Assignment 5 Walkthrough

SWE 642 Assignment: Full-Stack Student Survey Web Application

May 2, 2025

Submitted by: Hamaad Zuberi (G01413525)

1. Introduction

This document details the design and implementation of a full-stack web application developed to capture and manage student survey responses. The primary goal was to create a user-friendly interface for students to submit survey data and for administrators (implicitly, via the web interface) to view, update, and delete these responses.

The application was built using Python and the Django web framework for both backend logic and frontend template rendering. A MySQL database was utilized for persistent data storage, interfaced via Django's Object-Relational Mapper (ORM). The core features implemented include:

- A welcoming homepage with navigation.
- A comprehensive survey form with various input types (text, checkboxes, radio buttons, dropdown, text area).
- Data validation and storage upon form submission.
- A page listing all submitted survey responses in a tabular format.
- Functionality to update or delete individual survey records directly from the list view.

2. Design Philosophy and Architecture

The application adheres to Django's **Model-View-Template (MVT)** architectural pattern, a variant of the Model-View-Controller (MVC) pattern common in web development. This pattern promotes separation of concerns, leading to a more organized, maintainable, and scalable codebase.

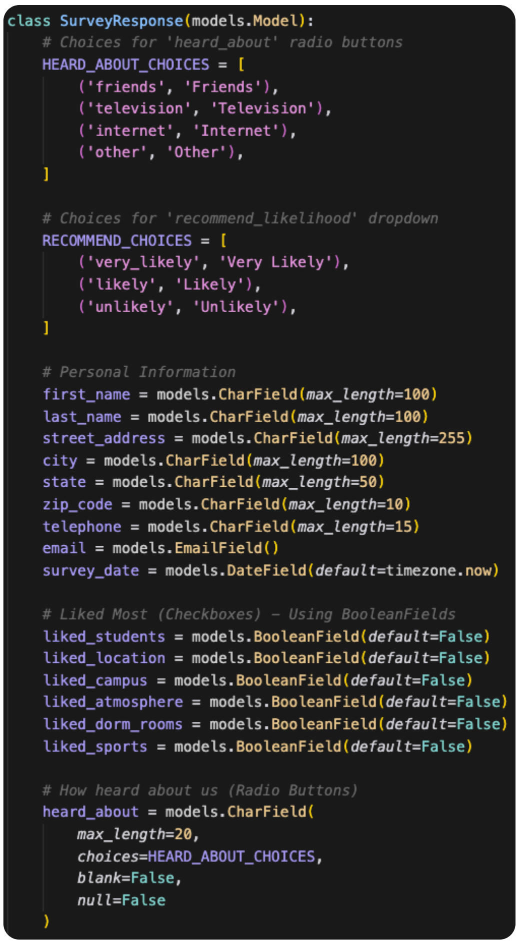
- **Model (`models.py`):** Represents the data structure and interacts with the database. The `SurveyResponse` model defines the schema for the survey data, including field types, constraints, and relationships. Django's ORM handles the translation between Python model objects and MySQL database rows, abstracting away raw SQL interactions.
- **View (`views.py`):** Handles the application's logic. Views receive HTTP requests, process them (e.g., interact with models to fetch or save data, validate form input), and return HTTP responses, typically by rendering a template with appropriate context data. Function-based views were used for simplicity and clarity in this project.
- **Template (`templates/` directory):** Defines the presentation layer (the user interface). Templates are HTML files incorporating Django's template language. They dynamically display data passed from the view and render HTML forms. A base template (`base.html`) was used for common structure (navigation, footer, CSS/JS includes), and other templates (`home.html`, `survey_form.html`, etc.) extend this base. Bootstrap 5 was included via CDN for basic styling and layout.

The project structure consists of a main Django project (`survey`) containing global settings and URL routing, and a Django app (`survey_app`) encapsulating the specific functionality related to the survey (models, views, forms, templates, app-specific URLs). This modular design allows for better organization.

3. Implementation Details

The implementation followed a logical progression, building components layer by layer:

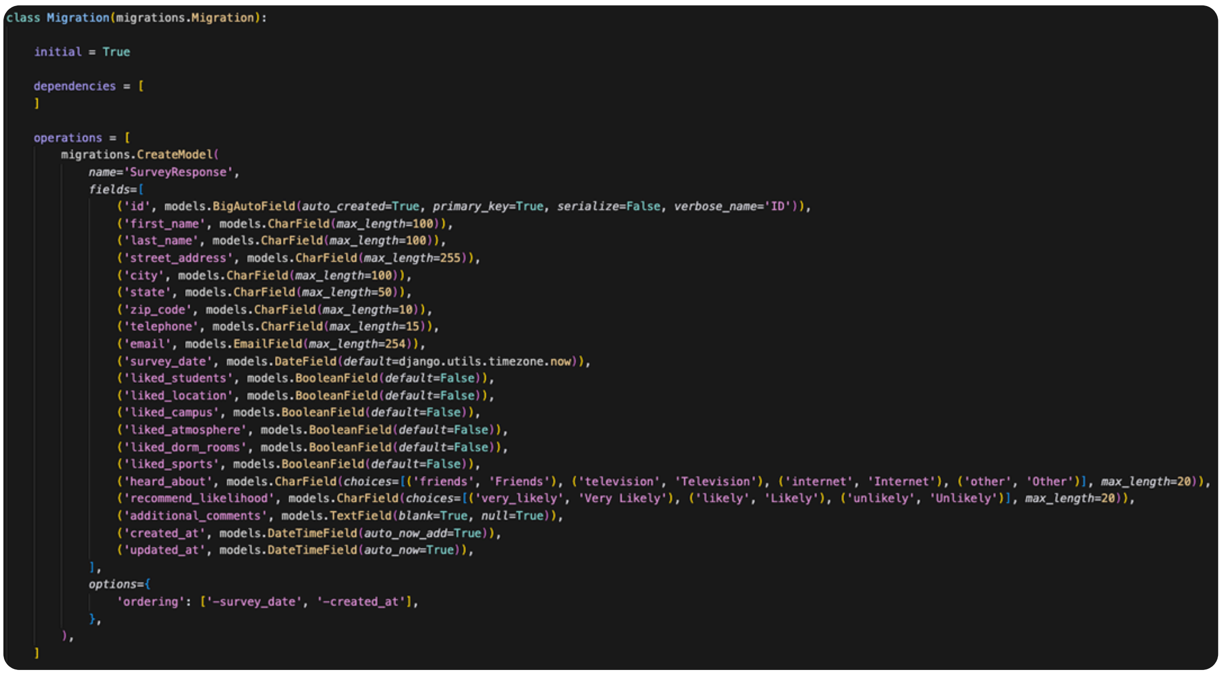
1. **Project Setup & Configuration:**
    - A Python virtual environment (`venv`) was created to manage project dependencies in isolation.
    - Required packages (`Django`, `mysqlclient`) were installed using `pip`. Note: `mysqlclient` required installing development dependencies (`pkg-config`, `mysql`) via Homebrew on the macOS development environment to build successfully.
    - A Django project (`survey`) and an application (`survey_app`) were initialized using `django-admin startproject` and `python manage.py startapp`.
    - The `survey_app` was registered in the `INSTALLED_APPS` list within the project's `survey/settings.py`.
    - The `DATABASES` setting in `survey/settings.py` was configured to connect to the target MySQL database, specifying the engine (`django.db.backends.mysql`), database name, user credentials, host, and port.
2. **Database Modeling (`survey_app/models.py`):**



```python
class SurveyResponse(models.Model):
    # Choices for 'heard_about' radio buttons
    HEARD_ABOUT_CHOICES = [
        ('friends', 'Friends'),
        ('television', 'Television'),
        ('internet', 'Internet'),
        ('other', 'Other'),
    ]

    # Choices for 'recommend_likelihood' dropdown
    RECOMMEND_CHOICES = [
        ('very_likely', 'Very Likely'),
        ('likely', 'Likely'),
        ('unlikely', 'Unlikely'),
    ]

    # Personal Information
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    street_address = models.CharField(max_length=255)
    city = models.CharField(max_length=100)
    state = models.CharField(max_length=50)
    zip_code = models.CharField(max_length=10)
    telephone = models.CharField(max_length=15)
    email = models.EmailField()
    survey_date = models.DateField(default=timezone.now)

    # Liked Most (Checkboxes) - Using BooleanFields
    liked_students = models.BooleanField(default=False)
    liked_location = models.BooleanField(default=False)
    liked_campus = models.BooleanField(default=False)
    liked_atmosphere = models.BooleanField(default=False)
    liked_dorm_rooms = models.BooleanField(default=False)
    liked_sports = models.BooleanField(default=False)

    # How heard about us (Radio Buttons)
    heard_about = models.CharField(
        max_length=20,
        choices=HEARD_ABOUT_CHOICES,
        blank=False,
        null=False
    )
    # Recommendation Likelihood (Dropdown)
    recommend_likelihood = models.CharField(
        max_length=20,
        choices=RECOMMEND_CHOICES,
        blank=False,
        null=False
    )

    # Additional Comments (Text Area)
    additional_comments = models.TextField(blank=True, null=True)

    # Timestamps (Optional but good practice)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.first_name} {self.last_name} - {self.survey_date}"

    class Meta:
        ordering = ['-survey_date', '-created_at']
```

- The `SurveyResponse` model was defined, inheriting from `django.db.models.Model`.
- Fields were created corresponding to the required survey inputs using appropriate Django model field types:
  - `CharField` for text inputs (first name, last name, address details, zip, telephone, state).
  - `EmailField` for the email address (provides basic format validation).
  - `DateField` for the survey date, with `default=timezone.now` to automatically set the date on creation.
  - `BooleanField` (with `default=False`) for each "liked most" checkbox option.
  - `CharField` with a `choices` argument for the 'heard about us' radio buttons and the 'recommend likelihood' dropdown, allowing storage of a short key while displaying user-friendly labels.
  - `TextField` for the multi-line 'additional comments' field (`blank=True, null=True` making it optional).
- Timestamp fields (`created_at`, `updated_at`) were added using `DateTimeField(auto_now_add=True)` and `DateTimeField(auto_now=True)` respectively for automatic tracking.
- A `__str__` method was defined for a user-friendly representation of model instances (e.g., in the admin interface).
- A `Meta` class was added to specify default ordering (`ordering = ['-survey_date', '-created_at']`).

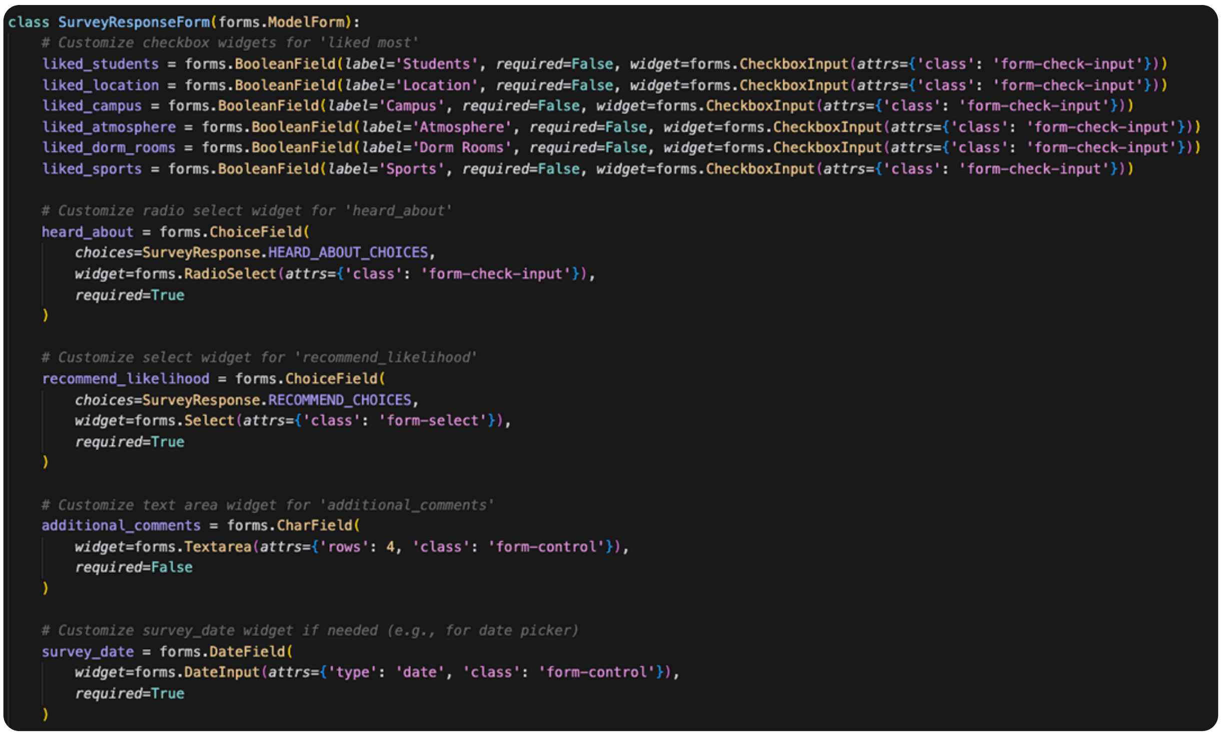
3. **Database Migrations:**



```python
class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='SurveyResponse',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('first_name', models.CharField(max_length=100)),
                ('last_name', models.CharField(max_length=100)),
                ('street_address', models.CharField(max_length=255)),
                ('city', models.CharField(max_length=100)),
                ('state', models.CharField(max_length=50)),
                ('zip_code', models.CharField(max_length=10)),
                ('telephone', models.CharField(max_length=15)),
                ('email', models.EmailField(max_length=254)),
                ('survey_date', models.DateField(default=django.utils.timezone.now)),
                ('liked_students', models.BooleanField(default=False)),
                ('liked_location', models.BooleanField(default=False)),
                ('liked_campus', models.BooleanField(default=False)),
                ('liked_atmosphere', models.BooleanField(default=False)),
                ('liked_dorm_rooms', models.BooleanField(default=False)),
                ('liked_sports', models.BooleanField(default=False)),
                ('heard_about', models.CharField(choices=[('friends', 'Friends'), ('television', 'Television'), ('internet', 'Internet'), ('other', 'Other')], max_length=20)),
                ('recommend_likelihood', models.CharField(choices=[('very_likely', 'Very Likely'), ('likely', 'Likely'), ('unlikely', 'Unlikely')], max_length=20)),
                ('additional_comments', models.TextField(blank=True, null=True)),
                ('created_at', models.DateTimeField(auto_now_add=True)),
                ('updated_at', models.DateTimeField(auto_now=True)),
            ],
            options={
                'ordering': ['-survey_date', '-created_at'],
            },
        ),
    ]
```

- `python manage.py makemigrations survey_app` was run to analyze changes in `models.py` and generate a migration file (`0001_initial.py`) describing the SQL needed to create the `SurveyResponse` table.
- `python manage.py migrate` was executed to apply the migration to the configured MySQL database, creating the `survey_app_surveyresponse` table along with tables for Django's built-in apps (auth, admin, etc.).

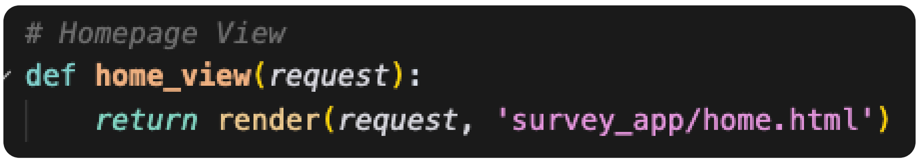
4. **Form Handling (`survey_app/forms.py`):**



```python
class SurveyResponseForm(forms.ModelForm):
    # Customize checkbox widgets for 'liked most'
    liked_students = forms.BooleanField(label='Students', required=False, widget=forms.CheckboxInput(attrs={'class': 'form-check-input'}))
    liked_location = forms.BooleanField(label='Location', required=False, widget=forms.CheckboxInput(attrs={'class': 'form-check-input'}))
    liked_campus = forms.BooleanField(label='Campus', required=False, widget=forms.CheckboxInput(attrs={'class': 'form-check-input'}))
    liked_atmosphere = forms.BooleanField(label='Atmosphere', required=False, widget=forms.CheckboxInput(attrs={'class': 'form-check-input'}))
    liked_dorm_rooms = forms.BooleanField(label='Dorm Rooms', required=False, widget=forms.CheckboxInput(attrs={'class': 'form-check-input'}))
    liked_sports = forms.BooleanField(label='Sports', required=False, widget=forms.CheckboxInput(attrs={'class': 'form-check-input'}))

    # Customize radio select widget for 'heard_about'
    heard_about = forms.ChoiceField(
        choices=SurveyResponse.HEARD_ABOUT_CHOICES,
        widget=forms.RadioSelect(attrs={'class': 'form-check-input'}),
        required=True
    )

    # Customize select widget for 'recommend_likelihood'
    recommend_likelihood = forms.ChoiceField(
        choices=SurveyResponse.RECOMMEND_CHOICES,
        widget=forms.Select(attrs={'class': 'form-select'}),
        required=True
    )

    # Customize text area widget for 'additional_comments'
    additional_comments = forms.CharField(
        widget=forms.Textarea(attrs={'rows': 4, 'class': 'form-control'}),
        required=False
    )

    # Customize survey_date widget if needed (e.g., for date picker)
    survey_date = forms.DateField(
        widget=forms.DateInput(attrs={'type': 'date', 'class': 'form-control'}),
        required=True
    )
    class Meta:
        model = SurveyResponse
        fields = '__all__' # Include all fields from the model
        # Exclude fields managed automatically
        exclude = ['created_at', 'updated_at']

        # Add widget attributes for Bootstrap styling (or other CSS frameworks)
        widgets = {
            'first_name': forms.TextInput(attrs={'class': 'form-control'}),
            'last_name': forms.TextInput(attrs={'class': 'form-control'}),
            'street_address': forms.TextInput(attrs={'class': 'form-control'}),
            'city': forms.TextInput(attrs={'class': 'form-control'}),
            'state': forms.TextInput(attrs={'class': 'form-control'}),
            'zip_code': forms.TextInput(attrs={'class': 'form-control'}),
            'telephone': forms.TextInput(attrs={'class': 'form-control'}),
            'email': forms.EmailInput(attrs={'class': 'form-control'}),
        }

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

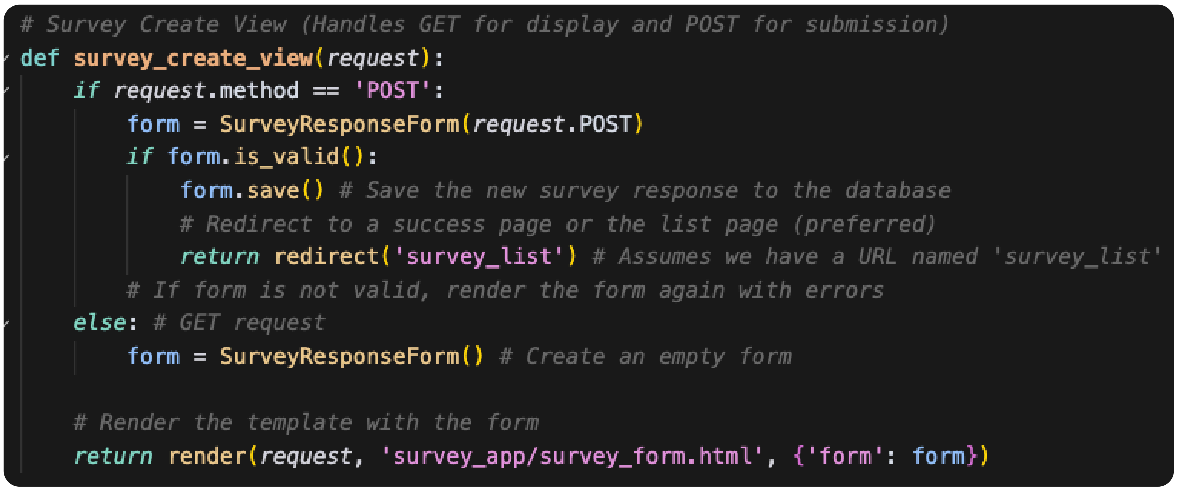
- A `SurveyResponseForm` class was created, inheriting from `django.forms.ModelForm`.
- `ModelForm` automatically introspects the `SurveyResponse` model (specified in the inner `Meta` class) to generate corresponding form fields.
- The `Meta` class specified `model = SurveyResponse` and `fields = '__all__'`, while `exclude = ['created_at', 'updated_at']` prevented form fields from being generated for the automatic timestamp fields.
- Widgets for specific fields were customized to control their HTML rendering and integrate with Bootstrap:
  - `BooleanField` fields used `CheckboxInput`. Explicit `label` arguments were added to override the default labels generated from field names (e.g., `label='Students'` instead of "Liked students").
  - `heard_about` used `RadioSelect`.
  - `recommend_likelihood` used `Select`.
  - `additional_comments` used `Textarea`.
  - `survey_date` used `DateInput(attrs={'type': 'date'})` for a native browser date picker.
  - Standard text inputs used `TextInput` or `EmailInput` with the `form-control` Bootstrap class added via the `Meta.widgets` dictionary or directly in the field definition.
- `ModelForm` provides built-in validation based on model field definitions (e.g., `max_length`, `required`) and handles the `save()` operation to persist data.

5. **View Logic (`survey_app/views.py`):**
   - Function-based views were implemented for each required action:
     - `home_view`: Renders the `home.html` template.



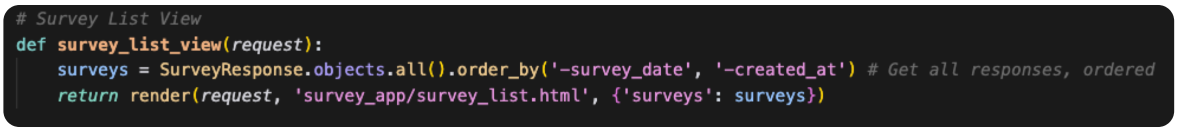
```python
# Homepage View
def home_view(request):
    return render(request, 'survey_app/home.html')
```

- `survey_create_view`: Handles GET requests by creating an empty `SurveyResponseForm` and rendering `survey_form.html`. Handles POST requests by binding `request.POST` data to the form, validating (`form.is_valid()`), saving (`form.save()`) if valid, and redirecting to the list view (`redirect('survey_list')`). If invalid, it re-renders `survey_form.html` with the bound form containing errors.



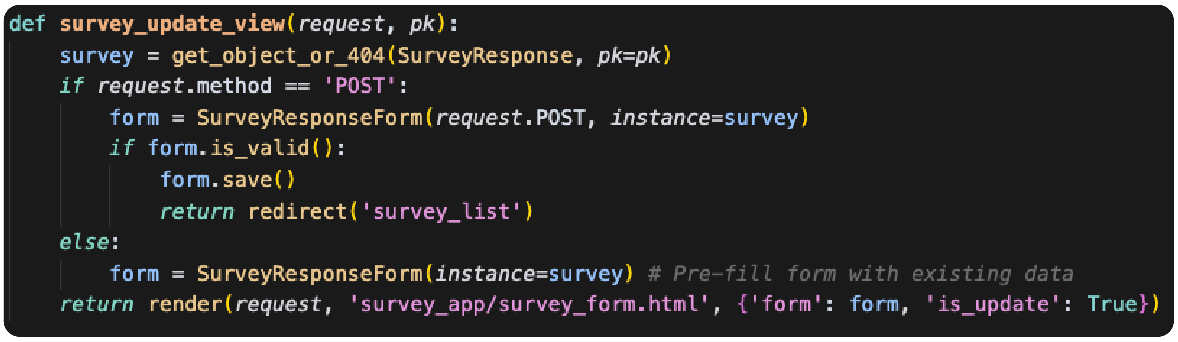
```python
# Survey Create View (Handles GET for display and POST for submission)
def survey_create_view(request):
    if request.method == 'POST':
        form = SurveyResponseForm(request.POST)
        if form.is_valid():
            form.save() # Save the new survey response to the database
            # Redirect to a success page or the list page (preferred)
            return redirect('survey_list') # Assumes we have a URL named 'survey_list'
        # If form is not valid, render the form again with errors
    else: # GET request
        form = SurveyResponseForm() # Create an empty form

    # Render the template with the form
    return render(request, 'survey_app/survey_form.html', {'form': form})
```

- `survey_list_view`: Fetches all `SurveyResponse` objects from the database using the ORM (`SurveyResponse.objects.all().order_by(...)`) and passes the queryset to the `survey_list.html` template for rendering.



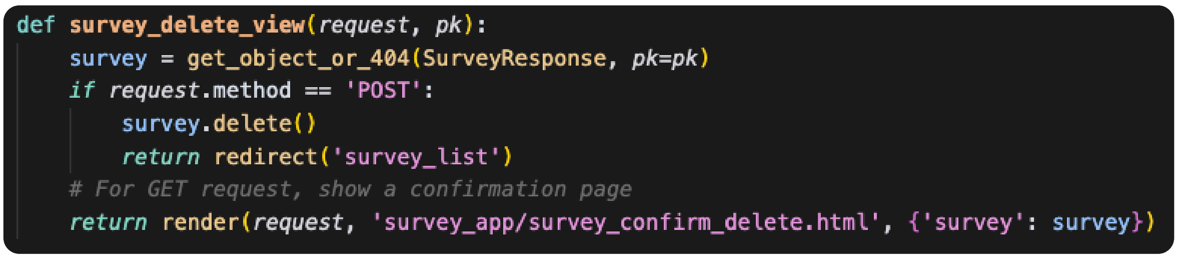
```python
# Survey List View
def survey_list_view(request):
    surveys = SurveyResponse.objects.all().order_by('-survey_date', '-created_at') # Get all responses, ordered
    return render(request, 'survey_app/survey_list.html', {'surveys': surveys})
```

- `survey_update_view`: Handles GET by fetching the specific `SurveyResponse` object (using `get_object_or_404` to handle cases where the ID doesn't exist) and rendering `survey_form.html` with the form pre-populated (`instance=survey`). Handles POST similarly to create, but passes the `instance` to the form constructor so `form.save()` performs an UPDATE instead of an INSERT.



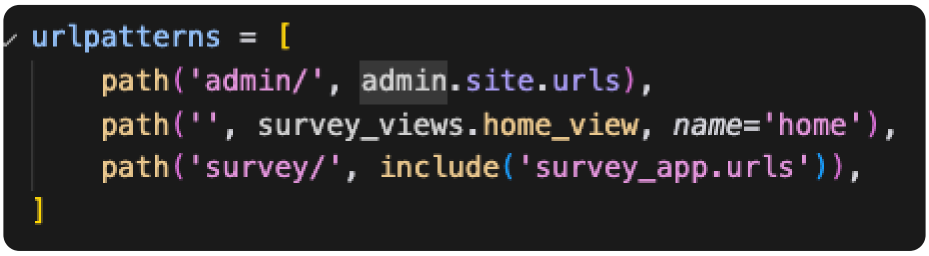
```python
def survey_update_view(request, pk):
    survey = get_object_or_404(SurveyResponse, pk=pk)
    if request.method == 'POST':
        form = SurveyResponseForm(request.POST, instance=survey)
        if form.is_valid():
            form.save()
            return redirect('survey_list')
    else:
        form = SurveyResponseForm(instance=survey) # Pre-fill form with existing data
    return render(request, 'survey_app/survey_form.html', {'form': form, 'is_update': True})
```

- `survey_delete_view`: Handles GET by fetching the object and rendering a confirmation template (`survey_confirm_delete.html`). Handles POST by fetching the object, deleting it (`survey.delete()`), and redirecting to the list view.



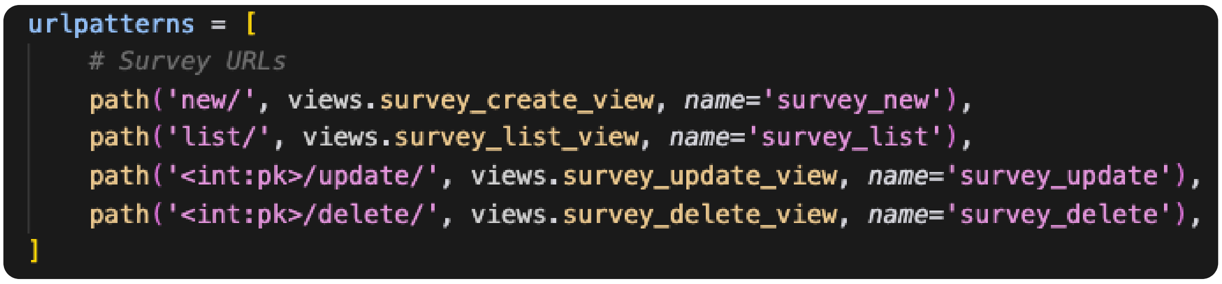
```python
def survey_delete_view(request, pk):
    survey = get_object_or_404(SurveyResponse, pk=pk)
    if request.method == 'POST':
        survey.delete()
        return redirect('survey_list')
    # For GET request, show a confirmation page
    return render(request, 'survey_app/survey_confirm_delete.html', {'survey': survey})
```

6. **URL Routing (`survey/urls.py`, `survey_app/urls.py`):**
   - The main `survey/urls.py` defines project-level URL patterns: `/admin/` for the Django admin site, `/` mapped to `home_view`, and `/survey/` which uses `include('survey_app.urls')` to delegate further routing to the app.



```python
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', survey_views.home_view, name='home'),
    path('survey/', include('survey_app.urls')),
]
```

   - `survey_app/urls.py` defines app-specific patterns relative to `/survey/`: `/new/`, `/list/`, `/<int:pk>/update/`, and `/<int:pk>/delete/`. The `<int:pk>` syntax captures the primary key from the URL and passes it as an argument (`pk`) to the update and delete views. Named URLs (`name='...'`) were used for robust referencing in redirects and templates (`{% url '...' %}`).



```python
urlpatterns = [
    # Survey URLs
    path('new/', views.survey_create_view, name='survey_new'),
    path('list/', views.survey_list_view, name='survey_list'),
    path('<int:pk>/update/', views.survey_update_view, name='survey_update'),
    path('<int:pk>/delete/', views.survey_delete_view, name='survey_delete'),
]
```

7. **Template Rendering (`survey_app/templates/survey_app/`):**
   - HTML templates were created within the `survey_app/templates/survey_app/` directory structure for namespacing.
   - `base.html`: Provided the main HTML structure, included Bootstrap 5 CSS and JS from a CDN, defined navigation, and included `{% block %}` tags (e.g., `{% block content %}`) for content injection by child templates.
   - `home.html`: Extended `base.html` and provided introductory content and links to the survey form and list.
   - `survey_form.html`: Extended `base.html`. Rendered the form passed from the view using Django template tags (`{{ form.as_p }}` or manual field rendering like `{{ form.field_name }}`). Included the essential `{% csrf_token %}` tag for security against Cross-Site Request Forgery. Displayed form errors (`{{ form.field_name.errors }}`, `{{ form.non_field_errors }}`). Used `{% if is_update %}` logic passed from the update view to change button labels and headings dynamically.

- `survey_list.html`: Extended `base.html`. Iterated through the `surveys` queryset passed from the view using `{% for survey in surveys %}`. Displayed survey data in an HTML table. Used Django template filters (`|date:"Y-m-d"`) and model methods (`{{ survey.get_..._display }}`) for formatting choice fields. Included dynamically generated links for Update and Delete using `{% url 'survey_update' survey.pk %}` and `{% url 'survey_delete' survey.pk %}`.
- `survey_confirm_delete.html`: Extended `base.html`. Displayed a confirmation message and included a form with a "Yes, Delete" button that POSTs to the `survey_delete_view`.

4. Key Features Walkthrough

- **Homepage:** Accessed via the root URL (`/`), served by `home_view` rendering `home.html`. Provides navigation links generated using `{% url %}` tags to the "Take Survey" (`/survey/new/`) and "List Surveys" (`/survey/list/`) pages.
- **Create Survey:** Accessed via `/survey/new/`. `survey_create_view` renders `survey_form.html` with an empty form (GET). Upon submission (POST), data is validated by `SurveyResponseForm`. If valid, `form.save()` inserts a new record into the MySQL database via the ORM, and the user is redirected to `/survey/list/`. If invalid, the form is re-rendered with error messages.
- **List Surveys:** Accessed via `/survey/list/`. `survey_list_view` fetches all `SurveyResponse` records using the ORM and passes them to `survey_list.html`, which renders them in a table.
- **Update Survey:** Accessed via `/survey/<pk>/update/` links on the list page. `survey_update_view` fetches the specific record and renders `survey_form.html` pre-filled with its data (GET). Upon submission (POST), data is validated, and `form.save()` updates the existing database record. The user is redirected back to `/survey/list/`.
- **Delete Survey:** Accessed via `/survey/<pk>/delete/` links on the list page. `survey_delete_view` renders `survey_confirm_delete.html` (GET). Upon confirmation via POST, the view deletes the record using `survey.delete()` and redirects to `/survey/list/`.

5. Conclusion

The implemented web application successfully fulfills the specified requirements. It leverages the Django framework and its MVT architecture to provide a well-structured and functional solution for collecting, displaying, and managing student survey data stored in a MySQL database. The use of Django's ORM, forms system, and template engine streamlined the development process and resulted in a maintainable codebase.
