Must use `requests` package (or equivalent) to access one of the following APIs:
  - GitHub to link to all your repositories
  - Mailgun API to create an "email me" contact page
- Previous requirements apply:
  - Must have a Pipfile generated by `pipenv`
  - Must use Django Templates to template the pages[3]

1.2 Soft requirements

- Previous 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 descriptive variable names are a must.

2 Homework Steps

To receive full credit, you must complete all phases. However, the earlier phases are worth more, so you should attempt to fully complete each phase before moving on to the next.

As with the previous homework, 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.

Where to start:

- Create a new repository - start fresh with this one!
- Feel free to use your templates or contents from either HW3 or HW4 – you don't have to have completed HW4 to start this homework.

[1]Note that if you are already using your profile page as your personal site, you may want to keep the previous static iteration as your main site. Make sure you keep these in separate repos, so you can decide which approach you like better.

[2]This iteration of your site should no longer use "static" files for your pages, and thus no longer have the `.html` ending to your URLs.

[3]Jinja templates are okay too, as long as you configure them in your manage.py (or settings.py)

2.1 Phase 1 - Converting to Django

Background: You've learned Django, now's the time to use it to transform your site from being a statically generated site to being a slick Django-powered web app.

Your first goal is to convert your static site into a Django application.

1. Copy the contents of the `minidjango` directory from the 5.3 content into your new repo as a clean, simple starting point for Django.[4]
2. Now, one by one, create new functions in the `views.py` using the existing functions as a reference, and add corresponding paths in `urls.py`.
3. Copy your `base.html` file from previous homework into your new `templates/` directory.
4. Finally, copy your content files into your new Django project. You can leave these in "content" directories for the time being. Add in code necessary to template these content files and send them back to the user, exactly as you had before.

2.1.1 Clue 1

Don't over think this: You don't need (yet) to do things the "correct Django way". Instead, your initial goal should only be to recreate your site, except powered by Django. One view might look like:

```
def about(request):
    content_html = open("content/content.html")
    context = {
        "content": content_html,
    }
    return render(request, "base.html", context)
```

2.1.2 Clue 2

Is the HTML not getting rendered as you expected? You might need to mark it as "safe" with a filter:

```
{{ content|safe }}
```

This will cause the HTML to render as opposed to be displayed.

2.1.3 Phase Summary

To accomplish this, you must recreate your site using the Django web framework.

2.2 Phase 2 - Launch on Heroku

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

See the Heroku Deployment Guide for steps on launching your page to Heroku. It's found in `5.3-recap/guides/heroku_deployment_guide.txt`

[4]If you'd prefer, you can start with the more canonical Django `django-admin startproject`

2.2.1 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.

2.2.2 Phase Summary

To accomplish this, your page must be published live and functioning on Heroku.

2.3 Phase 3 - Adding an API

Background: Now that you've transformed your site into a multi-page web app instead of a mere static site, you can add many more interactive capabilities.

You have two options for adding an API. **Pick one:**[5]

2.3.1 GitHub (Easier)

Use the GitHub API to fetch all of the repositories *you* created.

GitHub Clue Look at 5.3's `minidjango` activity – this does something similar for your instructor's repos. Modify it to be yours, and to include the extra functionality.

GitHub Requirements

- At least one of your pages must contain a list of all your GitHub repositories.
- Must use either `requests`, or equivalent package.[6]
- **Each item in the list must be a link to the GitHub repository itself.**[7]
- Optional: Fetch and display on your homepage other aspects of your repos or account.

2.3.2 Mailgun (Harder)

Use the Mailgun API to create a "mail me" contact form. This API is free for less than 10,000 emails, although it does require a credit card to sign up, and will require you to use an API key and secret.

[5]If you have a different API you want to use, just ask your instructor. If it's sufficiently complicated as ether GitHub or Mailgun, you can use it instead. Also, of course, feel free to add as many other APIs as you might want too :)

[6]If you'd want to use a different requests package or the built-in Python functionality for this, or a GitHub-specific API package on PyPI, feel free.

[7]This is NOT found in the example one – this is new functionality.

Mailgun Clue 1 To create an HTML form, use something like this:

```
<form action="/send-email" method="POST">
    <input name="name" placeholder="Name" />
    <input name="email" type="email" placeholder="Email" />
    <textarea name="message">
    </textarea>
    <button type="submit">Submit</button>
</form>
```

And within Django, use code something like this (must be exposed at the path `/send-email` for this example to work):

```
def send_email(request):
    name = request.POST["name"]
    email = request.POST["email"]
    message = request.POST["message"]

    # Do something with these three variables...

    return redirect("/")  # Return a redirect!
```

Mailgun Clue 2 *Background:* API secrets are like passwords: You must keep your API secrets from anyone who could abuse your API key to send emails with your account. Not only that, Mailgun will immediately disable your account if they notice you published your API secret, so you have to do it the right way!

In your source code, instead of having the API key embedded directly, write something like:

```
import os
mailgun_api_key = os.environ["MAILGUN_API_KEY"]
```

This is the best practice. It fetches the API key from the "environmental variables".[8] To set this environmental variable, you can run a command in Bash like the following:

```
export MAILGUN_API_KEY="XXXXXXXXXXXXXXXXXXXXXX"
```

You will have to run that command for every new terminal that you create.[9] Now, you need to set the variable on Heroku, also:

```
heroku config:set MAILGUN_API_KEY="XXXXXXXXXXXXXXXXXXXXXX"
heroku config # Run this to confirm it was set
```

[8]These are variables set and stored in your operating system as opposed to on your code. Think of them as "configuration on the server", in contrast to configuration stored inside your Python code.

[9]Alternatively, you can put it in your `.bashrc`, or use an ".env" file, which you then load by either doing `source .env`, or by using n environment variable manager such as `envparse`

Mailgun Requirements

- Must have an HTML form on a contact page.[10]
- Must use either `requests`, or equivalent package.[11]
- Must use the Mailgun API to send all contact requests to your email
- **Do not** include the Mailgun API key in your source code in your repo, or risk having your account disabled.
- *Optional:* Send an email to the user sending the message also, saying hi.

2.3.3 Phase Summary

To accomplish this, you must integrate a third-party API (Mailgun or GitHub) into your Django web app.

2.4 Phase 4 - Django template inheritance

Background: Currently, you are not using templating according to industry best-practices for Django, which limits our ability to have more complicated pages and functionality.

Using template inheritance allows us to have a "base" template, that other HTML code "inherits" from.[12] In practice, this is pretty similar to what you are already doing with content pages, except it will require use of different syntax and features of Django templates.

Modify your content pages and `base.html` template, such that the content pages are all templates extending the `base.html` template. Check out the Django Girls tutorial on extending: `tutorial.djangogirls.org/en/template_extending/`

2.4.1 Clue

Examine the following templating example code. This shows how to use template extension to properly manage infrequently changed content pages. Notice that this replaces the {{ `content` }} and {{ `title` }} template variables. Instead, we will use the template inheritance technique to insert the content.

base.html The "base" template.

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

<!-- navigation goes here... -->

<div class="content">
    {% block content %}
    {% endblock content %}
</div>
```

[10] Feel free to use Django's Form system.

[11] If you'd want to use a different requests package or the built-in Python functionality for this, or a GitHub-specific API package on PyPI, feel free.

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

```
<footer>
    (C) 2019 Jane Q. Hacker (janeq@hacker.com)
</footer>
</body>
</html>
```

about.html Individual “content” page.

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

{% block title %}
About Me
{% endblock title %}
{% block content %}
<h1>About me</h1>
<p>This is all about me...</p>
{% 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 content(request):
    context = {}
    return render(request, "about.html", context)
```

2.4.2 Phase Summary

To accomplish this, you must use blocks and `extends` keyword to change your different pages into extending the main page as opposed to being templated into it.