Tops Technology

Module 16)
Python DB and Framework

Presented By : Nandni Vala

Django Forms and Authentication

1.Using Django’s built-in form handling.

- Django’s built-in form handling simplifies the process of working with forms by providing tools to render, validate, and process form data seamlessly.
- **Steps for Using Django’s Built-In Form Handling**
- **1. Define a Form Class**
- Create a Python class that inherits from forms.Form or forms.ModelForm.
- **Using forms.Form**:
- from django import forms
- class ContactForm(forms.Form):
-     name = forms.CharField(max_length=100, required=True)
-     email = forms.EmailField(required=True)
-     message = forms.CharField(widget=forms.Textarea, required=True)

- **Using `forms.ModelForm`** (for models):
  - from django import forms
  - from .models import Feedbacs
  - class FeedbackForm(forms.ModelForm):
  -     class Meta:
  -         model = Feedback
  -         fields = ['user', 'comments']
  - **Render the Form in a Template**
  - Pass the form instance to the template context.
- **View Function**:
  - from django.shortcuts import render
  - from .forms import ContactForm
  - def contact_view(request):
  -     form = ContactForm()
  -     return render(request, 'contact.html', {'form': form})

- **Template**:

```
➢ <form method="post">
➢    {% csrf_token %}
➢  {{ form.as_p }} <!-- Render form fields -->
➢    <button type="submit">Submit</button>
➢ </form>
```

- **Handle Form Submission**
- Process the submitted data in the view.

```
➢ def contact_view(request):
➢     if request.method == "POST":
➢         form = ContactForm(request.POST)
➢         if form.is_valid():
➢             # Access cleaned data
➢             name = form.cleaned_data['name']
➢             email = form.cleaned_data['email']
```

- message = form.cleaned_data['message']
- # Process form data (e.g., save or send email)
- return render(request, 'success.html')
- else:
- form = ContactForm()
- return render(request, 'contact.html', {'form': form})

- **Key Features of Django Forms**
- **Validation**:
  - Automatic validation based on field types (e.g., EmailField checks for valid email format).
  - Custom validation using clean() methods.
- def clean_email(self):
- email = self.cleaned_data.get('email')
- if not email.endswith('@example.com'):
- raise forms.ValidationError("Email must end with @example.com")
- return email

- **Widgets**: Customize input fields using widgets:
- name = forms.CharField(widget=forms.TextInput(attrs={'class': 'form-control'}))
- **Error Handling**: Automatically displays errors for invalid fields
- **Advantages of Django's Form Handling**
- Simplifies form creation and validation.
- Provides automatic error messages and protection against CSRF attacks.
- Integrates seamlessly with Django models through `ModelForm`.

.

2.Implementing Django's authentication system (sign up, login, logout, password management).

- ➢ **Set Up Authentication**
- ➢ Ensure the following settings are configured in settings.py:
- ➢ # Authentication settings
- ➢ LOGIN_REDIRECT_URL = '/'   # Redirect after successful login
- ➢ LOGOUT_REDIRECT_URL = '/'   # Redirect after logout
- ➢ **Sign-Up (User Registration)**
- ➢ **Form**: Create a custom registration form or use UserCreationForm from Django.

```
from django.contrib.auth.forms import UserCreationForm
class SignUpForm(UserCreationForm):
    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']
```

- **View**: Handle the sign-up process.

```
from django.shortcuts import render, redirect
from .forms import SignUpForm
def signup_view(request):
    if request.method == 'POST':
        form = SignUpForm(request.POST)
        if form.is_valid():
            form.save()  # Create a new user
            return redirect('login')
    else:
        form = SignUpForm()
    return render(request, 'signup.html', {'form': form})
```

- **Template**: Render the form.

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Sign Up</button>
</form>
```

- **Login**
- **View**: Use Django’s built-in LoginView.
- from django.contrib.auth.views import LoginView
- class CustomLoginView(LoginView):
-     template_name = 'login.html'
- **Template**: Render the login form.
- <form method="post">
-     {% csrf_token %}
-     {{ form.as_p }}
-     <button type="submit">Log In</button>
- </form>
- **URL Configuration**: Add a route for the login view.
- from django.contrib.auth.views import LoginView
- urlpatterns = [
-     path('login/', LoginView.as_view(template_name='login.html'), name='login'),
-]

- **Logout**
- **View**: Use Django’s built-in LogoutView.
- from django.contrib.auth.views import LogoutView
- urlpatterns += [
-     path('logout/', LogoutView.as_view(), name='logout'),
-]
- **Template**: Provide a logout button.
- <a href="{% url 'logout' %}">Log Out</a>
- **Password Management**
- **Password Reset Views**: Django provides ready-to-use views for password reset and change:
- from django.contrib.auth import views as auth_views
- urlpatterns += [
-     path('password_reset/', auth_views.PasswordResetView.as_view(), name='password_reset'),
-     path('password_reset/done/', auth_views.PasswordResetDoneView.as_view(), name='password_reset_done'),

- path('reset/<uidb64>/<token>/', auth_views.PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
- path('reset/done/', auth_views.PasswordResetCompleteView.as_view(), name='password_reset_complete'),
- path('password_change/', auth_views.PasswordChangeView.as_view(), name='password_change'),
- path('password_change/done/', auth_views.PasswordChangeDoneView.as_view(), name='password_change_done'),
-]
- **Templates**: Django looks for templates named password_reset_form.html, password_change_form.html, etc. Customize them as needed.
- **Middleware for Authentication**
- Ensure AuthenticationMiddleware is enabled in MIDDLEWARE in settings.py:
- MIDDLEWARE = [
- ...
- 'django.contrib.auth.middleware.AuthenticationMiddleware',
- ...
-]

- **Restricting Access to Views**
- Use Django's decorators or mixins to restrict views to logged-in users:
- **Function-Based Views**:
- from django.contrib.auth.decorators import login_required
- @login_required
- def profile_view(request):
-     return render(request, 'profile.html')
- **Django Authentication Benefits**
- Built-in security features (e.g., password hashing, session management).
- Fully customizable forms and templates.
- Easy integration with third-party libraries for social login or multi-factor authentication.