

GNG 5300

Assignment 2

Github link: <https://github.com/locojk/GNG-5300-A2>

300440032

Jun Gao

Oct 4, 2024

1. Project Structure

This project is structured as a typical Django project, containing multiple apps and relevant files for templates, models, views, and authentication. Below is an explanation of each component:

(1) authentication (app)

This app handles user authentication, including login and registration functionalities.

- Key Files:
 - views.py: Contains the logic for login and registration.
 - urls.py: Defines the routes for login and registration.
 - templates/authentication/: Contains templates for login and registration pages:
 - login.html: Template for the user login page.
 - register.html: Template for user registration.
- Key Points:
 - This app is dedicated to handling the authentication of users, ensuring only authenticated users can add, edit, or delete students.

(2) student_management (project root)

- This is the main configuration folder for the Django project, containing settings and URL files.
- Key Files:
 - settings.py: Contains project-wide settings, such as database configuration and installed apps.
 - urls.py: Main URL configuration, which routes requests to the appropriate app.

(3) students (app)

- This app manages all functionality related to student management (CRUD operations).
- Key Files:
 - models.py: Defines the Student model with fields for first name, last name, email, date of birth, enrollment date, and grade.
 - views.py: Contains views for listing, adding, editing, and deleting students.
 - forms.py: Defines the forms used for creating and editing students, with validation for fields like email and grade.

- `urls.py`: Routes for managing students, including views for adding, editing, listing, and deleting students.
- `templates/students/`: Contains HTML templates related to the students app:
 - `student_list.html`: Template for listing all students.
 - `student_form.html`: Template for creating and editing students.
 - `student_detail.html`: Template for viewing detailed information about a student.
 - `student_confirm_delete.html`: Template for confirming the deletion of a student.
- Key Points:
 - The app implements CRUD functionality for students, integrating search, pagination, and user authentication to manage who can add, edit, and delete student records.

(4) templates

- The project contains a general `templates/` folder, which stores common templates like `base.html` that are shared across apps.
- Key Files:
 - `base.html`: Base template used to structure common elements like the navigation bar, ensuring consistent UI across pages.

(5) `db.sqlite3`

This is the SQLite database file, which stores the data for the project during development.

(6) `manage.py`

This is the command-line utility that allows you to interact with the Django project. Commands such as running the server and applying migrations are executed through this file.

2. Project setup

(1) Create a new directory for the project using commands:

```
mkdir GNG-5300-A2
```

```
cd GNG-5300-A2
```

Create a virtual environment to manage dependencies. This ensures that your system Python environment remains untouched and all project dependencies are managed in isolation.

For Windows:

```
python -m venv venv
```

```
.\venv\Scripts\activate
```

For Linux + macOS:

```
python3 -m venv venv
```

```
source venv/bin/activate
```

(2) Install Django and Required Packages

With the virtual environment activated, install Django and any other required dependencies using the pip command.

Run the following command to install Django:

```
python -m pip install Django
```

(3) Create a New Django Project

Now, create the new Django project using the django-admin command. This will set up the basic project structure inside the directory.

Run the following command:

```
django-admin startproject student_management .
```

(4) Set Up Git

After setting up the project, initialize Git in your project directory to manage version control:

Run the following commands:

```
git init
```

```
git add .
```

```
git commit -m "create project"
```

(5) Migrate the Database

Before running the project, you need to apply the initial database migrations. Django uses migrations to set up the database tables according to your models.

Run the following commands:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

(6) Create a Superuser

To access the Django admin panel and manage student records, create a superuser:

Run the following command:

```
python manage.py createsuperuser
```

Follow the prompts to set the username, email, and password.

(7) Run the Server

Finally, start the development server to access the application.

Run the following command:

```
python manage.py runserver
```

Once the server runs, you can access the application by navigating to <http://127.0.0.1:8000/> in your web browser.

2. Student Model Feature

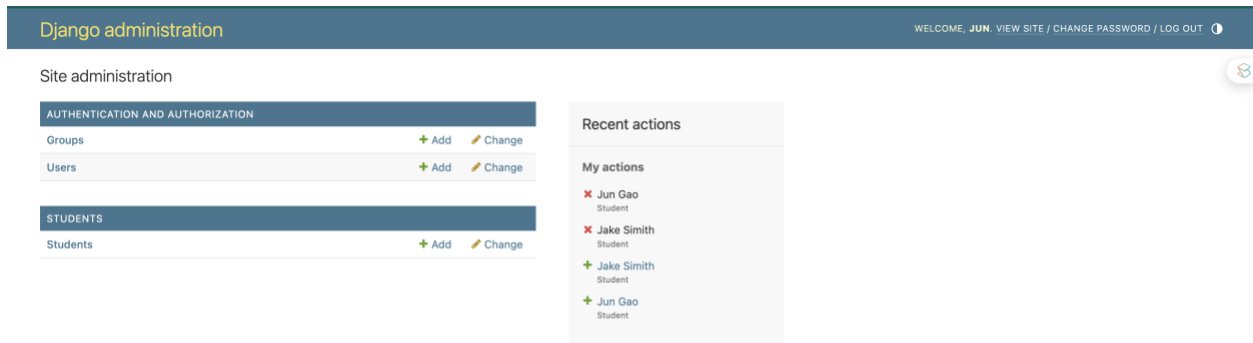
The core feature of this project is the **Student Model**, which represents the information about students. Each student has attributes such as `first_name`, `last_name`, `email`, `date_of_birth`, `enrollment_date`, and `grade`.

This feature was implemented by creating a Django model (`Student`) in the `models.py` file. The model was defined with the appropriate field types such as `CharField` for names, `EmailField` for email validation, and `DateField` for the date of birth and enrollment date. Once the model was defined, migrations were applied to create the necessary database tables.

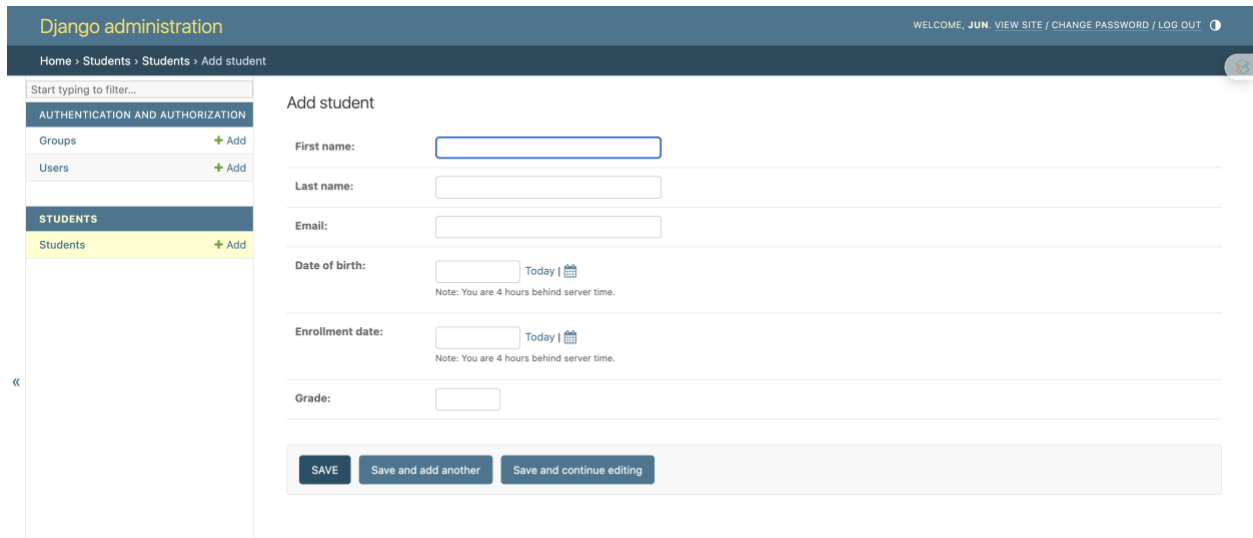
```
students > 🐞 models.py > ...
You, 5 days ago | 1 author (You)
1  from django.db import models
2
You, 5 days ago | 1 author (You)
3  class Student(models.Model):
4      first_name = models.CharField(max_length=30)
5      last_name = models.CharField(max_length=30)
6      email = models.EmailField()
7      date_of_birth = models.DateField()
8      enrollment_date = models.DateField()
9      grade = models.IntegerField()
10
11     def __str__(self):
12         return f"{self.first_name} {self.last_name}"
13
```

2. Admin Interface

The Admin Interface (<http://localhost:8000/admin>) allows administrators to manage student records through the Django admin panel. It provides an easy way to view, create, edit, and delete student records.



Also, you can view the model on the admin page.



3. Views and Templates: Overview

In this project, several views were implemented to handle different operations related to student management. Each view provides a specific feature such as listing students, viewing detailed information about a student, adding new students, editing existing records and login/register

4. Views and Templates: Student List View

The Student List View displays a list of all students in the database. It allows users to view all student records, with pagination and a search bar for easier navigation through large datasets.

The `student_list` view retrieves all student records from the database and passes them to the `student_list.html` template. Pagination was added to limit the number of students displayed per page, and search functionality allows users to filter students by name.

```
# View to list all students (No authentication required for viewing)
def student_list(request):
    query = request.GET.get('q') # Get the search query from the URL (if any)

    if query:
        students = Student.objects.filter(
            first_name__icontains=query
        ) | Student.objects.filter(
            last_name__icontains=query
        )
    else:
        students = Student.objects.all() # Get all students if no search query

    # Paginate the students list - Show 10 students per page
    paginator = Paginator(students, 10) # 10 students per page
    page_number = request.GET.get('page') # Get the current page number
    page_obj = paginator.get_page(page_number) # Get the appropriate page of students

    return render(request, 'students/student_list.html', {'page_obj': page_obj, 'students': page_obj})
```




Student List

Tip: You can search for students by name.

[Sarah Martinez](#)[Jane Miller](#)[David Smith](#)[David Johnson](#)[Sarah Johnson](#)[Emily Rodriguez](#)[Jane Williams](#)[Laura Jones](#)[Robert Johnson](#)[Emily Miller](#)[Page 1 of 5](#)[Next](#)[Last](#)

5. Views and Templates: Student Detail View

The Student Detail View shows detailed information about a single student. Users can view the student's full details, such as first name, last name, email, date of birth, enrollment date, and grade.

The `student_detail` view fetches the student by its primary key (`pk`) and displays the detailed information in the `student_detail.html` template. If the student does not exist, a 404 error is raised using `get_object_or_404`.

```
# View to display a single student's details (No authentication required for viewing)
def student_detail(request, pk):
    student = get_object_or_404(Student, pk=pk)
    return render(request, 'students/student_detail.html', {'student': student})
```



Sarah Martinez

Email: sarah.martinez_0@example.com

Date of Birth: Jan. 1, 1995

Enrollment Date: Sept. 1, 2022

Grade: 10

[Edit Student](#)[Back to student list](#)[Delete Student](#)

6. Views and Templates: Add Student View

The Add Student View allows users to add a new student to the system by filling out a form with the student's information.

The `student_create` view handles both GET and POST requests. On a GET request, it displays a blank form for the user to input student information. On a POST request, it validates the data and, if valid, saves the new student to the database.

```
@login_required(login_url='login')
def student_create(request):
    if request.method == 'POST':
        form = StudentForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('student_list')
    else:
        form = StudentForm()
    return render(request, 'students/student_form.html', {'form': form})
```



Add Student

First name

Last name

Email

Date of birth

Enrollment date

Grade

[Save](#) [Back to student list](#)


7. Views and Templates: Edit Student View

The Edit Student View allows users to update information for an existing student. It displays the current student data in a form, which the user can modify and submit.

The `student_edit` view also handles both GET and POST requests. When accessed via GET, it loads the current student's data into the form. On a POST request, it validates the updated data and saves the changes to the database.

```
@login_required(login_url='/login/')
def student_edit(request, pk):
    student = get_object_or_404(Student, pk=pk)
    if request.method == 'POST':
        form = StudentForm(request.POST, instance=student)
        if form.is_valid():
            form.save()
            return redirect('student_list')
    else:
        form = StudentForm(instance=student)
    return render(request, 'students/student_form.html', {'form': form, 'student': student})
```

Home Logout



Edit Student

First name

Last name

Email

Date of birth

Enrollment date

Grade

8. Views and Templates: Delete Student View

The Delete Student View allows users to delete a student record from the system after confirmation.

The `student_delete` view fetches the student to be deleted and, on confirmation, deletes the record from the database. The user is redirected to the student list view after the deletion.

```
@login_required(login_url='login')
def student_delete(request, pk):
    student = get_object_or_404(Student, pk=pk)
    if request.method == 'POST':
        student.delete()
        return redirect('student_list')
    return render(request, 'students/student_confirm_delete.html', {'student': student})
```

The `student_confirm_delete.html` template displays a confirmation message asking the user to confirm the deletion.

8. Forms

Forms are used for adding and updating student information. These forms ensure that data is correctly input by users, including validation for the email field and ensuring that the grade is between 1 and 12.

Django's ModelForm was used to create a form based on the Student model. Custom validation logic was added to check if the grade falls within the acceptable range and ensure the email field contains a valid email address.

```
You, 5 days ago | 1 author (You)
class StudentForm(forms.ModelForm):
    You, 5 days ago | 1 author (You)
    class Meta:
        model = Student
        fields = ['first_name', 'last_name', 'email', 'date_of_birth', 'enrollment_date', 'grade']
        widgets = {
            'date_of_birth': forms.DateInput(attrs={'type': 'date'}),
            'enrollment_date': forms.DateInput(attrs={'type': 'date'}),
        }

    # Custom validation for email
    def clean_email(self):
        email = self.cleaned_data.get('email')
        if not email or '@' not in email:
            raise forms.ValidationError("Please enter a valid email address.")
        return email

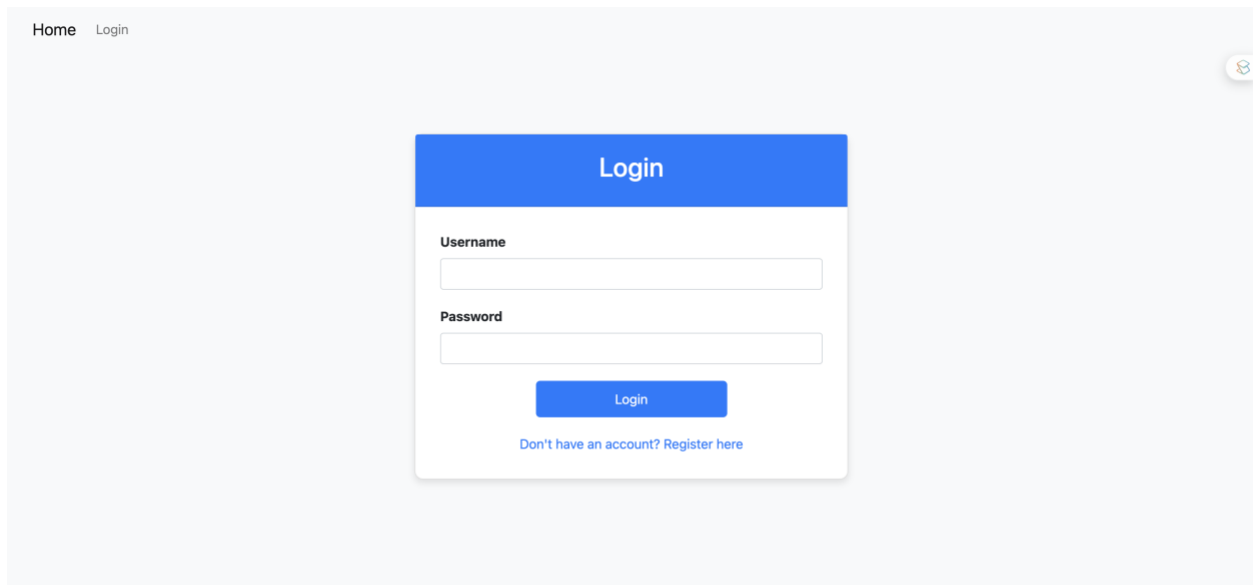
    # Custom validation for grade (between 1 and 12)
    def clean_grade(self):
        grade = self.cleaned_data.get('grade')
        if grade < 1 or grade > 12:
            raise forms.ValidationError("Grade must be between 1 and 12.")
        return grade
```

9. Authentication

The Authentication feature ensures that only authenticated users can access certain parts of the system, such as adding, editing, or deleting student records. This is done through two main functionalities: Login and Register. Users need to create an account (register) and log in to perform any of the restricted actions. Users who are not logged in will be prompted to log in when trying to access these features.

The **Login** feature allows registered users to log into the system. Once logged in, they can access restricted functions, such as adding, editing, or deleting student records.

The login template allows users to input their username and password, and it displays validation errors if the login fails.



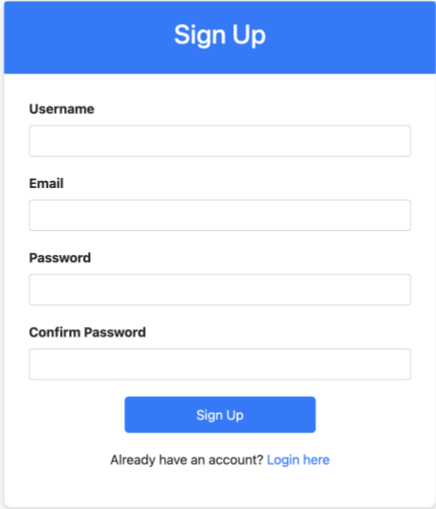
The **Register** feature allows new users to create an account in the system. Once registered, they can log in and perform actions like creating and managing student records.

The registration feature was implemented by creating a custom view that handles user creation. A registration form was built using Django's `UserCreationForm`, and a custom template (`register.html`) was provided for users to sign up.

```
# View for user registration
def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            return redirect('student_list')
    else:
        form = UserCreationForm()

    return render(request, 'authentication/register.html', {'form': form})
```

Home Login



The image shows a web page with a light blue background. At the top left, there are links for 'Home' and 'Login'. At the top right, there is a small circular icon with a blue and orange design. In the center, there is a white sign-up form with a blue header that says 'Sign Up'. The form contains four input fields: 'Username', 'Email', 'Password', and 'Confirm Password'. Below these fields is a blue button labeled 'Sign Up'. At the bottom of the form, there is a link that says 'Already have an account? Login here'.

10. Search Functionality and Pagination

A search bar is provided on the student list page, allowing users to search for students by name. This feature helps users quickly locate specific student records.

The search functionality was added by modifying the `student_list` view. The view checks for a query parameter in the GET request and filters the student list based on the query. A search form was added to the template to allow user input.

Pagination limits the number of students displayed per page, making it easier to navigate large lists of students.

Django's Paginator class was used to implement pagination. The `student_list` view was modified to break up the student list into pages, with a set number of students displayed on each page.

```
# View to list all students (No authentication required for viewing)
def student_list(request):
    query = request.GET.get('q') # Get the search query from the URL (if any)

    if query:
        students = Student.objects.filter(
            first_name__icontains=query
        ) | Student.objects.filter(
            last_name__icontains=query
        )
    else:
        students = Student.objects.all() # Get all students if no search query

    # Paginate the students list - Show 10 students per page
    paginator = Paginator(students, 10) # 10 students per page
    page_number = request.GET.get('page') # Get the current page number
    page_obj = paginator.get_page(page_number) # Get the appropriate page of students

    return render(request, 'students/student_list.html', {'page_obj': page_obj, 'students': page_obj})
```

Student List

Tip: You can search for students by name.

Sarah Martinez

Emily Miller

Page 1 of 5

Next

Last

Add a new student

11. Error Handling and Validation

Validation was applied in the StudentForm, ensuring that the email is valid, the grade falls between a certain range, and the date input is selected using a calendar picker for consistency and ease of use.

The email field ensures that a properly formatted email address is entered. Django's built-in EmailField automatically validates that the input is a valid email format.

To ensure that a valid **grade** is entered, custom validation was added to the clean_grade() method. The grade must be between 1 and 12, and an error is raised if the input is outside this range.

For **date of birth** and **enrollment date** fields, a calendar picker was added to the form to improve user input consistency and prevent invalid date formats.

```
You, 5 days ago | 1 author (You)
class StudentForm(forms.ModelForm):
    You, 5 days ago | 1 author (You)
    class Meta:
        model = Student
        fields = ['first_name', 'last_name', 'email', 'date_of_birth', 'enrollment_date', 'grade']
        widgets = {
            'date_of_birth': forms.DateInput(attrs={'type': 'date'}),
            'enrollment_date': forms.DateInput(attrs={'type': 'date'}),
        }

    # Custom validation for email
    def clean_email(self):
        email = self.cleaned_data.get('email')
        if not email or '@' not in email:
            raise forms.ValidationError("Please enter a valid email address.")
        return email

    # Custom validation for grade (between 1 and 12)
    def clean_grade(self):
        grade = self.cleaned_data.get('grade')
        if grade < 1 or grade > 12:
            raise forms.ValidationError("Grade must be between 1 and 12.")
        return grade
```

If no students are present in the database, a user-friendly message is displayed in the **Student List** view. Instead of showing an empty list, the template checks if any students exist and displays a message if none are found.

In the `student_list.html` template:

```
<!-- Student List -->
<ul class="list-group mb-4">
  {% if students %}
    {% for student in students %}
      <li class="list-group-item">
        <a href="{% url 'student_detail' student.pk %}">{{ student.first_name }} {{ student.last_name }}</a>
      </li>
    {% endfor %}
  {% else %}
    <li class="list-group-item">No students found matching.</li>
  {% endif %}

```

12. Error Handling and Validation

Throughout the development of this project, a few challenges arose that required careful attention and problem-solving. Below are the key challenges encountered during the implementation of the Student Management System:

Ensuring that only authenticated users could add, edit, or delete student records was crucial to maintaining security in the system. However, there were challenges in implementing Django's authentication system to work seamlessly with the custom views. For example, when an unauthenticated user attempted to access restricted pages, setting up proper redirection to the login page while preserving the original request path required careful configuration.

Implementing pagination and search functionality on the student list page was somewhat tricky, especially when combining both features. Making sure that pagination worked even when filtering results via the search query required managing URL parameters and handling pagination logic separately for each search request. Fine-tuning this functionality was necessary to ensure a smooth user experience when navigating through large datasets.