**GE05 Documentation**

Bits & Bytes

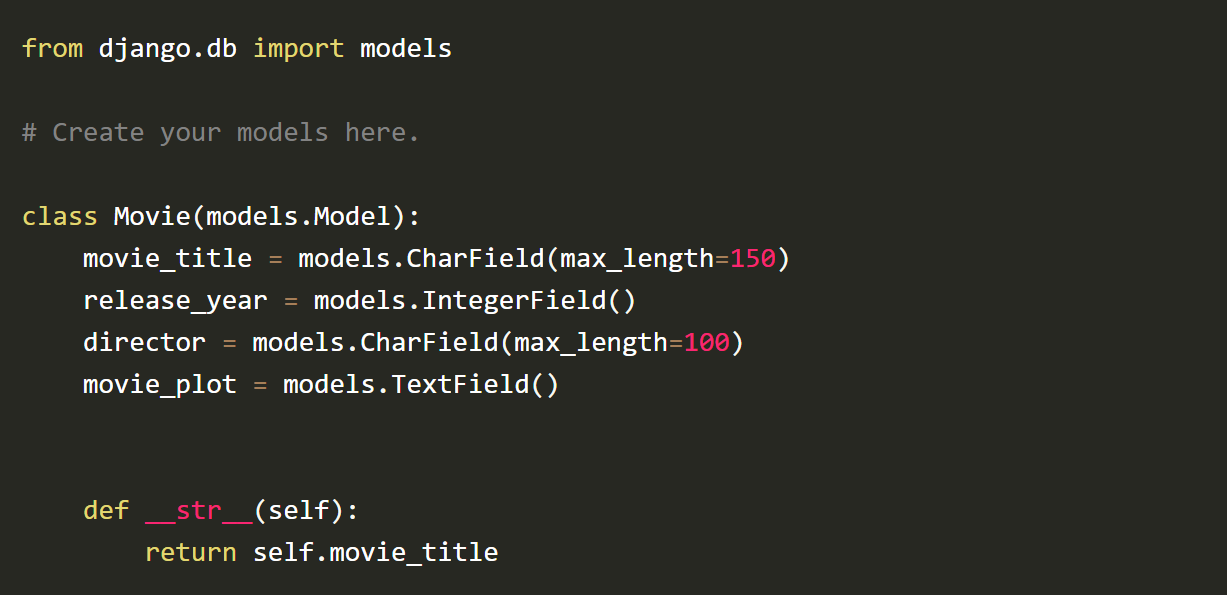
Christopher Romo

## Part 2 – MVT Model & Forms

Throughout this part, I am going to explain the MVT Model which includes forms, which will allow for easier reading. The MVT Model is what the Django framework uses in order to provide a system for making applications. Essentially, URLs, Views, Models, and Templates all work together to provide a site to users, which can perform a variety of tasks, including HTML forms, creating instances of the models, etc.

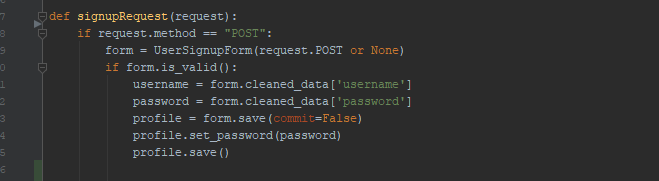
## Models

A model is a class that makes objects based on the database. For example, a database can have a row of projects and their information. A single project can then be made into an object using this class, the model. For example, here is a model:



## Views & Forms

The view does a lot of work. In updateProject, for example, you can see that this function opens up a form to edit a single instance of a portfolio. Templates are then referenced, which is where the user will be redirected. For example, once the user saves the changes, they are redirected back to the portfolio detail screen. Here is a form request using views.

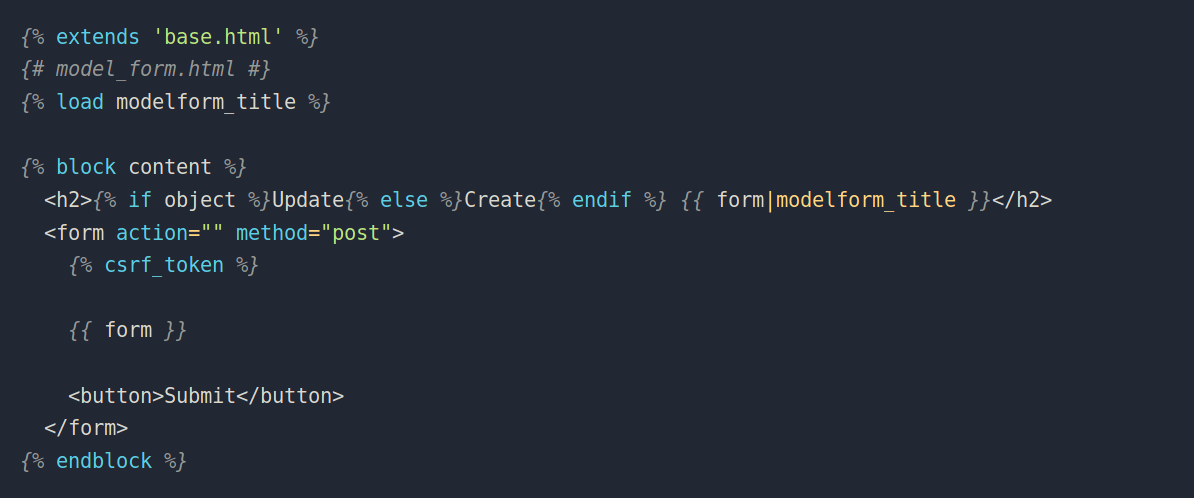


## URLs

URLs can be used to navigate the website, as they provide resources that help Django understand where the user is on the website.

## Templates

Templates act as the representation of the website, where things are situated, how they look, and what you can interact with. We are using bootstrap in the class, as well as elements of CSS and HTML. All of this works together to provide a visual to the user. Below is an example of a template for a form.



You can see “form.as\_table” which translates the PortfolioForm (defined in the forms.py file) into an interactable form.

## Overview & Resources

So, in short, the URL provides resources and navigation, the view provides the functionality, models provide objects and their instances, and templates provide the look for the website. All work together to make the website work.

Here are some resources I used to understand these concepts:

<https://openclassrooms.com/en/courses/6967196-create-a-web-application-with-django/7349667-update-a-model-object-with-a-modelform>

<https://chat.openai.com/share/e5e5d2b2-58ed-4420-ab7c-0e79732a697b>

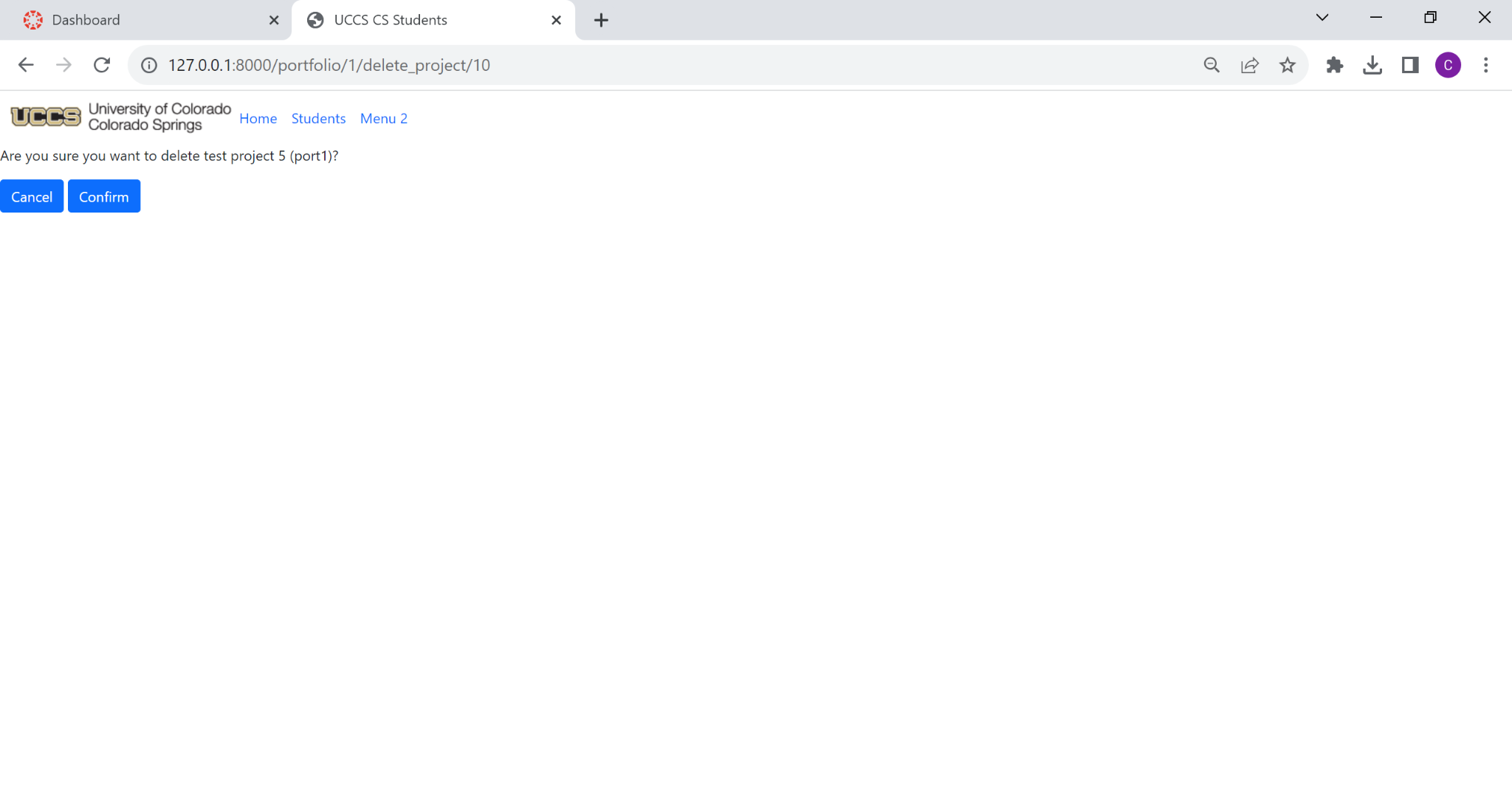
<https://www.youtube.com/watch?v=ffI24V35kfc>

## Part 4 - Bootstrap

## Buttons

Using bootstrap’s code for buttons, I was able to make the buttons match.

This is what it looks like on the website:



Before, the button used the HTML Form’s submit button. I was able to find the submit button on bootstrap’s documentation, and used it in place of the normal HTML form’s submit button, making the two buttons match perfectly.

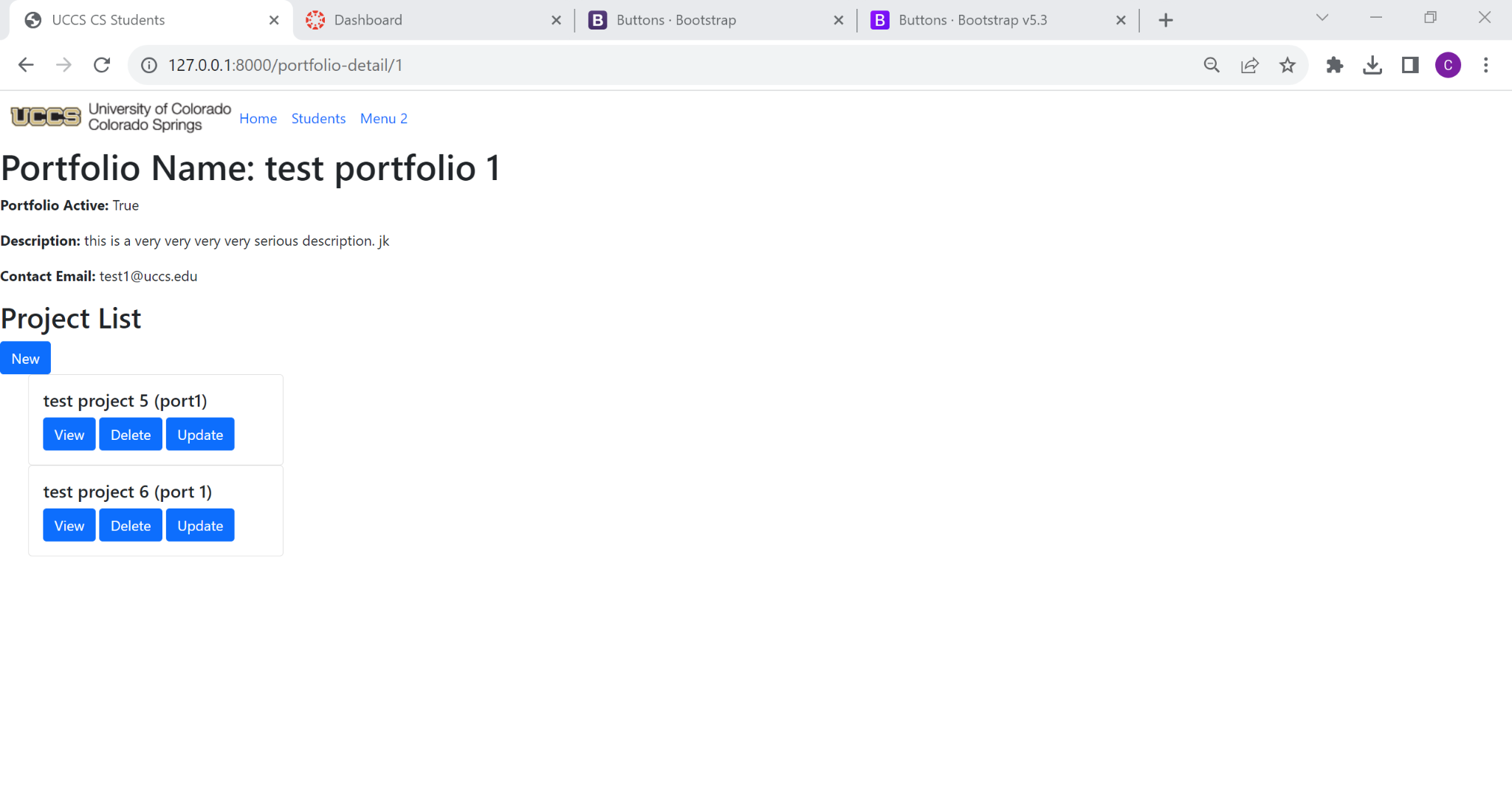
Here is the documentation I used:

<https://getbootstrap.com/docs/5.3/components/buttons/#base-class>

## Cards

Using bootstrap’s code for cards, I was able to make a card for each project.

This is what it looks like on the website:



I felt cards were the best way to replace the list that was already there, as they make it easy for everything to align inside. Using bootstraps documentation, I was able to make a card, which I placed in a for loop. For every project, there is a new card.

Here is the documentation I used:

<https://getbootstrap.com/docs/5.3/components/card/#about>

## Toggle Theme

While I tried hard to get this to work, a lot of sources stated that JavaScript can play a key role in creating a theme toggle dropdown. Here is the documentation that I tried to use to understand this concept.

<https://getbootstrap.com/docs/5.3/customize/color-modes/>

Darion Badillo

Part 2

# **Forms**

## **What are they?**

Forms are essential tools in building web applications that are necessary for handling user input. In Django, it provides several avenues for interactions such as handling, validating, and processing user input, website forms, and other user-based interactions.

Example Form (Django (HTML):

Source: [Django Official Forms Documentation](https://docs.djangoproject.com/en/4.2/topics/forms/)

This form above, coded in CSS, is used to provide a layout of how the information will be displayed for the user to interact with. Combining this form with the Python code below, will allow the user data to be implemented, saved, and stored within the database, effectively creating, updating, or deleting objects.

Example Form (Python):

#create class for project form

class PortfolioForm(ModelForm):

class Meta:

model = Portfolio

fields =('title', 'contact\_email', 'is\_active', 'about')

Source: GE05 Portfolio App – Darion Badillo

## **Using Forms**

Forms can be used for a multitude of reasons. In our app GE05, we implemented forms to Create, Update, and Delete certain objects based on user input and interaction.

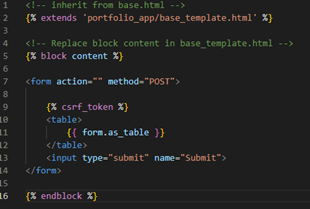
### **Update Object:**

To help you better understand forms, I will provide an example of how we updated existing objects stored in the database with new, updated user information.

First, we need to create an html form. This HTML form, known as project\_form.html, we used for our Project is displayed as such:

This form can be broken down as follows.

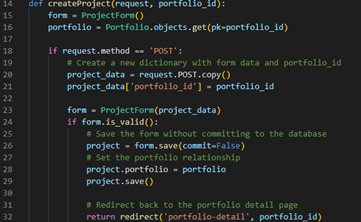
· This form, starting from line 1, extends an existing html template that we have already created.

· Line 7 is where the form creation truly begins. This line essentially states that the data submitted by the form will be submitted by the POST http request.

· Line 9 is an essential part of Django security that protects against CSRF, Cross-Site Request Forgery, attacks.

· Then, a table element is created that takes an already existing form, displays it for the user, and fills it with its existing information.

· The user can then manipulate the data within and submit for query form using line 13’s submit function.



How do we interact with this data, though? We would need to create a function inside of our views.py folder. The code is displayed as such. What does this code do? It creates a project object, verifies the form, submits user input, saves it, and redirects the user back to where the began.

The data is saved, and the world is safe. Similar feats can be accomplished using this base knowledge on forms. For instance, we can now take the same layout for the html create\_project and implement that same code in other forms such as update\_project.

# **Button**

Users can’t truly interact with the User interface unless there are interactable icons or buttons for them to click on. Without forms, the user could not interact with the page. Thanks to bootstrap5 and Django, we can implement these.

There are a multitude of different ways buttons can be displayed on your website.

These examples provide a solid glimpse of what can be accomplished.

These buttons were created using the following code:

These buttons, in order with picture, display the color, look, feel, and even context of what a button can accomplish. We implemented this in our project by providing a clickable option for the user to create, update, or delete a project or portfolio.

Button Code for Project:

<a class="btn btn-primary" href="{% url 'create\_project' portfolio.id %}" role="button">New</a>

<ul>

{% for proj in portfolio.project\_set.all %}

<li>{{ proj.title }}</a>

<a class="btn btn-primary" href="{{ proj.get\_absolute\_url }}" role="button">View</a>

<a class="btn btn-primary" href="{% url 'update\_project' portfolio.id proj.id %}" role="button">Update</a>

<a class="btn btn-danger" href="{% url 'delete\_project' portfolio.id proj.id %}" role ="button">Delete</a></li>

{% endfor %}

</ul>

This code essentially iterates through all existing projects in a currently active portfolio and lists them alongside buttons that will allow the user to view, update, or delete that project. How does it look? Like this!

As you can tell, the buttons allow you to see what the USER can accomplish using context from the page. Blue implies safety while red implies a dangerous warning about what can happen with it.

## **Forms Resources:**

·  [Official Django Documentation](https://docs.djangoproject.com/en/4.2/topics/forms/)

·  [Official Django Dev Documentation for Forms](https://docs.djangoproject.com/en/4.2/ref/forms/)

·  [Bootstrap 5.2 Buttons](https://getbootstrap.com/docs/5.2/components/buttons/)

·  [Geeks For Geeks: Forms](https://www.geeksforgeeks.org/django-forms/)

Part 4

# **Light Mode and Dark Mode Implementation**

Light mode and dark mode were extremely difficult to implement for my project. (Our whole team collectively had issues with it).   
 Bootstrap natively offers a light mode for their user to interface with. In recent years, a dark mode has been gaining popularity frequently and quickly. It is accessible and allows users to *feel* better about what they’re using.

### **Resources:**

·  [Bootstrap Light/Dark Mode Documentation](https://getbootstrap.com/docs/5.3/customize/color-modes/)

·  [W3Schools](https://www.w3schools.com/bootstrap5/bootstrap_dark_mode.php)

Emiliano Chavez De La Torre

# **Parts 2 & 4**

## **Table of Contents**

*Part 2*

1. [Forms in Django and in General](#_s1k0jajodg0w)

a. [General Ouline of a Form](#_a7k3f1rvd4d)

b.  [Common Parts of a Form](#_tuakxav3akyv)

2. [General Forms on Bootstrap](#_aqs3a0kn7cd4)

3. [HTTP Methods with Forms](#_jd09gosksmg3)

4. [Accessibility](#_mnltew58fdlz)

*Part 4*

1. [Adding Buttons and Making them Consistent](#_hljrrlye9gha)

a.  [Html Code from Bootstrap](#_9g1gnw8fyjts)

2. [Toggle Themes](#_do2zunchk4u7)

a.  [Dark Mode](#_u55lzepbc52d)

b. [Dropdowns](#_dwh7161z0xgj)

## **Forms in Django and in General**

### **Sources**

Ø <https://docs.djangoproject.com/en/4.2/topics/forms/>

Ø <https://www.geeksforgeeks.org/html-action-attribute/>

Ø <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#attributes>

· Forms are just a collection of elements that allows the user to enter information like:

o Text

o Options

o Manipulate objects or controls.

· The purpose of a form is to gather information from the user to send back to the server.

#### **General Outline of a Form**



*Source of Screenshot:* [*https://docs.djangoproject.com/en/4.2/topics/forms/*](https://docs.djangoproject.com/en/4.2/topics/forms/)

#### **Common Parts in a Form**

· <form></form>

o The form tag is how we define a form in html.

o Attributes in the form tag:

§  [action= “ ”](https://www.geeksforgeeks.org/html-action-attribute/)

· The action attribute specifies where to send the form data once a user fills out and submits the form. Generally, the form data is sent to a server.

§ method *(refer to the HTTP Methods with Forms)*

· <label></label>

o The label tag shows a text label for a specific input.

§ From the screen shot, we can see that the attribute for=”your\_name” is being used.

· The for attribute shows that the label is for the input value that I denoted by id=”your\_name”

· [<input></input>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#attributes)

o The input tag shows an area where to input information. There are many types of inputs.

§ To denote the type of input we want we use the type=” ” attribute.

· Other type of inputs

o Checkbox

o Button

o Date

o Email

o File

o Image

o Number

o Password

o Radio

o Range

o Reset

o Tel

o Submit

o Text

o URL

o etc.….

§ In our screenshot, all we are asking is for text input, which will display a text box.

## **General Form on Bootstrap:**

### **Sources**

Ø <https://getbootstrap.com/docs/5.3/forms/overview/>

HTML Code

|  |
| --- |
| <form>  <div *class*="mb-3">  <label *for*="exampleInputEmail1" *class*="form-label">Email address</label>  <input *type*="email" *class*="form-control" *id*="exampleInputEmail1" *aria-describedby*="emailHelp">  <div *id*="emailHelp" *class*="form-text">We'll never share your email with anyone else.</div>  </div>  <div *class*="mb-3">  <label *for*="exampleInputPassword1" *class*="form-label">Password</label>  <input *type*="password" *class*="form-control" *id*="exampleInputPassword1">  </div>  <div *class*="mb-3 form-check">  <input *type*="checkbox" *class*="form-check-input" *id*="exampleCheck1">  <label *class*="form-check-label" *for*="exampleCheck1">Check me out</label>  </div>  <button *type*="submit" *class*="btn btn-primary">Submit</button>  </form> |
|  |

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## **HTTP Methods with Forms:**

### **Sources**

Ø <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form>

· The HTTP method to submit the form with. The only allowed methods/values are (case insensitive):

· post: The POST method; form data sent as the request body.

|  |
| --- |
| · This is a form that does a post request. It will post to the current URL/URI that the form is on. The form does the following:  o Prompts the user for their name.  o There is a default button for the user to click. Once the user clicks this button, the POST method is initialized and sent to the server.      *<!-- Form which will send a POST request to the current URL -->*  <form *method*="post">  <label>  Name:  <input *name*="submitted-name" *autocomplete*="name" />  </label>  <button>Save</button>  </form> |
| · This is a form that does a post request. It will post to the current URL/URI that the form is on. The form does the following:  o The field set tag is just adding something additional to the form. In this case there are two radio buttons being added so that user can click yes or no to the terms.    *<!-- Form with fieldset, legend, and label -->*  <form *method*="post">  <fieldset>  <legend>Do you agree to the terms?</legend>  <label><input *type*="radio" *name*="radio" *value*="yes" /> Yes</label>  <label><input *type*="radio" *name*="radio" *value*="no" /> No</label>  </fieldset>  </form> |

· get (default): The GET; form data appended to the action URL with a? separator. Use this method when the form has no side effects.

|  |
| --- |
| · This is a form that does a get request. What this form sets out to achieve is:  o The user types in their name.  o Once the user clicks save, the user’s information will most likely be retrieved back from the database. The precondition to this is that the user must have information in the database for this to work.      *<!-- Form which will send a GET request to the current URL -->*  <form *method*="get">  <label>  Name:  <input *name*="submitted-name" *autocomplete*="name" />  </label>  <button>Save</button>  </form> |

## **Accessibility**

### **Sources**

Ø <https://getbootstrap.com/docs/5.3/forms/overview/>

Ø <https://www.w3.org/WAI/tutorials/forms/>

· Providing accessibility is a critical aspect of forms. Ways to achieve accessibility in forms is by providing the following:

§ A label

· <label> Enter Label Here</label>

§ Descriptive text for buttons

· <button> Descriptive Text Enter Here </button>

§ Providing a title

· <title>Title name here</title>

§ Form Instructions

· Providing instructions to complete a form can provide users with understanding of what the form is trying to achieve. Also, it tells users how to complete the form.

§ Field sets and Legends

· Field sets are used to group related elements together.

· Legends serve as labels to those fields sets.

|  |
| --- |
| <fieldset>  <legend>Do you agree to the terms?</legend>  <label><input *type*="radio" *name*="radio" *value*="yes"/>Yes</label>  <label><input *type*="radio" *name*="radio" *value*="no" />No</label>  </fieldset> |

§ Multiple Page Forms

· Multiple page forms help to chunk pieces of information into logical steps.

· This helps users to visualize the progress made on a form as well.

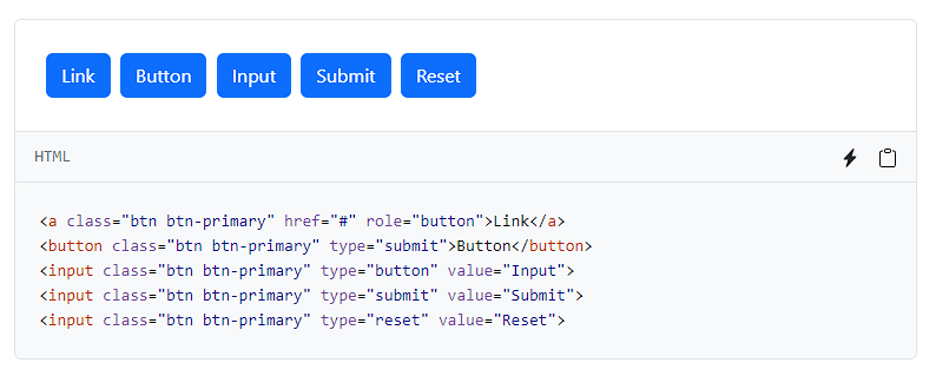
## **Adding Buttons and Making them Consistent**

### **Sources**

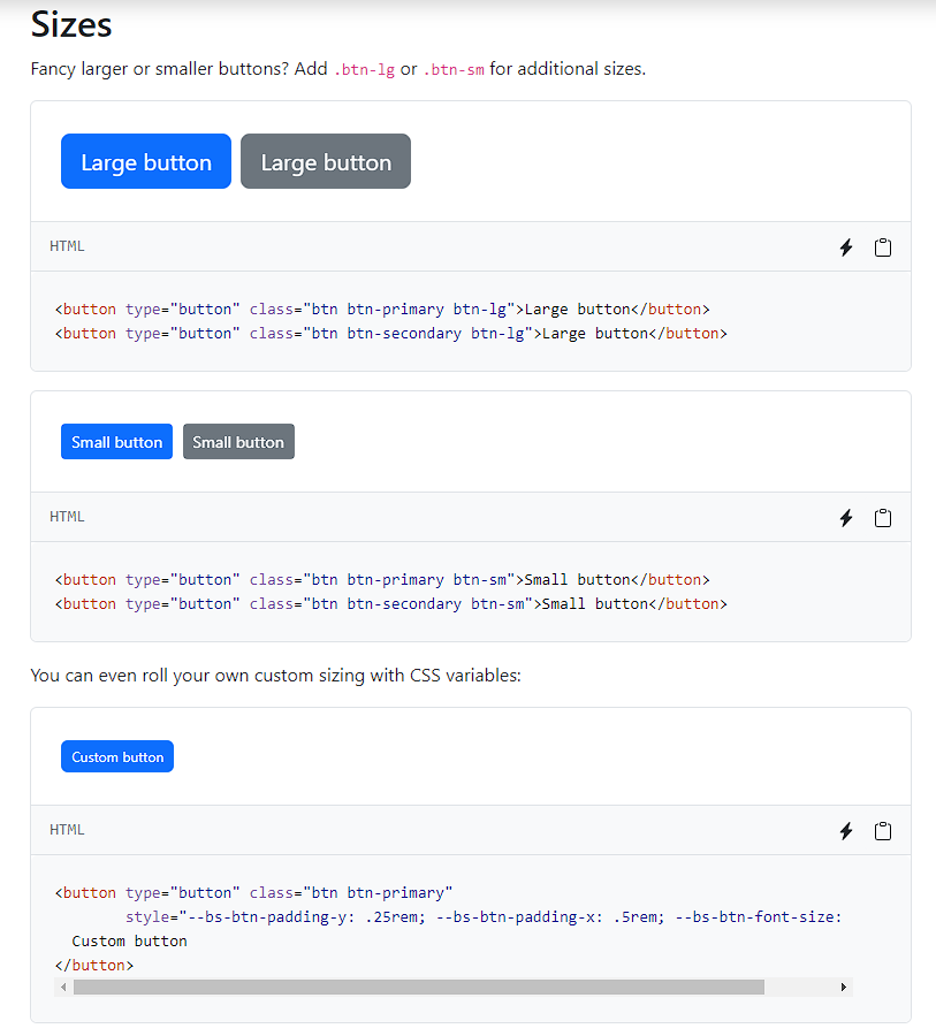
Ø <https://getbootstrap.com/docs/5.2/components/buttons/>

#### **Html Code from Bootstrap**

General Buttons



Button Sizes



*Source for screenshots:* [*https://getbootstrap.com/docs/5.2/components/buttons/*](https://getbootstrap.com/docs/5.2/components/buttons/)

## 

## **Toggle Themes**

### **Sources**

Ø <https://getbootstrap.com/docs/5.3/customize/color-modes/#building-with-sass>

Ø <https://getbootstrap.com/docs/5.3/components/dropdowns/>

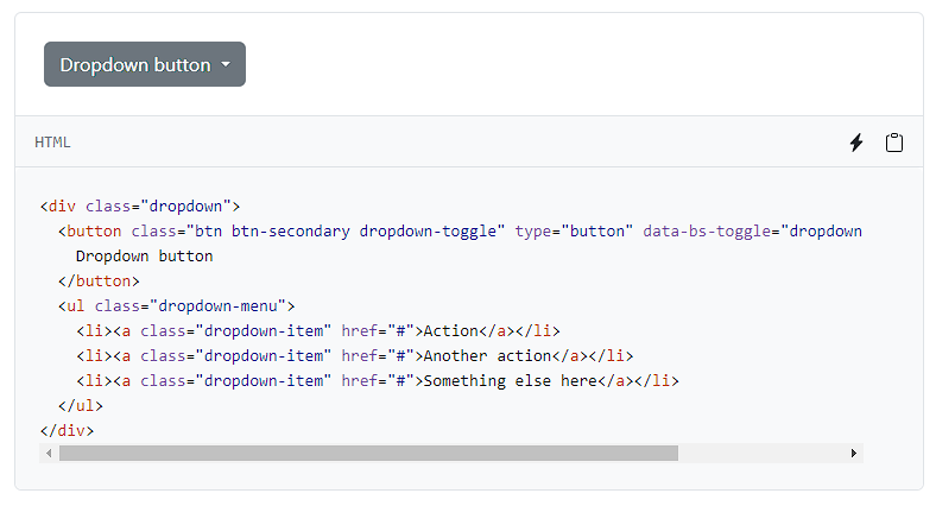
#### **Dark Mode**

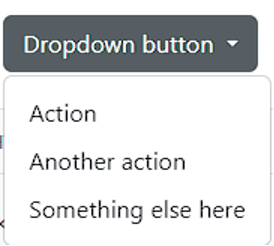
· To enable dark mode throughout the whole project we need to modify the html tag.

*Source from:* [*https://getbootstrap.com/docs/5.3/customize/color-modes/#building-with-sass*](https://getbootstrap.com/docs/5.3/customize/color-modes/#building-with-sass)

#### **Dropdowns**

Dropdown button





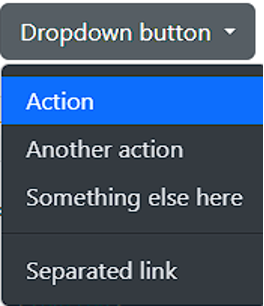
Changing the Dropdown to Dark Mode

Change one line of code from:



To:





End of Emiliano Chavez De La Torre documentation

Kenyou Teoh

**Part 1 and 3, Please note that all pictures are not in here. Please check the file I turned in to see the full version.**

# Part 1: Implementing Generic List and Detail Views

## How do you set up the generic list and detailed views for a model to be displayed in the web app?

### Step 1: Define your model.

Basically, the things in your model.py. At this case, student model.

### Step 2: Create views.

This is about the views.py file. Remember to import the models, I got lazy, so I just import all of them.

With the “model = Student”, this allow Django automatically generate a query to fetch all instances of Student model from the database and pass them to the template.

### Step 3: Create URLs.

I am talking about the urls.py file. Remember to import the models.

This allows people able to request it from the website browser. “students/” are the things that come after the IP. . “views.StudentListView.as\_view()”, this basically coverts the class-based view into function-based view, that means it can be used in the URL patterns.

“name=’students’”, this allows you to refer it in Django without the need to hardcoding the URL itself. You can use it like this . Or with reverse such as

### Step 4: Create Templates.

This is the .html files under templates folder. The following are example of how you can display the models onto the website.

That looks like this:

## What are the CRUD actions and routes you have created so far?

CRUD stands for Create, Read, Update, and Delete – the four basic operations that can be performed on most database systems.

### Create (C):

HTTP Method: POST

Django Route: http://127.0.0.1:8000/admin/portfolio\_app/student/add/ Note: This is pre-created by Django.

This route is used to create a new instance of the model. The form data is typically submitted via a POST request.

### Read (R):

HTTP Method for List View: GET

Django Route for List View: path('students/', views.StudentListView.as\_view(), name= 'students'),

This route is used to retrieve and display a list of instances of the model.

HTTP Method for Detail View: GET

Django Route for Detail View: path('student/<int:pk>', views.StudentDetailView.as\_view(), name='student-detail'),

This route is used to retrieve and display details of a specific instance of the model. The <int:pk> part represents the primary key of the instance. Primary key is a unique identifier for each record (or row) in a table.

### Update (U):

HTTP Method: POST or PUT or PATCH

Django Route: http://127.0.0.1:8000/admin/portfolio\_app/student/1/change/ Note: Pre-created by Django.

This route is used to update an existing instance of the model. The form data is typically submitted via a POST, PUT, or PATCH request.

### Delete (D):

HTTP Method: POST or DELETE

Django Route: http://127.0.0.1:8000/admin/portfolio\_app/student/1/delete/ Note: Pre-created by Django.

This route is used to delete an existing instance of the model.

## How do you pass information from the database to display the active portfolios on the home page?

We will use the following example with explanation:

student\_active\_portfolios = Student.objects.select\_related('portfolio').all().filter(portfolio\_\_is\_active=True)

Student.objects: This part refers to the manager for the student model.

.select\_related('portfolio'): It is used to perform a SQL JOIN and include the related portfolio data in the query.

.all(): This retrieves all objects in the Student model.

.filter(portfolio\_\_is\_active=True): This filters the queryset to include only those Student instances whose associated portfolio has is\_active set to True. The \_\_ (double underscore) is used in Django to traverse relationships between models.

With the above code we are able to filter the portfolio that is active and form a list and pass it to student\_active\_portfolios.

### More information on Django (djangoproject.com)

Matthew Michela

GE05 Documentation: Part 1

**Generic List and Detail Views**

Generic lists and Detail views are both views inherited from Django’s *views.generic* class. They are both examples of **Class-based views.**

**Class-Based Views:**

- Implements Django views as a Python object.

o Django’s common implementation of a View uses function-based views, where the function view receives a request and returns a response.

- Class-based views allow for more code reuse and inheritance from other classes.

- Django provides many generic Class-based views such as: DetailView, ListView, FormView, TemplateView

o  [Django’s Built-in class-based views Documentation](https://docs.djangoproject.com/en/4.2/ref/class-based-views/)

- Creates another level of abstraction by combining a view’s data and behavior for an object-oriented approach.

<https://docs.djangoproject.com/en/4.2/topics/class-based-views/>

Setting up a generic list and detail views for a model to be displayed in the web app:

**ListView**

- Displays a list of objects.

- Can be **paginated**.

-  [ListView Django documentation](https://docs.djangoproject.com/en/4.2/ref/class-based-views/generic-display/#listview)

**DetailView**

- Used to display the details of a single object from a database.

- Needs an object’s unique identifier, usually a **primary key (pk)**.

-  [DetailView Django documentation](https://docs.djangoproject.com/en/4.2/ref/class-based-views/generic-display/#detailview)

**1.** **Setup URL Mapping**

In portfolio\_app’s urls.py:

Include the following within urlpatterns:

path('students/', views.StudentListView.as\_view(), name= 'students'),

path('student/<int:pk>', views.StudentDetailView.as\_view(), name='student-detail'),

path('portfolio/<int:pk>', views.PortfolioDetailView.as\_view(), name='portfolio-detail'),

**2.** **Implement Class-Based Views**

In portfolio\_app’s views.py:

Add python import statements for Generic Views and portfolio\_app’s Student and Portfolio models:

from django.views import generic

from .models import Student, Portfolio

Add the following classes:

class StudentListView(generic.ListView):

model = Student

class StudentDetailView(generic.DetailView):

model = Student

class PortfolioDetailView(generic.DetailView):

model = Portfolio

**3.** **Implement HTML Templates**

Implement HTML Templates for StudentListView, StudentDetailView and PortfolioDetailView from views.py:

Create student\_list.html, student\_detail.html and portfolio\_detail.html files.

Each html file extends base\_template.html with:

{% extends 'portfolio\_app/base\_template.html' %}

Following code is added between {% block content %} and {% endblock %}

For student\_list.html, add the following:

<h1>Student List</h1>

<ul>

{% for student in student\_list %}

<li>

<a href="{{ student.get\_absolute\_url }}">{{ student.name }}</a>

</li>

{% endfor %}

</ul>

{% else %}

<p>There are no students registered.</p>

{% endif %}

o {% if student\_list %}

§ Checks if student\_list is empty.

o <ul>

§ Unordered List

o <a href="{{ student.get\_absolute\_url }}">{{ student.name }}</a>

§ Creates a link to a student's detail page.

o {% for student in student\_list %}

§ Iterates over each student within student\_list

o {{ student.get\_absolute\_url }}

§ Method of the student object to returns the students detail page URL.

For student\_detail.html, add the following:

<h1>Student Name: {{ student.name }}</h1>

<p><strong>Email:</strong> {{ student.email }}</p>

<p><strong>Major:</strong> {{ student.major }}</p>

<p><strong>Portfolio:<a href="{{ student.portfolio.get\_absolute\_url }}">{{ student.portfolio.title }} </a>

{{ student.name }}, {{ student.email }}, {{ student.major }

o For the previous statements, these use Django’s template language to return the data stored within student as text to be displayed.

For student\_list.html, add the following:

<h1>Portfolio Title: {{ portfolio.title }}</h1>

<p><strong>Portfolio Active: </strong> {{ portfolio.is\_active }}</p>

<p><strong>About: </strong> {{ portfolio.about }}</p>

<p><strong>Contact Email: </strong> {{ portfolio.contact\_email }}</p>

CRUD Actions and Routes

**CRUD Paradigm**

*Create, Read, Update, Delete*

· Abstract Concept for creating API’s.

· For building APIs, models need to provide four basic types of functionalities. The model must be able to Create, Read, Update, and Delete resources.

· 4 basic HTTP verbs used in requests to interact with resources in a **REST system** (the implementation of CRUD):

o **Create**: POST

o **Read**: GET

o **Update**: PUT

o **Delete**: DELETE

Route created in previous steps.

**Read**

· Retrieves and displays objects or data from a database.

· **HTTP Verb:** GET

· Implemented by generic Class-Based view, *ListView* through StudentListView

· Implemented by generic Class-Based view, *DetailView* through StudentDetailView and PortfolioDetailView

· **Route:** URL route added to portfolio\_app’s urls.py:

path('students/', views.StudentListView.as\_view(), name= 'students'),

path('student/<int:pk>', views.StudentDetailView.as\_view(), name='student-detail'),

path('portfolio/<int:pk>', views.PortfolioDetailView.as\_view(), name='portfolio-detail'),

Passing information from the database to display active portfolios on the home page.

*Commands to view a QuerySet using Python Shell from this project:*

*python manage.py shell*

*>>> from portfolio\_app.models import Student, Portfolio, Project*

Homepage URL is set to the Function-Based View views.index

path('', views.index, name='index'),

The function-based view “index” is passed the request:

def index(request):

student\_active\_portfolios = Student.objects.select\_related('portfolio').all().filter(portfolio\_\_is\_active=True)

return render(request,'portfolio\_app/index.html', {'student\_active\_portfolios':student\_active\_portfolios})

· student\_active\_portfolios

o Holds the data returned by the QuerySet

· render(request,'portfolio\_app/index.html', {'student\_active\_portfolios':student\_active\_portfolios})

o The render function returns an HTTP Request using 3 pieces of data for this case:

§ The request

§ The HTML template to use ('portfolio\_app/index.html')

§ A dictionary contatining the queryset from student\_active\_portfolios

The template index.html contains a for loop that displays the active portfolios held within the dictionary that it was passed from calling the index function from views.py.

{% for i in student\_active\_portfolios %}

<li class="list-group-item"><strong>Name: </strong>{{i.name}}

<p><strong>Portfolio: </strong>{{ i.portfolio.title }}

<a class="btn btn-primary" href="{{ i.portfolio.get\_absolute\_url }}" role="button">View</a>

</li>

{% endfor %}

Matthew Michela, end of Part 1 Documentation

GE05 Documentation: Matthew Michela

**Deploying on Replit**

Replit allows for easy and real-time collaboration. This allows multiple users to work on the same project simultaneously. This feature is particularly useful for team projects and pair programming. It also allows developers to access and work on their Django projects from anywhere, on any device with an internet connection.

My process and attempt of deploying on Replit:

Setup project by following the steps from<https://blog.replit.com/deploying-django> with a public repository for my project.

- I first uploaded my project but forgot to merge the new branch into the main branch. Made a new Replit project, with the updated main branch.

- Ran into problems with the correct packages not installing. Realized I also forgot to update requirements.txt for the bootstrap package

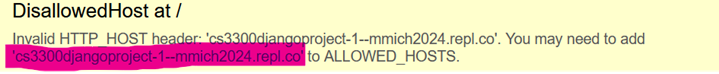
Entered: *pip install -r portfolio/requirements.txt* on Replit, by using their shell tool

- Had to change it to *pip install -r portfolio/requirements.txt*,since my project is not in the root directory

Added: X\_FRAME\_OPTIONS = '\*' to settings.py

The website was able to load after that, although I had to add the name of the new URL it was using to settings.py under ALLOWED\_HOSTS.

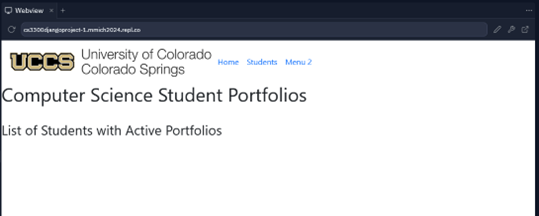
- add URL without ‘https://’



Had to migrate database with *python portfolio/manage.py makemigrations and python portfolio/manage.py migrate*

- No changes detected, but still had to migrate the database.

Website ran successfully after this but with an empty database:



I was able to navigate to the admin panel login page on my own browser, however I could not create a super user on Replit’s shell command window.

- python portfolio/manage.py createsuperuser

Kept running into the error:

