**Student Management System: Project Report**

**1. Setup Process:**

* Create a new directory for the project.

**mkdir django-management**

* After creating the directory, to go into the directory.

**cd django-management**

* Now, create a new virtual environment using the below command

**python -m venv venv**

* To enter the virtual environment:

**.\venv\Scripts\activate**

* Now after entering the virtual environment, to install Django and dependencies:

**python -m pip install Django**

* Create a project named “student\_management”:

**django-admin startproject student\_management .**

After executing the above command, the project structure will look like below:

A screen shot of a computer program

Description automatically generated

* Within the project, an app named “management” was created to handle the core student management features:

**python manage.py startapp management**

After executing the above command, the application structure inside the student\_management project will look like below:

**A screen shot of a computer program

Description automatically generated**

* After completing the above steps, add the app “management” to INSTALLED\_APPS in settings.py

**A screen shot of a computer

Description automatically generated**

**2. Database Setup:**

* Django uses SQLite by default. And in this project also the default database is being used.
* Now, create a model “Student” in model.py which is inside management app.

**A screen shot of a computer program

Description automatically generated**

In the “Student” model, first\_name, last\_name with datatypes to accept characters with maximum length of 50, email field, date\_of\_birth, enrollment\_date, grade. After this, to migrate the model.

**python manage.py makemigrations**

**python manage.py migrate**

(**Note: The above two commands are executed every time a model is updated)**

**3. Creating a super user:**

* For creating a super user, execute the following command in the command prompt and enter the details which are asked (Username, Email address, Password).

**python manage.py createsuperuser**

* After creating the super user, to run the application, execute the following command in the command prompt

**python manage.py runserver**

* Now go to the browser, and navigate to <http://localhost:8000/admin> or <http://127.0.0.1:8000/admin> and log in using the super user credentials which you have created in the above steps, you will be seeing the below page:

A screenshot of a computer

Description automatically generated

* In management directory, open the admin.py file, and add the student model like below:

A screen shot of a computer program

Description automatically generated

* After re-compiling, and refresh the browser page, you will be seeing the below page:

A screenshot of a computer

Description automatically generated

**4. Views and Templates:**

**4.1 Creating Views:**

* The `forms.py` file needs to be created in the `management` application, followed by importing the `Student` model using an import statement at the top.
* Few input validations were added for email (email validation) and grades (It need to be between 1 and 12) before saving into database.
* In the \_\_init\_\_ method, the form fields and placeholders are specified.
* These things are used in the forms while adding a student. And while editing an existing student, the same fields are displayed.

**4.2 Creating Templates:**

* Create a “templates” directory inside management app. And inside the templates directory, three HTML files were created:

1. student\_list.html

2. student\_detail.html

3. student\_form.html

**1. student\_list.html:** It will display the list of students present. Inside this student\_list.html there are few functionalities which are implemented.

**1.1 Search Functionality:** Q class which is provided by Django in django.db.models module is used . If a query is present, the Student model is queried to filter results that contain the search term, either in the first\_name or last\_name fields, using the Q object with the icontains lookup for case-insensitive matching.

**1.2 Displaying Student list:** The list of students is displayed in a <ul> (unordered list), with each student shown as a list item (<li>).For each student, it displays their first and last name, which is a link to the detailed page for that student. It also shows the enrollment date as a badge.

**1.3** **Records per page:** This form provides an option to select the number of student records to display per page. When the user selects a different option, the form is automatically submitted to adjust the number of students displayed.

**1.4 Pagination:** This section handles pagination, allowing users to navigate through pages of student records. Users can navigate to the first, previous, next, or last pages.

**2. student\_detail.html:** It will display detailed information about a specific student which is selected from student list. It displays the complete name, email, date of birth, enrollment date and grade. Three buttons are provided (Edit Student, Back to Student List, Delete Student).

Edit student will allow users to edit the student’s details.

Back to Student List will redirect user to the students list page.

Delete Student will trigger a modal that asks for confirmation before deleting the student.

**3. student\_form.html:** This template is used to create and edit student information.

**5. Routing**

Routing in Django is managed through URL configurations, defined using urls.py files. The project and application-specific URL patterns in this system are divided into two parts: student\_management/urls.py and management/urls.py.

* **Project-Level URL Configuration (student\_management/urls.py):** The project-level URL configuration, located in student\_management/urls.py, serves as the main entry point for the routing system. It includes two main URL patterns:

**1.** path("", include("management.urls")): Routes all requests to the management application by including its URLs. This modular approach helps organize the URLs for specific functionalities of the application.

**2.** path("admin/", admin.site.urls): Configures the URL path for accessing the Django admin interface, allowing administrators to manage the project's data.

* **App-Level URL Configuration (management/urls.py):** The management/urls.py file defines the URLs specific to the student management application. Each URL pattern is associated with a specific view function that processes the request and returns the appropriate response.

1. path('', views.student\_list, name='student\_list'): Defines the home page of the student management system, which displays the list of students.

2. path('student/<int:pk>/', views.student\_detail, name='student\_detail'): Routes to the detail page of a specific student, where <int:pk> represents the primary key of the student in the database.

3. path('student/new/', views.student\_create, name='student\_create'): Routes to the form for adding a new student.

4. path('student/<int:pk>/edit/', views.student\_edit, name='student\_edit'): Routes to the form for editing an existing student's information.

5. path('student/<int:pk>/delete/', views.student\_delete, name='student\_delete'): Routes to a view that allows users to delete a student.

6. path('register/', views.register, name='register'): Provides the URL for the registration page.

7. path('login/', views.user\_login, name='login'): Defines the login page URL.

8. path('logout/', views.user\_logout, name='user\_logout'): Routes to the logout functionality, redirecting users to the login page after logging out.

9. path('forgot\_password/', views.forgot\_password, name='forgot\_password'): Provides the URL for the forgot password functionality.

**6. Base Template**

* Created a base.html inside the templates folder and this template will extend to the child templates from before.
* {% extends %} tag needs to be added at the beginning of a child template. By adding {% extends "base.html" %} to any html page, the template will inherit the structure of base.html.
* In this base template, navbar is created in the base.html and student\_form.html, student\_list.html, student\_detail.html pages will inherit the base.html.

For example,

A screen shot of a computer

Description automatically generated

The above image is from student\_list.html page. student\_lost.html is inheriting base.html page.

**7. Authentication**

The authentication functionality in the student management system has been implemented to provide user registration, login, logout, password recovery features. A new directory “registration” is created inside templates and created register.html, login.html, forgot\_password.html files. After that add the line LOGIN\_URL = 'login' in settings.py to specify the URL where users should be redirected when they attempt to access a view that requires authentication but are not logged in.

**7.1 Registration**

* **View (register function):** The user registration is handled using Django's UserCreationForm. The register view method processes the POST request when a user submits the registration form, ensuring the data is valid. If the form is successfully validated, a new user account is created, and the user is redirected to the login page. Error messages are shown if the form validation fails.
* **Template (register.html):** The register.html template contains the registration form, which uses Bootstrap for styling. There are additional helpful bullet points under each field to provide guidelines for creating a strong username and password, ensuring clarity for the user.

**7.2 Login**

* **View (user\_login function):** The user\_login view handles the login process, using Django's built-in AuthenticationForm. After a successful login, the user is redirected to the student list page. Error messages are provided if the login credentials are incorrect.
* **Template (login.html):** The login.html template has a well-styled login form with animation and uses Bootstrap to make it more user-friendly. The template also provides links for forgotten passwords and new user registration.

**7.3 Forgot Password**

* **View (forgot\_password function):** The forgot password feature allows users to reset their password if they forget it. Users are required to provide their username and new password. The view verifies if the username exists and whether the new password matches the confirmation. It also uses Django's built-in validate\_password function to ensure that the password meets the security criteria. Once the password is successfully updated, the user is redirected to the login page.
* **Template (forgot\_password.html):** The forgot\_password.html template provides a form for users to input their username and the new password. It is styled consistently with the other templates and has appropriate feedback messages for users.

**7.4 Logout**

* **View (user\_logout function):** The user\_logout function uses Django's logout() method to end the user session. After logout, the user is redirected to the login page with a success message.

**7.5 Authentication Requirements for Sensitive Actions:** The sensitive views that allow creating, editing, or deleting student records are protected using the explicit authentication checks (if not request.user.is\_authenticated). These checks ensure that only authenticated users can perform these actions. This prevents unauthorized access and ensures that users need to log in before performing these actions.

A screen shot of a computer program

Description automatically generated

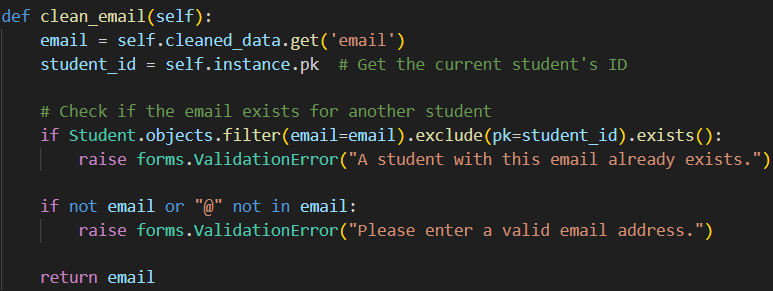
In this code snippet, it is understood that only the authenticated users can be able to add, edit and delete the student. If he is not authenticated it will redirect the user to the login page.

**7.6 Notification Messages:** The implementation includes Django's messaging framework to provide real-time feedback for actions such as successful login, account creation, password update, and logout. These notifications appear as green pop-ups on the top right corner of the screen and disappear after a short time, enhancing the user experience.

**8. Features Implemented**

**8.1 Student CRUD Functionality:** The core of the system involved CRUD (Create, Read, Update, Delete) operations for students records.

* **Forms:** The StudentForm was implemented using Django's ModelForm, providing user input validation for fields like email and grade.

****

The above code snippet will check in the database whether there is any existing record with the same email entered by the user. If it exists, it raises an error or else it proceeds with the validation whether the user’s input is matching the email criteria or not.

A computer screen shot of text

Description automatically generated

The above code snippet will check whether the user input for the grade field is between 1 and 12 or not. These two are an extra level of validation, the other level of validation is with the model.

A screen shot of a computer

Description automatically generated

In the Student model, database level of validation is added to the email and grade field.

* **Views:** Views for adding, editing, deleting, listing, and viewing student details were created.
* **Templates:** The templates (student\_form.html, student\_list.html, student\_detail.html) were designed with Bootstrap to create a responsive user interface.

**8.2 User Authentication:** Authentication was critical to ensure data security.

* **User Registration and Login:** Forms for user registration (register.html) and login (login.html) were developed, with validation to ensure that email and username were unique.
* **Restricted Access:** Only authenticated users could add, edit, or delete student records, enforcing security for sensitive operations.
* **Password Reset:** A "Forgot Password?" feature was added, allowing users to reset their password through a dedicated page.

**8.3 Notifications:** Dynamic notifications were added to improve user interaction.

**8.4 Pagination and Search Functionality**

* **Pagination:** The student list view was paginated to display a manageable number of students per page, improving usability for larger datasets.
* **Search Functionality:** A search bar was added to filter students by first or last name, making it easier to find specific records.

**8.5 Error Handling and Validation:** Robust error handling and validation were implemented to ensure data integrity.

* **Email Validation:** The StudentForm checked whether an email was already in use and ensured the email format was valid.
* **Grade Validation:** Grades were validated to ensure they fell within a defined range (1-12).
* **Username and Email Uniqueness:** Both fields were checked for uniqueness during user registration to prevent duplicates.

A screen shot of a computer program

Description automatically generated

This is my final project structure.

**9. Challenges Encountered**

**9.1 Dynamic Notifications**

* **Challenge:** The goal was to send notifications after each save or error action. Initially, a separate notification logic was created for this purpose. However, it was later realized that this approach was neither ideal nor efficient, and it became somewhat challenging to implement.
* **Solution:** Created a dynamic notification system where the notification message is to be sent and whether it is a success or error message. The solution involved using Django's built-in messages framework, which provides an easy way to pass one-time messages to the user. JavaScript was used to add fading effects and automatically remove the notifications after a few seconds. Proper CSS styling was also used to position notifications effectively, ensuring they were clearly visible to the user without disrupting the flow of the page. The dynamic notification HTML code is written in base.html which is reusable and to render that dynamic notification, we should send only the messages like below.

****

****

These messages will create a notification right top corner with red color and green color simultaneously.

Base.html notification implementation:  
A screen shot of a computer program

Description automatically generated

The notifications will be faded away after three seconds. And the message which was sent, will be displayed in place of {{ message }}.

**9.2 Authentication Implementation**

* **Challenge:** Adding authentication and restricting access to specific views based on user roles posed challenges, especially in maintaining a good user experience and ensuring proper redirection. Ensuring that unauthenticated users were redirected to the login page while retaining their original intent was difficult to implement.
* **Solution:** This was addressed by using Django's built-in @login\_required decorator, which ensured that only authenticated users could access certain views. Additionally, the next parameter was utilized to redirect users to their intended destination after successful login.

**9.3 Authentication requirements for sensitive actions**

* **Challenge:** At first, for the sensitive views that allow creating, editing, or deleting student records are protected using the @login\_required decorator. But to send a notification when the user gets redirected to the login page, we need to create a custom login or create a separate route.
* **Solution:** Instead, removed the @login\_required decorator, and used “if not request.user.is\_authenticated” statement and with this and achieved both the functionalities (authentication required for sensitive actions and sending notifications).