Student Management System Documentation

5 , May 2023

Shoubra Faculty of Engineering

Computer Engineering Department 3rd Year

Project Overview

Student Management system that allows for creating reading updating and deleting students information this core functionality is known by the acronym crude create read update delete almost every web application uses crude and by the end of this tutorial you'll be able to use django to build your own web apps that implement crude operations .This is the application we are going to build on the home page we have a list of students presented in a table for each student we have available three action buttons that allow us to view edit and delete students from a database if we press the first button with the information icon you can view the information of a selected student we use here a bootstrap model to present the data to the user if we press the second button with the edit icon we can edit the information of a selected student if we press the third button with the delete icon we can delete a selected student from a database before deleting a student we are first presented with a bootstrap model that asks us to confirm the deletion we can also add a new student when adding a new student we need to provide the student number a first and last name and email a field of study and a gpa.We also have subjects for each students actually six subjects assigned for each academic year and each subject has an instructor . we can also add new subjects and instructors for these subjects.

Installation

Prerequisites

Install the following prerequisites:

1. Python 3.10.4 or higher
2. Visual Studio Code

Procedures

Why Django ?

1. Rapid development: Django's built-in features allow developers to quickly build and deploy web applications without sacrificing quality.
2. Scalability: Django is designed to handle large-scale projects, making it a good choice for companies or organizations that require a robust and scalable web application.
3. Security: Django provides a number of built-in security features, including protection against common web-based attacks such as cross-site scripting (XSS) and SQL injection.
4. Versatility: Django can be used for a wide range of web development projects, including content management systems, social networking sites, e-commerce platforms, and more.
5. Community support: Django has a large and active community of developers who contribute to the framework and provide support through online forums, documentation, and other resources.

Overall, Django is a powerful and flexible web framework that can help you build web applications quickly and securely, while also allowing for scalability and versatility.

Why Bootstrap ?

1. Consistency: Bootstrap provides a consistent set of styles and design elements that can be used throughout your web application, giving it a polished and professional look.
2. Responsiveness: Bootstrap is designed to be mobile-first, meaning that it's optimized for viewing on smaller devices like smartphones and tablets. This makes it easier to create web applications that work well across different screen sizes and resolutions.

3. Customization: Bootstrap provides a wide range of customization options, allowing developers to create unique and personalized designs within the framework.
4. Cross-browser compatibility: Bootstrap is compatible with all major web browsers, meaning that your web application will function properly regardless of which browser your users are using.
5. Large community: Bootstrap has a large and active community of developers who contribute to the framework and provide support through online forums, documentation, and other resources.

Overall, Bootstrap can help you create high-quality, responsive, and mobile-friendly web applications quickly and efficiently, while also providing a consistent and professional look and feel.

Virtual Environment

virtual environment for our project you might be wondering why do we need to create a virtual environment installing the latest version of python and django is the correct approach for any new django project however if we work on several different projects over time it is common for these existing projects to use different versions of python and django by default python and django are installed globally on a computer and it would be very inconvenient to install and reinstall different versions of python in django every time we want to switch between different projects and this is where virtual environments come in handy virtual environments allow us to create and manage separate environments for each python project on the same computer and it is recommended that we create a dedicated virtual environment for each of our django projects there are several ways to create a virtual environment in python but the simplest method is to use the venv module already installed as part of the python standard library to create a virtual environment we use the command python mvenv environment name it is up to you to select the name of the environment but it is a common practice to call it vnv to create a new virtual environment in visual studio code navigate to the menu on the top click terminal and then select new terminal next run the proper commands

if you want to exit the virtual environment just type deactivate at the command prompt.

Overall

1. Create a virtual environment

From the root directory run : python -m venv venv

2. Activate the virtual environment

From the root directory run:

On macOS : source venv/bin/activate

On Windows : venv\scripts\activate

Django Project Structure

- manage.py: A command-line utility for interacting with the project.
- project_name/: The root of the project directory. It contains the main configuration files for the project.
- settings.py: Contains the project-specific settings, such as database configuration, installed apps, and middleware.
- urls.py: Contains URL configurations for the project.
- asgi.py and wsgi.py: Contains configuration for the ASGI and WSGI servers, respectively.
- app1/, app2/: The individual app directories.
- models.py: Defines the data models used by the app.
- views.py: Contains the logic for handling HTTP requests and returning responses.
- admin.py: Defines the admin interface for the app.
- tests.py: Contains test cases for the app.
- migrations/: Contains database migration files.
- static/: Contains static files, such as CSS, JavaScript, and image files.
- templates/: Contains HTML templates used by the app.

Views.py File

In Django, views are Python functions that take an HTTP request and return an HTTP response. Views are responsible for processing user requests and returning appropriate responses.The views.py file in a Django app contains all of the app's views. A view is typically a function that is mapped to a URL pattern defined in the app's urls.py file. When a user makes a request to a URL that matches a URL pattern defined in urls.py, Django calls the corresponding view function.

Code here…

Templates Folder

In Django, templates are files that define how a web page will look like. They typically use HTML markup with some additional syntax provided by Django's template language.

To create a template in Django, you need to follow these steps:

1. Create a new file with a .html extension in your app's templates directory (e.g. myapp/templates/mytemplate.html).
2. Start the file with the following code:

```
{% extends "base.html" %}
{% block content %}
<!-- Your HTML code goes here -->
{% endblock %}
```

1. The {% extends "base.html" %} tag tells Django that this template should inherit from another template called "base.html". This allows you to reuse common elements across multiple pages, such as headers and footers.The {% block content %} and {% endblock %} tags define a section of the template that can be overridden or extended in the inherited template, allowing you to customize the content for each page.
2. Add any desired HTML markup inside the content block.
3. Use the Django template language to insert dynamic content into the template using variables, loops, conditionals, and other constructs. For example:

```
<h1>{{ page_title }}</h1>
<p>{{ article.body }}</p>
{% if user.is_authenticated %}
<a href="{% url 'logout' %}">Logout</a>
{% else %}
<a href="{% url 'login' %}">Login</a>
{% endif %}
```

Here, page_title and article.body are variables passed into the template from the view function, and {% url 'logout' %} and {% url 'login' %} generate URLs based on named URL patterns defined in your app's urls.py file.That's the basic structure of a Django template. You can create as many templates as you need for your app, and use them to render different views based on user input or other factors.

urls.py File

In a Django project, URLs are defined in the urls.py file. The urls.py file defines a list of URL patterns that Django uses to match incoming requests with views.To define a URL pattern, you first import the view function or class that will handle the request. You then use Django's URL routing syntax to map a URL pattern to the view. Here's an example:

```
from django.urls import path
from . import views
```

```
urlpatterns = [
    path('blog/', views.blog_list),
    path('blog/<int:pk>/', views.blog_detail),
    path('about/', views.about),]
```

In this example, we've defined three URL patterns using the path() function. The first maps the URL /blog/ to the blog_list view, the second maps URLs like /blog/123/ to the blog_detail view (where 123 is the primary key of a blog post), and the third maps the URL /about/ to the about view.Note that you can also use regular expressions in your URL patterns to match more complex URLs. For example:

```
from django.urls import re_path
from . import viewsurlpatterns = [
re_path(r'^blog/(?P<year>\d{4})/$', views.blog_year_archive),
re_path(r'^blog/(?P<slug>[\w-]+)/$', views.blog_detail),
]
```

In this example, we've used re_path() instead of path() to allow for regular expressions. The first URL pattern matches URLs like /blog/2019/ and passes the year as a named argument to the blog_year_archive view. The second URL pattern matches URLs like /blog/my-blog-post/ and passes the slug as a named argument to the blog_detail view.

Models.py file

The models.py file in a Django project defines the database schema for your application. It is where you define the tables and fields that will be used to store data in your database.Each model in models.py represents a database table, and each attribute of the model represents a column in that table. For example, if you were building a blog application, you might have a Post model with attributes like title, body, author, and date_published. Here's an example:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)
    body = models.TextField()
    author = models.ForeignKey('auth.User', on_delete=models.CASCADE)
    date_published = models.DateTimeField(auto_now_add=True)
```

In this example, we're using Django's Model class as our base class. We then define our Post model with four attributes: title, body, author, and date_published. The title and body attributes are both CharField and TextField respectively, which are text-based fields. author is a foreign key that links to the User model provided by Django's built-in authentication system, and date_published is a DateTimeField that automatically sets the current date and time whenever a new Post instance is created.Once you have defined your models in models.py, you can use Django's ORM to interact with your database and perform operations such as creating, reading, updating, and deleting records.

Migrations

In a Django project, migrations are used to manage changes to your application's database schema over time. Whenever you make changes to your models in models.py, you create a migration to apply those changes to the database.

Here are the basic steps for creating and applying migrations:

1. Make changes to your models in models.py.
2. Run python manage.py makemigrations to create a new migration file. This will generate a Python file in the migrations directory that contains instructions for the changes you made.
3. Review the generated migration file to ensure it looks correct.
4. Run python manage.py migrate to apply the changes to the database.

The makemigrations command generates a migration file based on the differences between the current state of your models and the previous state recorded in the last applied migration. The migration file includes instructions to modify the tables in the database to match the new model structure.

When you run the migrate command, Django applies any outstanding migrations in order, ensuring that the database schema is in sync with your model definitions. If there are no new migrations to apply, the migrate command does nothing.

Migrations are an important feature of Django because they allow you to evolve your database schema over time without manually modifying the database tables. This makes it easier to maintain and update your application's database schema as your requirements change.

MySql VS SQlite3

There are several reasons why we choose to use MySQL instead of SQLite3 in Django:

1. Scalability: MySQL is designed to handle large-scale applications, while SQLite3 is better suited for smaller projects. If you anticipate that your application will grow in size, MySQL may be a better choice.
2. Performance: MySQL generally performs faster than SQLite, particularly when dealing with large datasets and complex queries.
3. Compatibility: MySQL is widely used across many different technology stacks, so if you plan on integrating your Django application with other systems, it may be easier to work with MySQL.

4. Advanced features: MySQL offers a number of advanced features such as stored procedures, triggers, and views that may be useful for more complex applications.

However, it's worth noting that SQLite3 can still be a good choice for certain types of Django applications, particularly those that don't require extensive scaling or complex data relationships. Ultimately, the choice between MySQL and SQLite3 will depend on the specific needs of your project.

Static Files

In Django, static files are files that don't change during the runtime of the application. These include images, CSS files, JavaScript files, and other assets that are used to build the user interface.To serve static files in a Django application, you need to add some configuration to your project's settings file and set up the URL patterns for serving the files.Here are the basic steps to serve static files in Django:

1. Add the following lines to your settings.py file:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
]
```

2. Create a folder named 'static' at the root of your project directory and put your static files in it.
3. In your urls.py file, add the following line at the end:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ... your URL patterns here ...
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

With these configurations in place, Django will automatically serve static files from the static folder in your project directory when requested through the URL pattern specified in STATIC_URL.

Note: It is also important to properly configure your web server (e.g. Apache, Nginx) to serve static files efficiently in production environments.

Create a Superuser

To create a superuser in Django using VS Code, you can follow these steps:

1. Open up your VS Code editor and navigate to the terminal.

2. Navigate to your Django project directory using the cd command.

3. Run the following command:

```
python manage.py createsuperuser
```

4. You will be prompted to enter a username, email address, and password for the superuser.
5. After entering the required information, press Enter. If everything goes smoothly, you should see a message confirming that the superuser was created successfully.

HTTP Request-Response Cycle in django

In Django, the HTTP request-response cycle is handled by the web server and the Django framework working together.

When a client sends an HTTP request to a server running a Django application, the web server receives the request and passes it on to Django. Django then processes the request and generates an appropriate response, which is returned to the web server for transmission back to the client.

The request-response cycle in Django follows the same basic steps as any other HTTP transaction:

1. The client sends an HTTP request to the server.

2. The server receives the request and passes it to Django.

3. Django processes the request, including any middleware or view functions defined in the application.

4. Django generates an appropriate response, which includes any requested data or resources, along with any necessary metadata like headers or status codes.

5. The server receives the response from Django and transmits it back to the client.

6. The client receives the response and can use the received data to render the requested resource or perform additional actions based on the response content.

Django provides a powerful set of tools for handling the request-response cycle, including middleware, views, templates, and URL routing. With these tools, developers can build complex web applications that handle requests and generate responses in a highly customizable and extensible way.

HTTPS GET and POST Requests

In Django, the GET and POST methods are used to handle HTTP requests. Here is a brief explanation of how they work:

- GET: This method requests data from a specified resource. It's commonly used for retrieving data from a server. In Django, you can handle GET requests by defining a view function that accepts a HttpRequest object and returns an HttpResponse object.

  For example:

```
from django.http import HttpResponse
def my_view(request):
    if request.method == 'GET':return HttpResponse('This is a GET request')
```

- POST: This method submits an entity to the specified resource, often causing a change in state or side effects on the server. It's commonly used for submitting data to a server. In Django, you can handle POST requests by defining a view function that checks if the request method is POST, retrieves the submitted data, performs any necessary validation or processing, and returns an HttpResponse object.

  For example:

```
from django.http import HttpResponse

def my_view(request):
    if request.method == 'POST':
        username = request.POST.get('username');password = request.POST.get('password')
        return HttpResponse('This is a POST request')
```

Note that in the POST example above, we retrieve the submitted data using the request.POST attribute, which contains a dictionary-like object of the submitted data. We then perform any necessary validation or processing before returning an HttpResponse object.

Template Inheritance - Extending Templates

Template inheritance is a powerful feature in Django that allows you to create reusable templates by building on top of existing ones. With template inheritance, you can define a "base" template that contains the common elements of your site (such as the header and footer) and then extend that template to create child templates that add additional content.

To extend a template in Django, you first need to create a base template. This template should contain the parts of your page that will be common across all pages of your site. For example, you might create a "base.html" template that contains the header, navigation, and footer:

```html
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}My Site{% endblock %}</title>
</head>
<body>
  <header>
    <!-- Header content -->
  </header>

  <nav>
    <!-- Navigation links -->
  </nav>

  <main>
    {% block content %}
    {% endblock %}
  </main>

  <footer>
    <!-- Footer content -->
  </footer>
</body>
</html>
```

Note the use of the {% block %} tags here. These define areas of the template that can be overridden or extended by child templates. In this case, we have defined a title block and a content block, which child templates can use to provide their own values.

To extend the base template in a child template, you can use the {% extends %} tag. For example, if you wanted to create a "home" page that includes some specific content in addition to the standard header and footer, you could create a template like this:

```html
{% extends "base.html" %}{% block title %}Home | {{ block.super }}{% endblock %}
{% block content %}
<h1>Welcome to my site!</h1>
<p>This is the home page.</p>
{% endblock %}
```

Note that we are using the {% extends %} tag to specify that this template should extend the "base.html" template. We are also overriding the title block to add the text "Home | " before the default value (which is "My Site").

Finally, we are defining a new content block that contains the specific content for this page. When Django renders this template, it will replace the contents of the content block in the base template with the contents of this block.

Overall, template inheritance can be a powerful tool for creating reusable and maintainable templates in Django. By breaking your templates into smaller pieces and using inheritance to combine them as needed, you can create a flexible and modular system that makes it easy to update your site's design and content over time.

CSRF Token and Security in Django

Django provides built-in protection against CSRF (Cross-Site Request Forgery) attacks, which are a common type of web application security vulnerability.

CSRF attacks occur when a malicious website or attacker tricks a user into performing an action on another website without their consent. For example, if a user is logged in to a bank's website and visits a malicious site, the attacker could use code on that site to make unauthorized transfers from the user's account.

To prevent these attacks, Django includes a CSRF middleware that adds a unique token to each form submitted by a user. This token is validated when the form is processed, ensuring that the request came from the same site and preventing any unauthorized actions.

To use this protection, you simply need to include the {% csrf_token %} template tag in your forms. Django takes care of the rest.

You can also customize the behavior of the CSRF middleware by modifying the CSRF_COOKIE_* settings in your Django settings file. For example, you can change the name of the cookie used to store the CSRF token or adjust the expiration time.

Overall, using Django's built-in CSRF protection is an important step in securing your web application against common attacks.