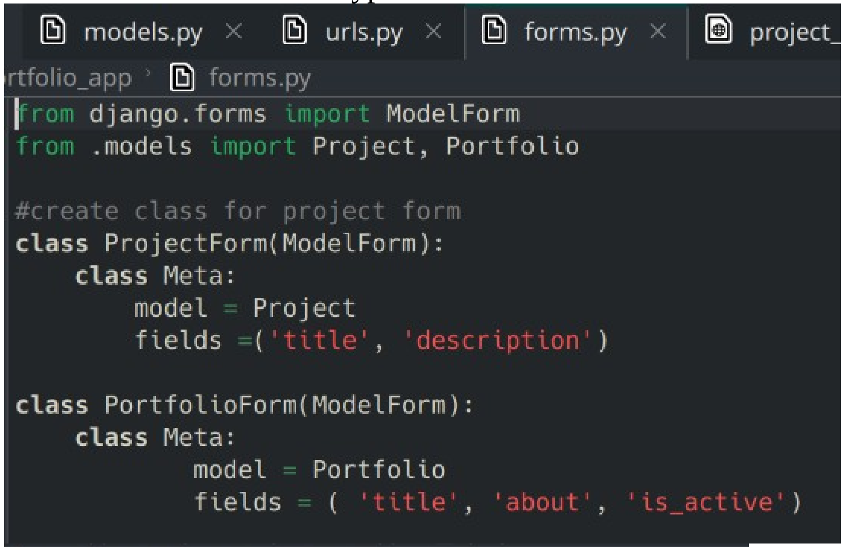
Austin Byrd
Django Forms

Part 2: Create Documentation on Implementing Forms

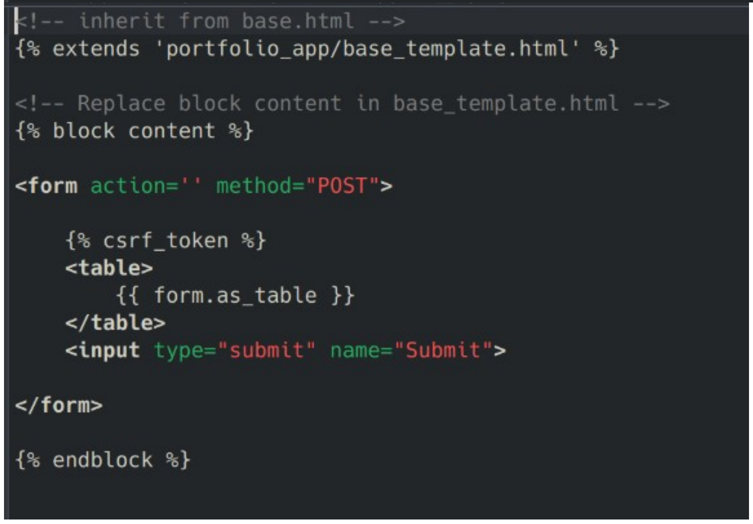
Explore what you just implemented for creating, updating and deleting items using forms. Start making connections, find resources and create technical notes to share with your team. This will be a part of your document to demonstrate communicating effectively in a professional written context. Submit **a separate document with examples and explanations.**

2.
Forms and Links to resources:
https://docs.djangoproject.com/en/4.2/ref/forms/
https://docs.djangoproject.com/en/4.2/topics/forms/
Explanation of CRUD in Django
https://www.geeksforgeeks.org/django-crud-create-retrieve-update-delete-function-based-views/
https://medium.com/@20ce125/django-crud-create-retrieve-update-delete-operations-441a8a296119

Forms are made from classes (structures of what data is to be filled in)
and HTML Form Templates (pages users will enter the data into)
and Views that handle the type of Form Submission that will be used (Create, Update etc.)



```
from django.forms import ModelForm
from .models import Project, Portfolio

#create class for project form
class ProjectForm(ModelForm):
    class Meta:
        model = Project
        fields =('title', 'description')

class PortfolioForm(ModelForm):
    class Meta:
            model = Portfolio
            fields = ( 'title', 'about', 'is_active')
```

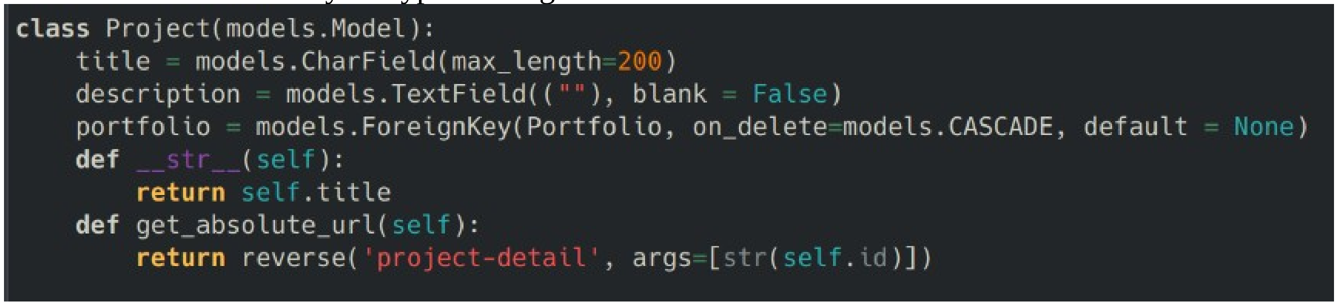


```
<!-- inherit from base.html -->
{% extends 'portfolio_app/base_template.html' %}

<!-- Replace block content in base_template.html -->
{% block content %}

<form action='' method="POST">

    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" name="Submit">

</form>

{% endblock %}
```

Create

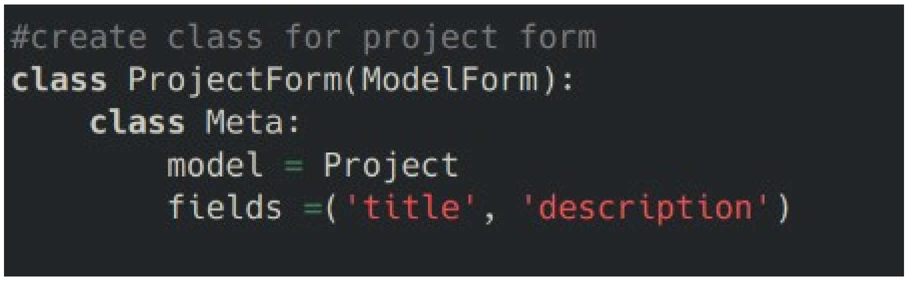
Allow users to create new records in the database using a form.

From the Model class of your type of thing to create



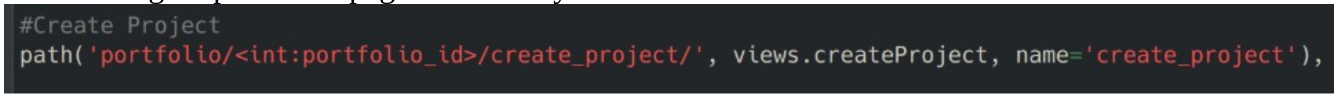
```python
class Project(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField((""), blank = False)
    portfolio = models.ForeignKey(Portfolio, on_delete=models.CASCADE, default = None)
    def __str__(self):
        return self.title
    def get_absolute_url(self):
        return reverse('project-detail', args=[str(self.id)])
```

You make a form



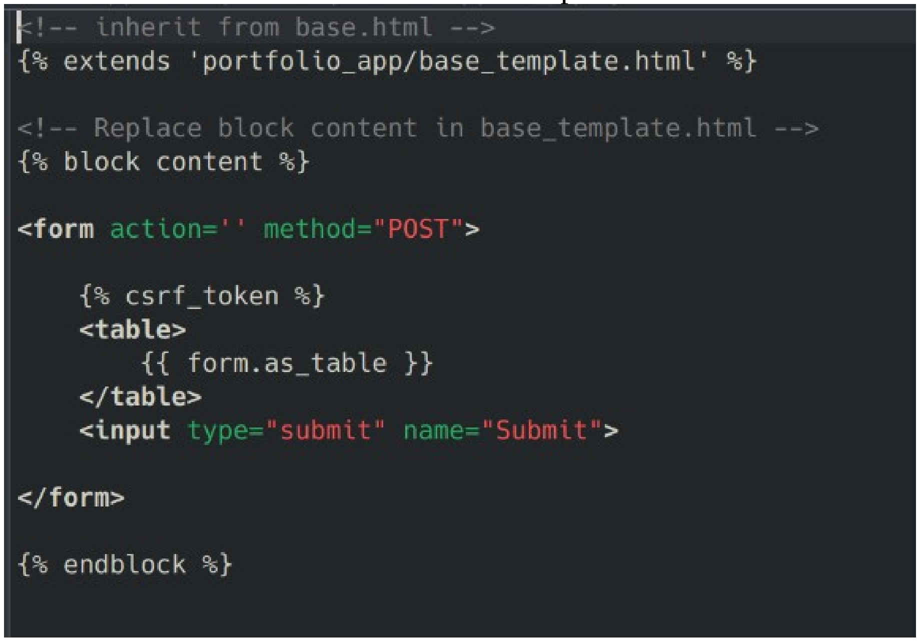
```python
#create class for project form
class ProjectForm(ModelForm):
    class Meta:
        model = Project
        fields =('title', 'description')
```

that is brought up within a page accessed by a url



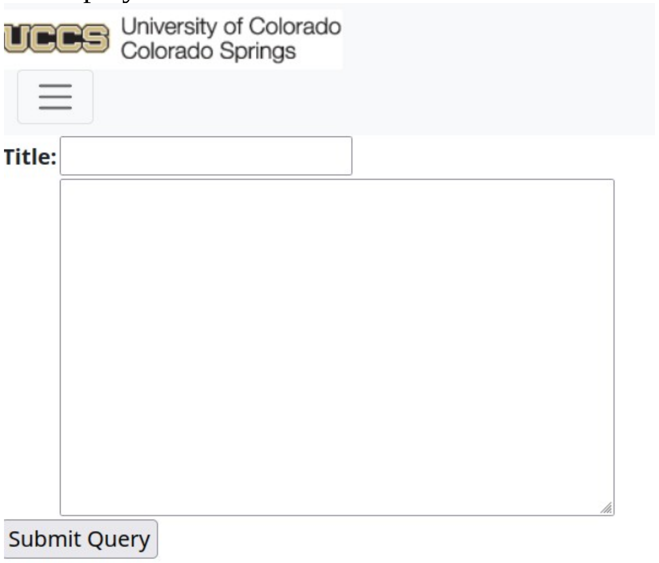
```python
#Create Project
path('portfolio/<int:portfolio_id>/create_project/', views.createProject, name='create_project'),
```

That is the class that is then used in the template



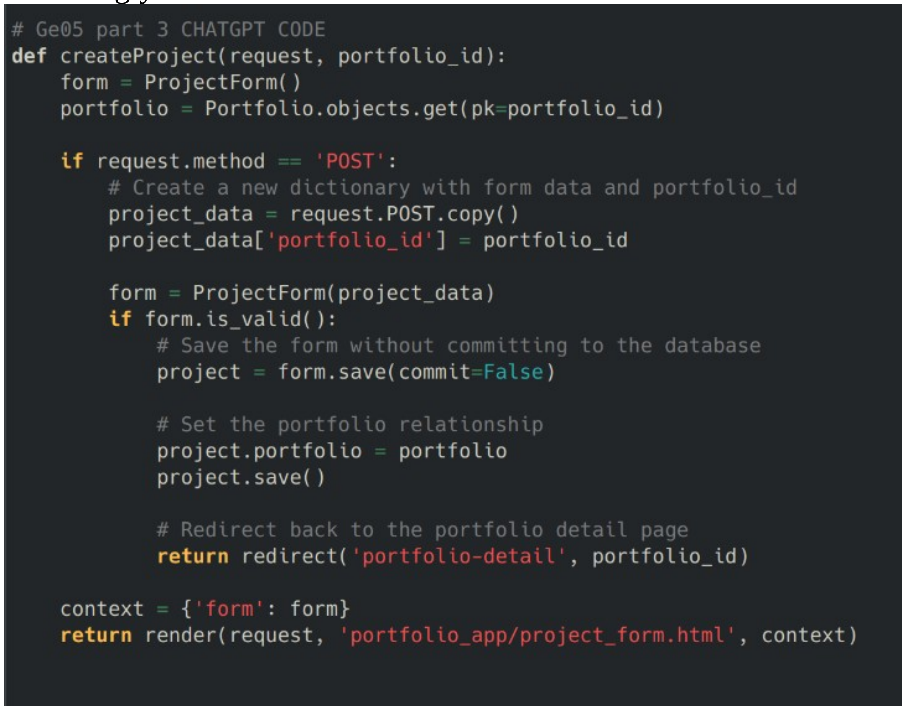
```
<!-- inherit from base.html -->
{% extends 'portfolio_app/base_template.html' %}

<!-- Replace block content in base_template.html -->
{% block content %}

<form action='' method="POST">

    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" name="Submit">

</form>

{% endblock %}
```

To display the form to the user



They can then add that project to the database which updates any lists/details views so that the new project is available.

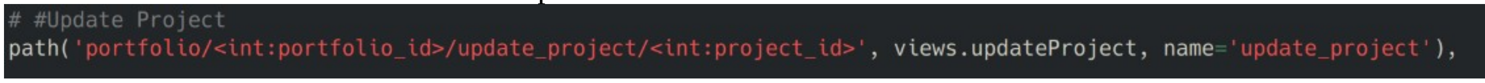
This is done with a view that can take in HTTP requests and act on the database and redirect the page accordingly.



```python
# Ge05 part 3 CHATGPT CODE
def createProject(request, portfolio_id):
    form = ProjectForm()
    portfolio = Portfolio.objects.get(pk=portfolio_id)

    if request.method == 'POST':
        # Create a new dictionary with form data and portfolio_id
        project_data = request.POST.copy()
        project_data['portfolio_id'] = portfolio_id

        form = ProjectForm(project_data)
        if form.is_valid():
            # Save the form without committing to the database
            project = form.save(commit=False)

            # Set the portfolio relationship
            project.portfolio = portfolio
            project.save()

            # Redirect back to the portfolio detail page
            return redirect('portfolio-detail', portfolio_id)

    context = {'form': form}
    return render(request, 'portfolio_app/project_form.html', context)
```

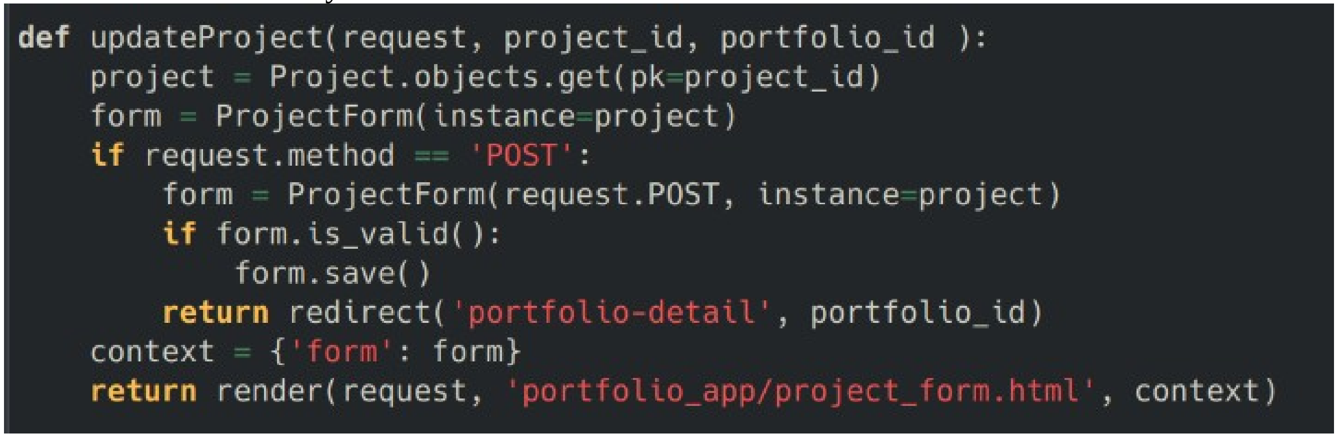
The view takes in an HTTP request, as well as the portfolio_id on which the project will be added to. It can then display the form and when the submit button is pushed it can validate the form and then save the project ot the according portfolio. Once this is done it takes the user back to the portfolio-details page they were on before and that is the creation of a new project.

Update: Update uses most of the same code as Create but the View handles it differently.
The Model of Project is the same
The form used to create and update is the same (update just already has data in it)
The URL is different as this is a different place on the site:



```
# #Update Project
path('portfolio/<int:portfolio_id>/update_project/<int:project_id>', views.updateProject, name='update_project'),
```

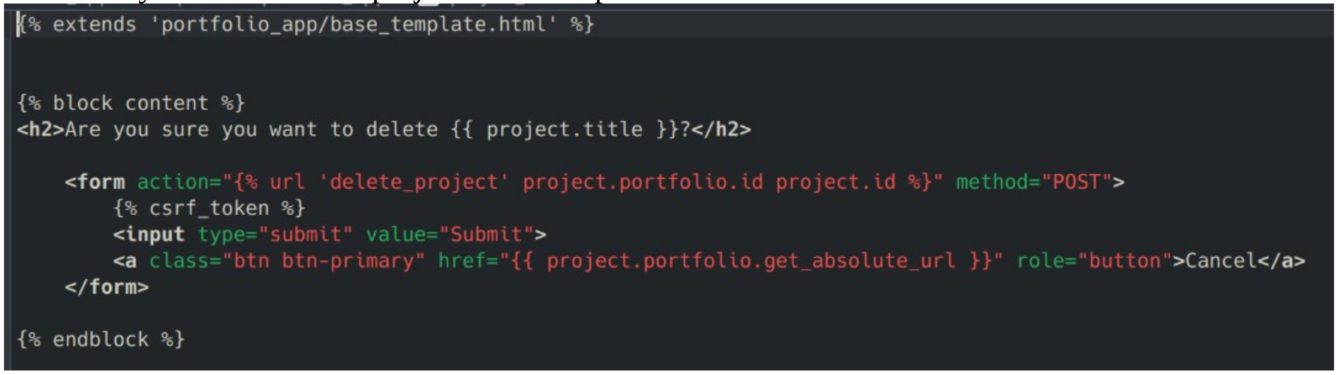
The HTML page is the same as they are entering it into the same Form
The View however is very different:



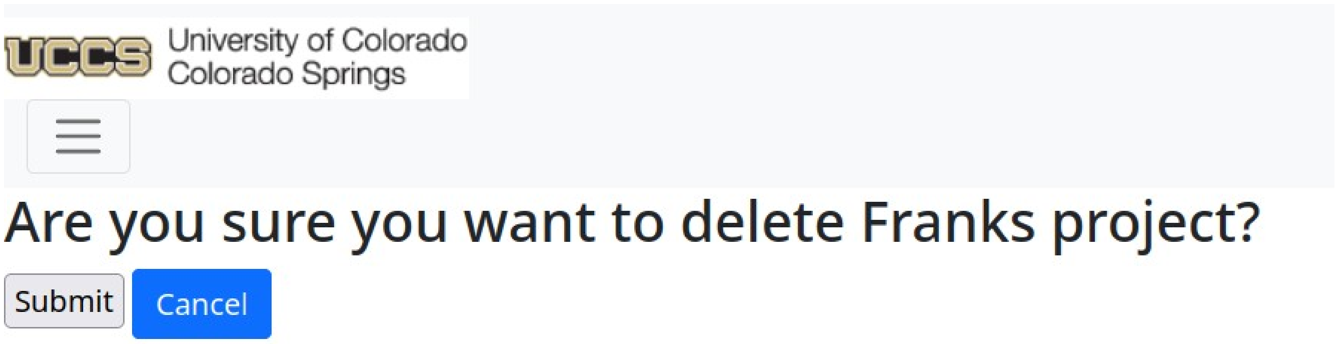
```
def updateProject(request, project_id, portfolio_id ):
    project = Project.objects.get(pk=project_id)
    form = ProjectForm(instance=project)
    if request.method == 'POST':
        form = ProjectForm(request.POST, instance=project)
        if form.is_valid():
            form.save()
        return redirect('portfolio-detail', portfolio_id)
    context = {'form': form}
    return render(request, 'portfolio_app/project_form.html', context)
```

This view takes a request and a portfolio_id the same as the create view, but the Update view also requires the project_id as it is updating an existing project not creating one, Update also fills in the form with the data already provided for the project being updated. Then when the user is ready to submit their changes they hit the same submit button as create but it updates that existing projects data with the new values in the fields. Thus updating the database with the data from the user.

Delete:
Model: To delete a project we use the same project model as always
Form: No form needed to delete
HTML/Page
The html of the delete page includes a confirmation and a cancel option which allow the user to be certain they want to delete a project from that portfolio.

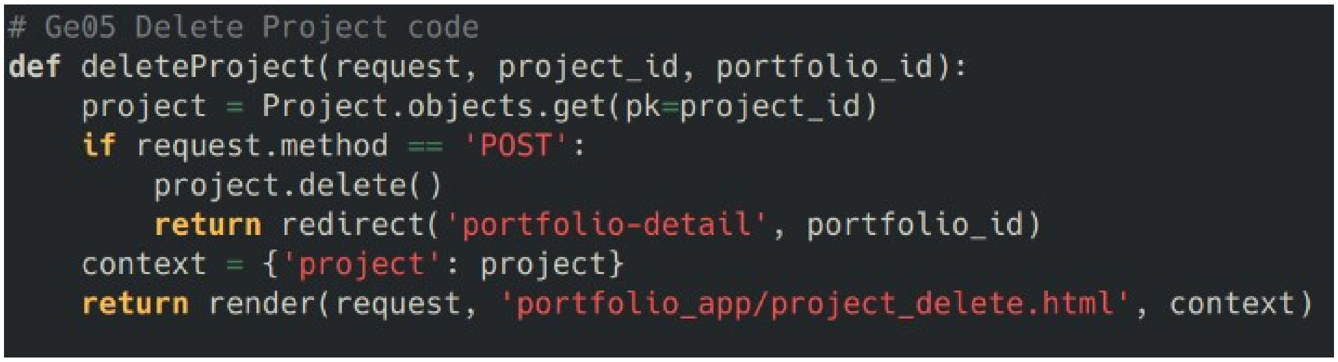


```
{% extends 'portfolio_app/base_template.html' %}


{% block content %}
<h2>Are you sure you want to delete {{ project.title }}?</h2>

    <form action="{% url 'delete_project' project.portfolio.id project.id %}" method="POST">
        {% csrf_token %}
        <input type="submit" value="Submit">
        <a class="btn btn-primary" href="{{ project.portfolio.get_absolute_url }}" role="button">Cancel</a>
    </form>

{% endblock %}
```



Are you sure you want to delete Franks project?

View: The Delete view



```
# Ge05 Delete Project code
def deleteProject(request, project_id, portfolio_id):
    project = Project.objects.get(pk=project_id)
    if request.method == 'POST':
        project.delete()
        return redirect('portfolio-detail', portfolio_id)
    context = {'project': project}
    return render(request, 'portfolio_app/project_delete.html', context)
```

This view takes in the HTTP request, project_id, and portfolio_id. It can then confirm the deletion and delete the project or it can cancel both redirect you to the portfolio-details page you arrived from.

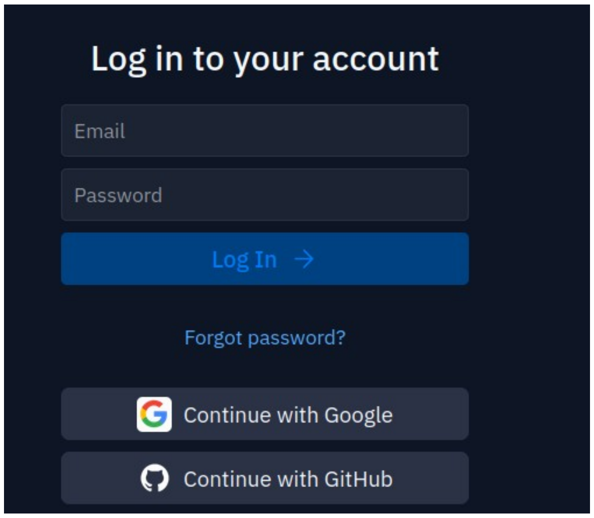
Part 3 and 4: Create Documentation

Split up the following 2 tasks within your team to explore (about 60 minutes). If you have 3 people on your team you can all do the same task. This will be a part of your document to demonstrate communicating effectively in a professional written context the task you explored below. Do not worry if you do not figure everything out but you should have documentation showing what resources you used, the steps you did, issues encountered and a summary of what you learned.

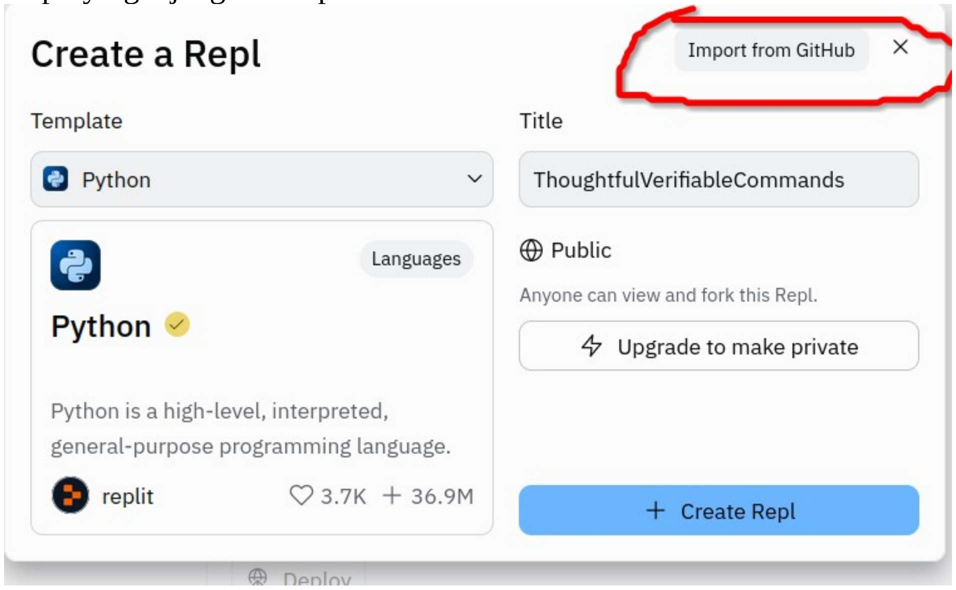
1. **Deploy on Replit**
   a. Deploying Django to Replit
   b. Connecting Replit to GitHub
   c. Using Git on Replit
   d. Deployment checklist | Django documentation

Replit Setup and Deployment
3.

Getting Started with Replit:
You can log into Replit with your Github account making it very easy to make an account/log in and this also streamlines bringing in github repositories



Deploying Django to Replit:



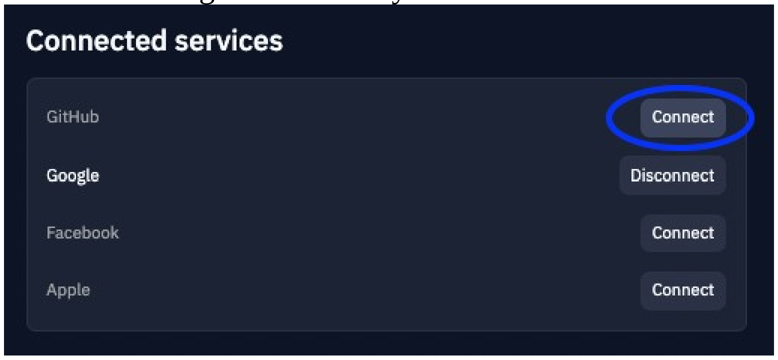
ThoughtfulVerifiableCommands

Anyone can view and fork this Repl.

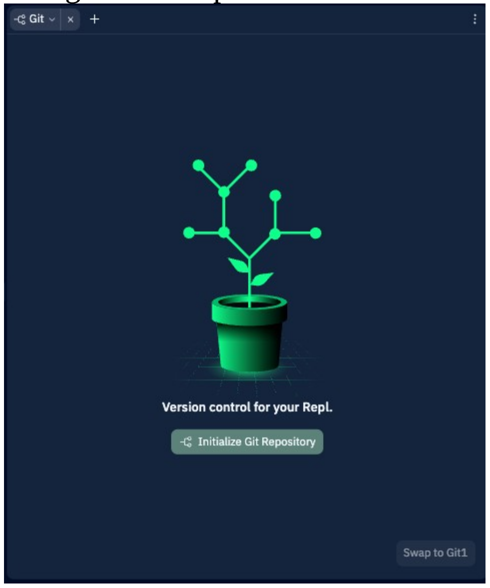
Python is a high-level, interpreted, general-purpose programming language.

This is the way to create a Repl from a Django app as the Documentation provided (Deploying Django to Replit) indicates to use a template called Django App Template but replit did not have such a template. Therefore the Python template is used and then you can Import from Github the repo that is the Django project.

Connecting Replit to Github:
If you did not login with Github then to connect Github to Replit you must go to your Replit account and connect to github manually

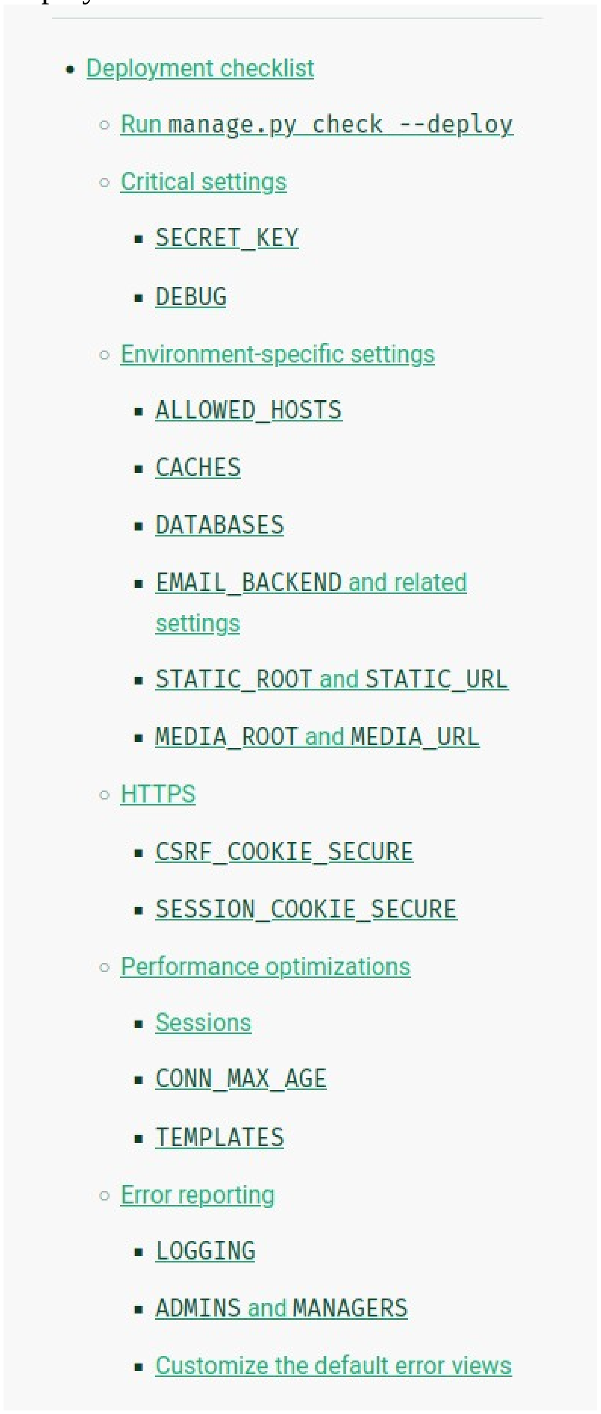


Using Git on Replit:



Select the Git from the Tools section
Then initialize a repository
If you have not already connect to Github
Create a repository
Add something
Review the staged changes
Push your initial commit
All the regular git utils are available from the replit interface.

Deployment Checklist:



- Deployment checklist
  - Run manage.py check --deploy
  - Critical settings
    - SECRET_KEY
    - DEBUG
  - Environment-specific settings
    - ALLOWED_HOSTS
    - CACHES
    - DATABASES
    - EMAIL_BACKEND and related settings
    - STATIC_ROOT and STATIC_URL
    - MEDIA_ROOT and MEDIA_URL
  - HTTPS
    - CSRF_COOKIE_SECURE
    - SESSION_COOKIE_SECURE
  - Performance optimizations
    - Sessions
    - CONN_MAX_AGE
    - TEMPLATES
  - Error reporting
    - LOGGING
    - ADMINS and MANAGERS
    - Customize the default error views

The Deployment Checklist is the list of dependent statuses and values that must be confirmed before deploying with Replit. This ensures the Repl is ready for deployment.
Manage.py check –deploy option makes the a lot of the rest of the settings set to the desired values automatically.
That is how it is usually done to save time.

Otherwise: manually set the settings for the rest of the setting in the deployment checklist