

Report on the Student Management System Project

1. Introduction

The **Student Management System** is a web-based application developed using the Django framework. It allows administrators to manage student information, assign grades, and handle other academic data efficiently.

This report outlines the setup process, challenges encountered during development, and how each feature was implemented. Key code snippets and screenshots are included to illustrate important aspects of the project.

2. Setup Process

2.1 Clone the Repository

```
git clone https://github.com/Mingyang096/StudentManagementSystem.git
```

```
cd StudentManagementSystem
```

2.2 Create a Virtual Environment

```
python -m venv student_env
```

```
source student_env/bin/activate
```

2.3 Install Dependencies

```
pip install -r requirements.txt
```

2.4 Apply Migrations

```
python manage.py migrate
```

2.5 Create a Superuser

```
python3 manage.py createsuperuser
```

2.6 Run the Development Server

```
python manage.py runserver
```

2.7 Access the Application

Open your web browser and navigate to:

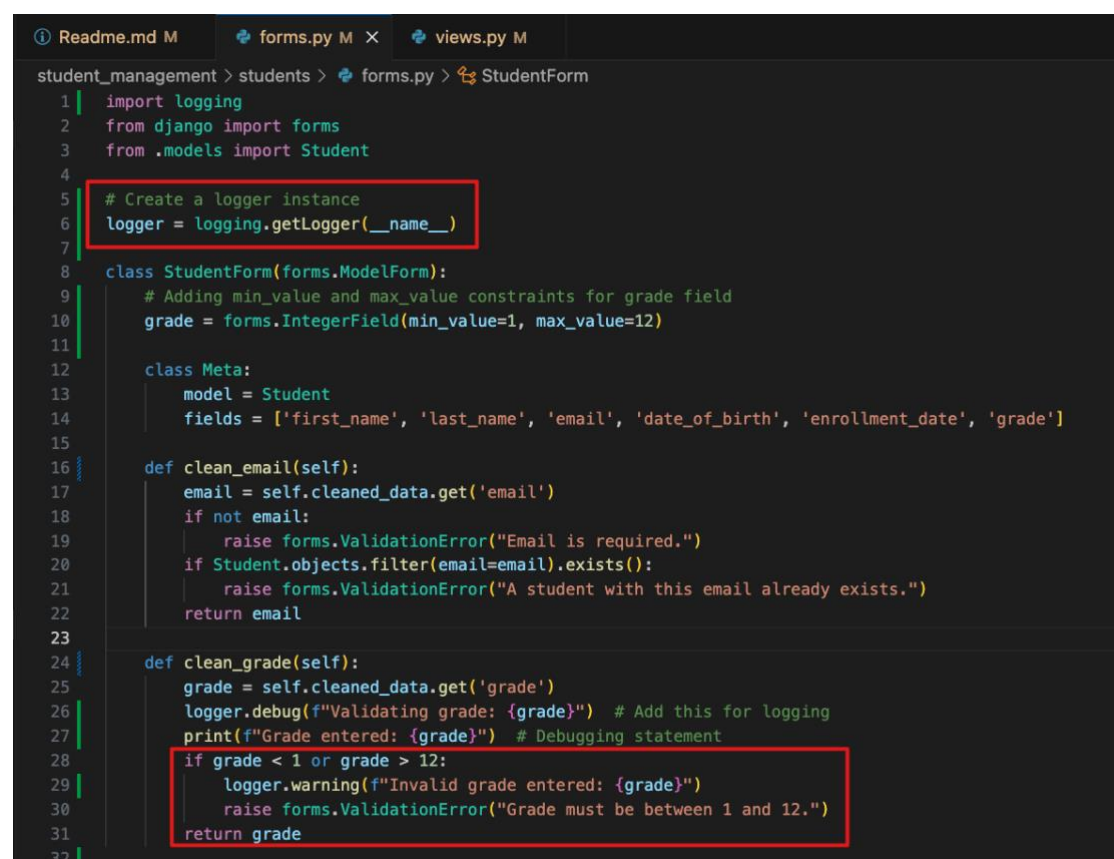
<http://127.0.0.1:8000/admin/>

Use your admin credentials to log in. Then you can access the Django admin panel to manage students and other models.

3. Challenges Encountered

The biggest challenge I met was the input validation for grades. According to the requirement, the grade should be between 1 and 12. However, after I developed the code for grade input validation in `student_management/students/forms.py`, it didn't work. Users could still input numbers larger than 12 and save the forms.

To deal with this, I added a lot of prints to see what happens. Then I found out there was no output in my console. Then I added log settings to save the log to a log file. I get the conclusion: my grade input validation did not work.



```
student_management > students > forms.py > StudentForm
1 | import logging
2 | from django import forms
3 | from .models import Student
4 |
5 | # Create a logger instance
6 | logger = logging.getLogger(__name__)
7 |
8 | class StudentForm(forms.ModelForm):
9 |     # Adding min_value and max_value constraints for grade field
10 |    grade = forms.IntegerField(min_value=1, max_value=12)
11 |
12 |    class Meta:
13 |        model = Student
14 |        fields = ['first_name', 'last_name', 'email', 'date_of_birth', 'enrollment_date', 'grade']
15 |
16 |    def clean_email(self):
17 |        email = self.cleaned_data.get('email')
18 |        if not email:
19 |            raise forms.ValidationError("Email is required.")
20 |        if Student.objects.filter(email=email).exists():
21 |            raise forms.ValidationError("A student with this email already exists.")
22 |        return email
23 |
24 |    def clean_grade(self):
25 |        grade = self.cleaned_data.get('grade')
26 |        logger.debug(f"Validating grade: {grade}") # Add this for logging
27 |        print(f"Grade entered: {grade}") # Debugging statement
28 |        if grade < 1 or grade > 12:
29 |            logger.warning(f"Invalid grade entered: {grade}")
30 |            raise forms.ValidationError("Grade must be between 1 and 12.")
31 |        return grade
32 |
```

Then I searched for the internet to learn how to solve this problem. Till I found the `MinValueValidator` and `MaxValueValidator` classes. I added them to the `students/models.py` file so when the Grade field was set; it was restricted between 1 and 12. Finally, it worked.

```
Readme.md M  forms.py M  models.py M X  views.py M
student_management > students > models.py > ...
1  from django.db import models
2  from django.core.validators import MinValueValidator, MaxValueValidator
3
4  class Student(models.Model):
5      first_name = models.CharField(max_length=50)
6      last_name = models.CharField(max_length=50)
7      email = models.EmailField(unique=True)
8      date_of_birth = models.DateField()
9      enrollment_date = models.DateField()
10     grade = models.IntegerField(
11         validators=[MinValueValidator(1), MaxValueValidator(12)],
12         verbose_name='grade'
13     )
14     def __str__(self):
15         return f"{self.first_name} {self.last_name}"
16
```

4. Features Implementation

4.1 Student Management

Student Management implemented Create, Read, Update, and Delete functionalities for managing student records.

Models: Defined a Student model in models.py with fields such as first_name, last_name, grade, etc.

```
Readme.md M  forms.py M  models.py M X  views.py M
student_management > students > models.py > ...
1  from django.db import models
2  from django.core.validators import MinValueValidator, MaxValueValidator
3
4  class Student(models.Model):
5      first_name = models.CharField(max_length=50)
6      last_name = models.CharField(max_length=50)
7      email = models.EmailField(unique=True)
8      date_of_birth = models.DateField()
9      enrollment_date = models.DateField()
10     grade = models.IntegerField(
11         validators=[MinValueValidator(1), MaxValueValidator(12)],
12         verbose_name='grade'
13     )
14     def __str__(self):
15         return f"{self.first_name} {self.last_name}"
```

Views: Created views to handle each CRUD operation.

```
student_management > students > views.py > student_add
1  from django.shortcuts import render, get_object_or_404, redirect
2  from django.contrib.auth.decorators import login_required
3  from django.core.paginator import Paginator
4  from .models import Student
5  from .forms import StudentForm
6  from django.contrib import messages
7
8
9  > def student_list(request): ...
22
23  > def student_detail(request, pk): ...
26
27  @login_required
28  > def student_add(request): ...
43
44  @login_required
45  > def student_edit(request, pk): ...
59
60  @login_required
61  def student_delete(request, pk):
62      student = get_object_or_404(Student, pk=pk)
63      if request.method == 'POST':
64          student.delete()
65          return redirect('student_list')
66      return render(request, 'students/student_confirm_delete.html', {'student': student})
67
```

Templates: Designed HTML templates for listing students, displaying details, and forms for adding/editing students.

4.2 Admin Interface

Utilized Django's built-in admin interface to manage student data efficiently. The admin panel provides a user-friendly interface for administrators to perform management tasks without requiring custom front-end development for these functions.

Admin Registration: Registered the Student model in admin.py to make it accessible through the admin interface.

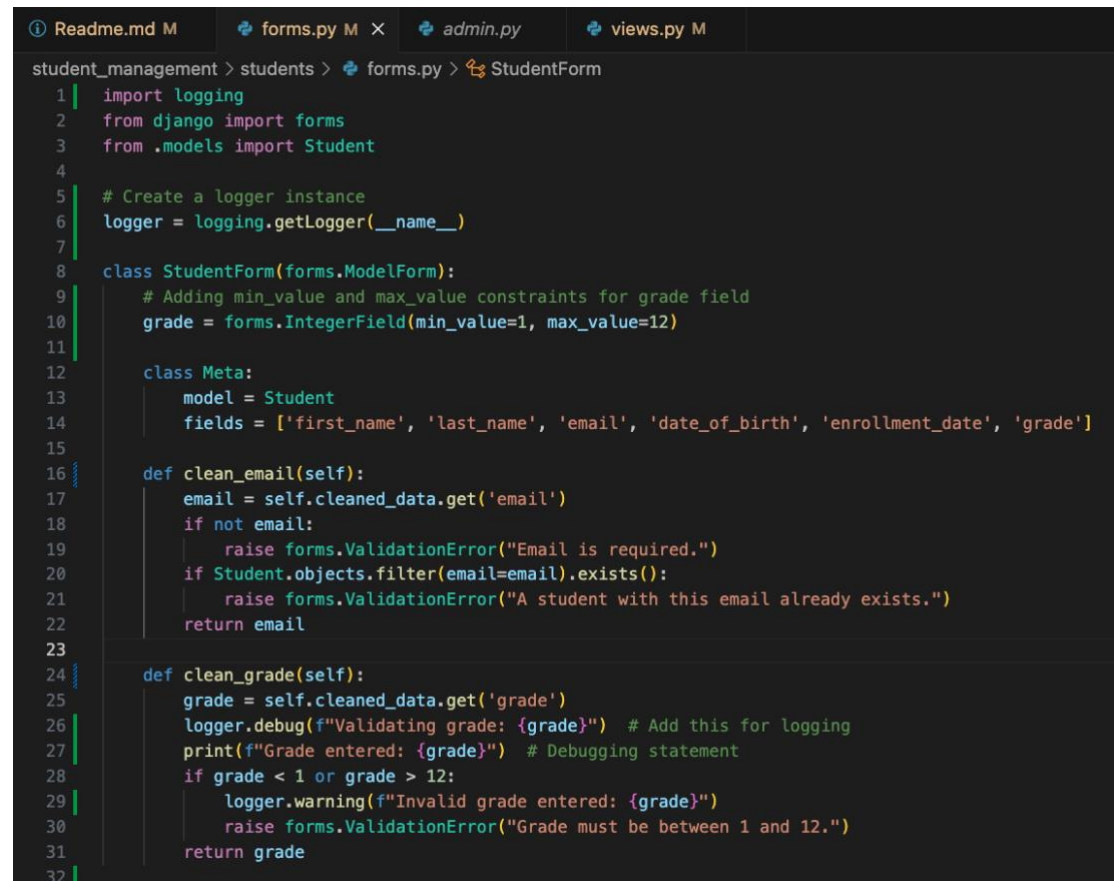
```
student_management > students > admin.py > StudentAdmin
1  from django.contrib import admin
2  from .models import Student
3
4  @admin.register(Student)
5  class StudentAdmin(admin.ModelAdmin):
6      list_display = ('first_name', 'last_name', 'enrollment_date')
```

4.3 Forms

Implemented forms to handle user input for creating and updating student records. Forms ensure that data is validated before being processed and saved to the database.

Forms Module: Created a StudentForm in forms.py based on the Student model.

Validation: Leveraged model validators defined in the Student model to ensure data integrity.

A screenshot of a code editor with a dark theme. The editor has tabs at the top for 'Readme.md M', 'forms.py M X', 'admin.py', and 'views.py M'. The active tab is 'forms.py M X', showing the code for the StudentForm class. The code is as follows:

```
student_management > students > forms.py > StudentForm
1 | import logging
2 | from django import forms
3 | from .models import Student
4 |
5 | # Create a logger instance
6 | logger = logging.getLogger(__name__)
7 |
8 | class StudentForm(forms.ModelForm):
9 |     # Adding min_value and max_value constraints for grade field
10 |    grade = forms.IntegerField(min_value=1, max_value=12)
11 |
12 |    class Meta:
13 |        model = Student
14 |        fields = ['first_name', 'last_name', 'email', 'date_of_birth', 'enrollment_date', 'grade']
15 |
16 |    def clean_email(self):
17 |        email = self.cleaned_data.get('email')
18 |        if not email:
19 |            raise forms.ValidationError("Email is required.")
20 |        if Student.objects.filter(email=email).exists():
21 |            raise forms.ValidationError("A student with this email already exists.")
22 |        return email
23 |
24 |    def clean_grade(self):
25 |        grade = self.cleaned_data.get('grade')
26 |        logger.debug(f"Validating grade: {grade}") # Add this for logging
27 |        print(f"Grade entered: {grade}") # Debugging statement
28 |        if grade < 1 or grade > 12:
29 |            logger.warning(f"Invalid grade entered: {grade}")
30 |            raise forms.ValidationError("Grade must be between 1 and 12.")
31 |        return grade
32 |
```

Templates: Used Django's template language to render forms in HTML templates.

4.4 Authentication

Secured the application by implementing user authentication, ensuring that only authorized users (administrators) can access certain views and perform management tasks.

Django Authentication: Used Django's built-in authentication system.

Login Required Decorator: Applied @login_required decorators to views that require authentication.

```
Readme.md M  forms.py M  student_form.html M  views.py M x
student_management > students > views.py > student_add
1  from django.shortcuts import render, get_object_or_404, redirect
2  from django.contrib.auth.decorators import login_required
3  from django.core.paginator import Paginator
4  from .models import Student
5  from .forms import StudentForm
6  from django.contrib import messages
7
8
9  > def student_list(request): ...
22
23  > def student_detail(request, pk): ...
26
27  @login_required
28  > def student_add(request): ...
43
44  @login_required
45  > def student_edit(request, pk): ...
59
60  @login_required
61  def student_delete(request, pk):
62      student = get_object_or_404(Student, pk=pk)
63      if request.method == 'POST':
64          student.delete()
65          return redirect('student_list')
66      return render(request, 'students/student_confirm_delete.html', {'student': student})
67
```

4.5 Search Functionality

Added the ability for users to search for students based on certain criteria, such as first name, last name, or grade.

Search Form: Included a search form in the template.

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

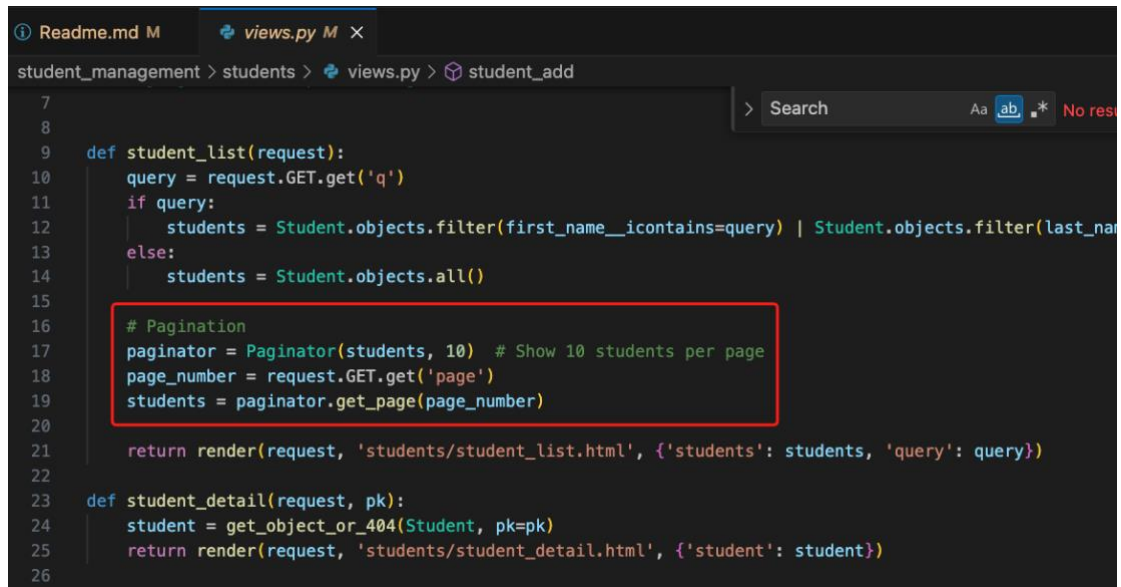
{% block content %}
    <h1>Student List</h1>
    <form method="GET">
        <input type="text" name="q" placeholder="Search by name..." value="{{ query }}">
        <button type="submit">Search</button>
    </form>
    <a href="{% url 'student_add' %}">Add New Student</a>
    <ul>
        {% for student in students %}

```

4.6 Pagination

Implemented pagination to manage the display of large numbers of student records by dividing them into pages.

Paginator: Used Django's Paginator class in the view.



```
7
8
9 def student_list(request):
10     query = request.GET.get('q')
11     if query:
12         students = Student.objects.filter(first_name__icontains=query) | Student.objects.filter(last_name__icontains=query)
13     else:
14         students = Student.objects.all()
15
16     # Pagination
17     paginator = Paginator(students, 10) # Show 10 students per page
18     page_number = request.GET.get('page')
19     students = paginator.get_page(page_number)
20
21     return render(request, 'students/student_list.html', {'students': students, 'query': query})
22
23 def student_detail(request, pk):
24     student = get_object_or_404(Student, pk=pk)
25     return render(request, 'students/student_detail.html', {'student': student})
26
```

4.7 Error Handling and Validation

Ensured that the application gracefully handles errors and validates user input to maintain data integrity and provide a better user experience.

Model Validators: Added validators to the Student model fields.

Form Validation: Relied on model validators in forms to automatically enforce validation rules.

Error Messages in Templates: Displayed validation errors in the templates.

Readme.md M student_form.html M X

student_management > students > templates > students > student_form.html > form > ul

```

1  {% extends 'base.html' %}
2
3  {% block content %}
4      <h1>{% if form.instance.pk %}Edit Student{% else %}Add Student{% endif %}</h1>
5      <form method="POST">
6          {% csrf_token %}
7
8          <!-- Display general messages (success or error) -->
9          {% if messages %}
10             <ul>
11                 {% for message in messages %}
12                     <li class="alert {{ message.tags }}">{{ message }}</li>
13                 {% endfor %}
14             </ul>
15          {% endif %}
16
17          <!-- Display form errors if any -->
18          {% if form.errors %}
19              <ul class="error-messages">
20                  {% for field in form %}
21                      {% for error in field.errors %}
22                          <li>{{ field.label }}: {{ error }}</li>
23                      {% endfor %}
24                  {% endfor %}
25                  {% for error in form.non_field_errors %}
26                      <li>{{ error }}</li>
27                  {% endfor %}
28              </ul>
29          {% endif %}
30

```