Module-03

Django Admin Interfaces and Model Forms

Activating Admin Interfaces

Update settings.py:

Add to INSTALLED_APPS:

- 'django.contrib.admin'
- Ensure 'django.contrib.auth', 'django.contrib.contenttypes', and 'django.contrib.sessions' are included. Uncomment if previously commented.

Update MIDDLEWARE_CLASSES:

Ensure the following middlewares are included and uncommented:

- 'django.middleware.common.CommonMiddleware'
- 'django.contrib.sessions.middleware.SessionMiddleware'
- 'django.contrib.auth.middleware.AuthenticationMiddleware'

Sync the database:

Run database synchronization:

- Execute python manage.py syncdb to install the necessary database tables for the admin interface.
- If not prompted to create a superuser, run python manage.py createsuperuser to create an admin account.

Update urls.py:

Include the admin site in URLconf:

- Ensure the following import statements are present

```
from django.contrib import admin
admin.autodiscover()
```

- Add the admin URL pattern

```
urlpatterns = patterns('',
    # ...
    (r'^admin/', include(admin.site.urls)),
    # ...
)
```

Access the admin interface:

- Run the development server and access the admin site:
- Start the server with python manage.py runserver.
- Visit http://127.0.0.1:8000/admin/ in your web browser.

Using Admin Interfaces

Logging In:

- Visit the admin site and log in with the superuser credentials you created.
- If you can't log in, ensure you've created a superuser by running python manage.py createsuperuser.

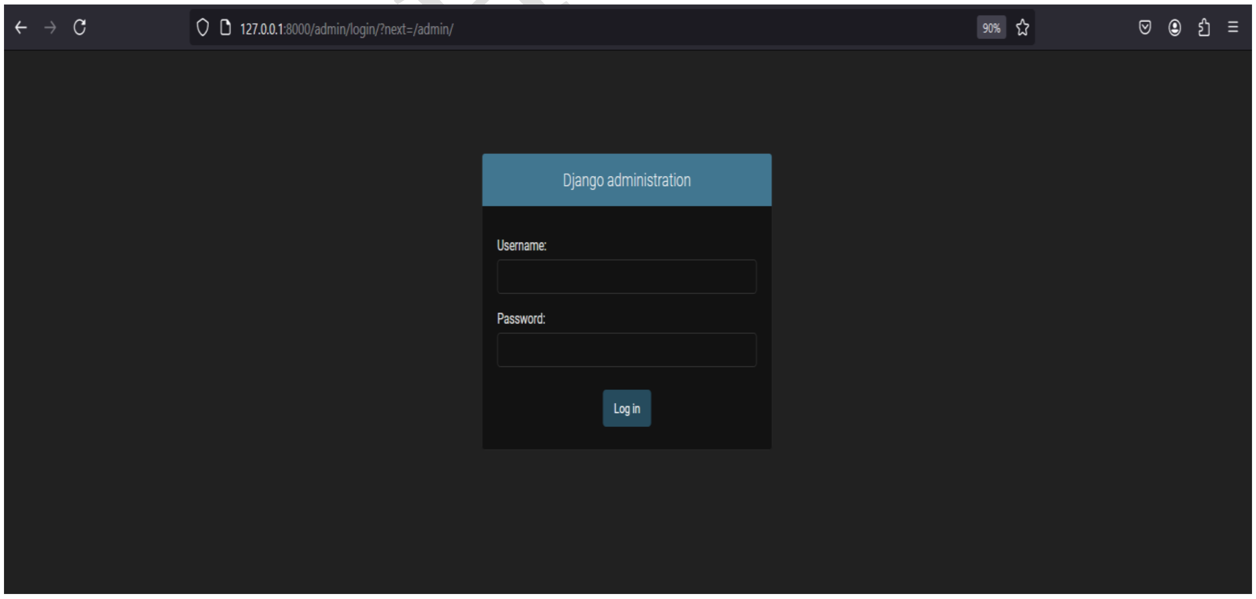


Figure: Django's login screen

Admin Home Page:

- After logging in, you'll see the admin home page listing all data types available for editing. Initially, it includes only Groups and Users.

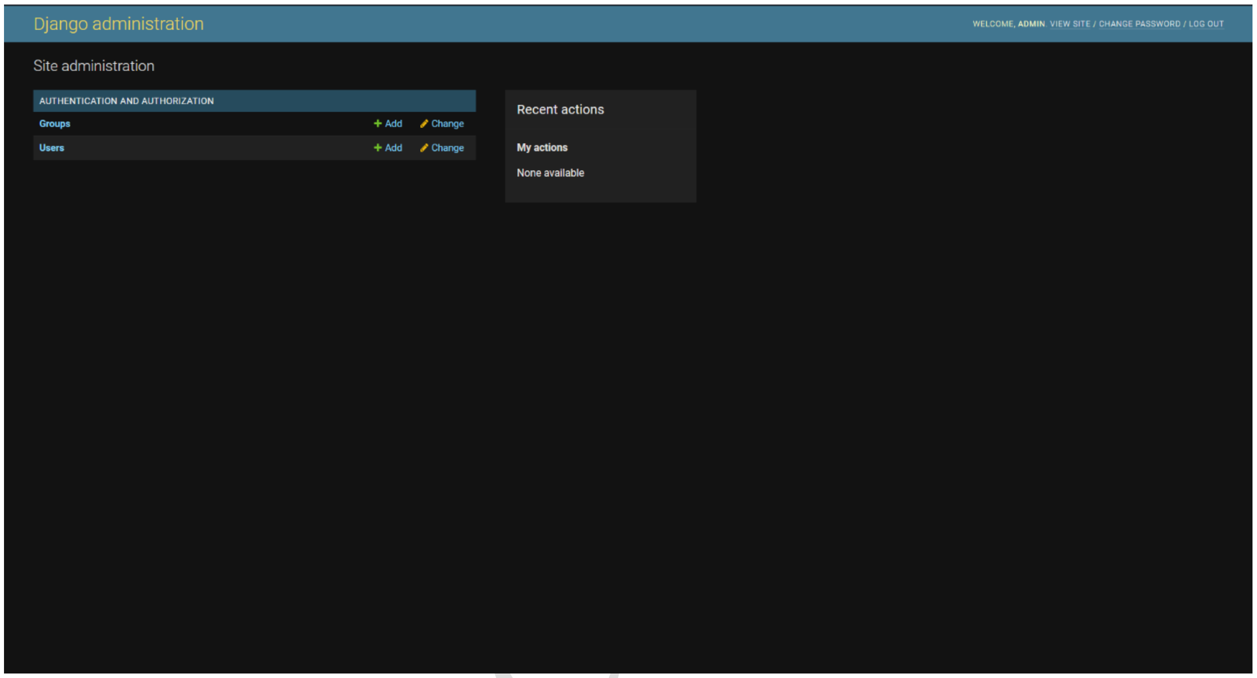


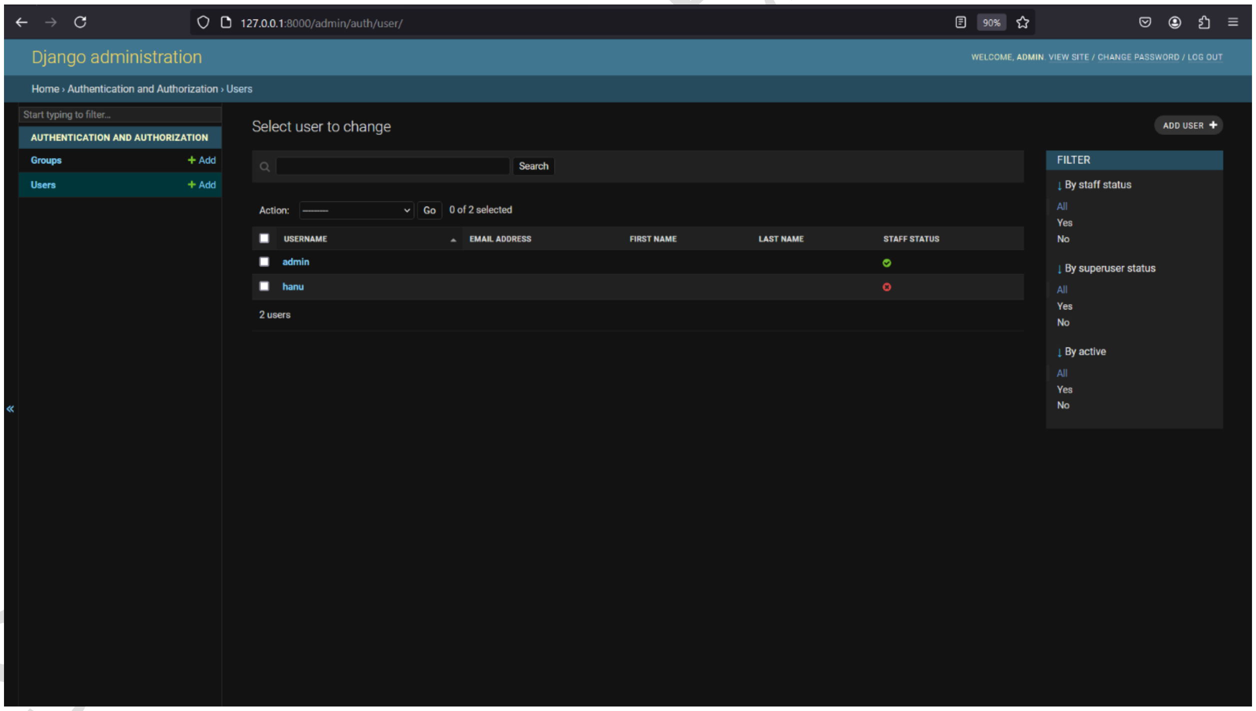
Figure: - The Django admin home page

Data Management:

- Each data type in the admin site has a change list and an edit form.
- Change List: Displays all records of a data type, similar to a SELECT * FROM <table> SQL query.
- Edit Form: Allows you to add, change, or delete individual records.

Managing Users:

- Click the Change link in the Users row to load the change-list page for users.
- This page shows all users in the database, with options for filtering, sorting, and searching.
- Filtering: Options are on the right.
- Sorting: Click a column header.
- Search: Use the search box at the top.
- Click a username to see the edit form for that user.
- Change user attributes such as first/last names and permissions.
- To change a user's password, click the Change Password Form link.
- Different field types have different input widgets (e.g., calendar controls for date fields, checkboxes for Boolean fields).


127.0.0.1:8000/admin/auth/user/
Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Authentication and Authorization › Users
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
Select user to change
ADD USER +
Search
Action: --------- Go 0 of 2 selected
FILTER
↓ By staff status
All
Yes
No
↓ By superuser status
All
Yes
No
↓ By active
All
Yes
No

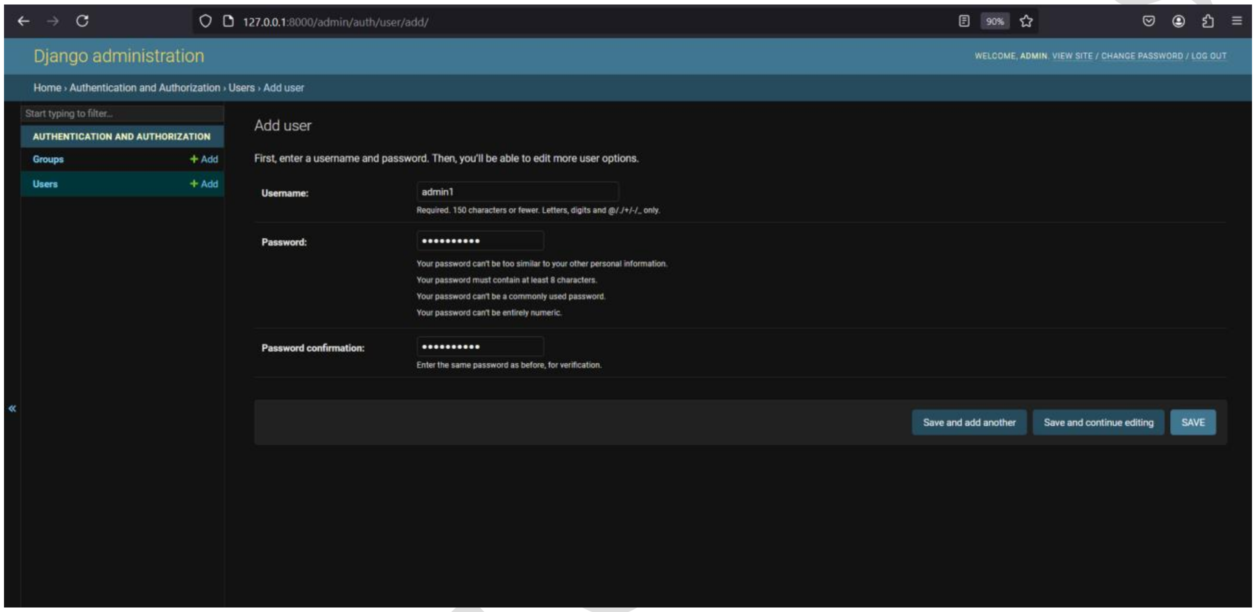
USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
admin				✓
hanu				✗

2 users

Figure: - The user change-list page

Adding and Deleting Records:

- Add Record: Click Add in the appropriate column on the admin home page to access an empty edit form for creating a new record.
- Delete Record: Click the Delete button at the bottom left of an edit form. Confirm the deletion on the subsequent page, which may list dependent objects to be deleted as well.


127.0.0.1:8000/admin/auth/user/add/
90%
Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Authentication and Authorization › Users › Add user
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
Save and add another
Save and continue editing
SAVE

Add user

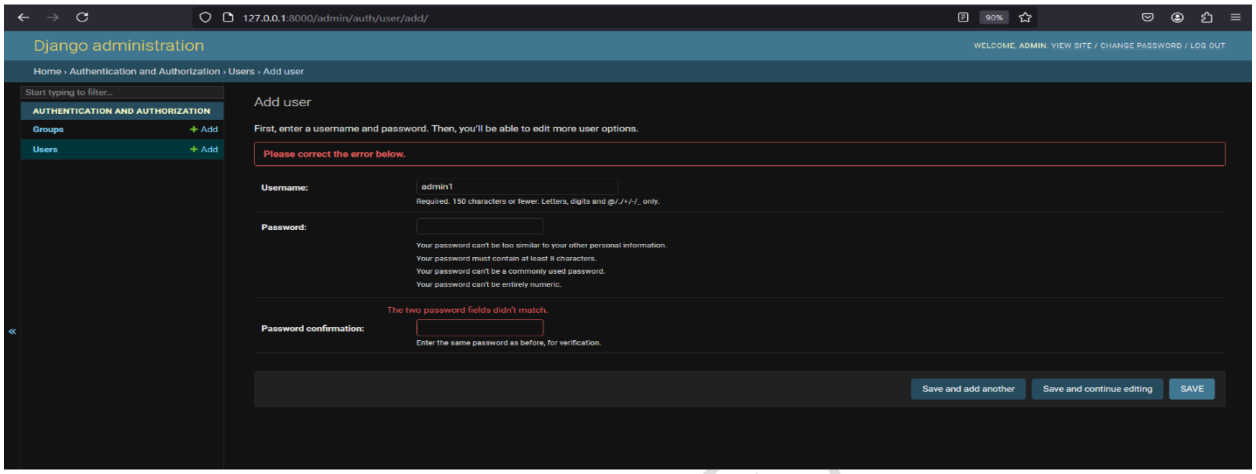
First, enter a username and password. Then, you'll be able to edit more user options.

Username:	admin1
	Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.
Password:	••••••••••
	Your password can't be too similar to your other personal information.
	Your password must contain at least 8 characters.
	Your password can't be a commonly used password.
	Your password can't be entirely numeric.
Password confirmation:	••••••••••
	Enter the same password as before, for verification.

Figure: - An Adding Records

Input Validation:

- The admin interface validates input automatically. Errors will be displayed if you leave required fields blank or enter invalid data.



127.0.0.1:8000/admin/auth/user/add/

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Authentication and Authorization › Users › Add user

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Please correct the error below.

Username: admin1

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

The two password fields didn't match.

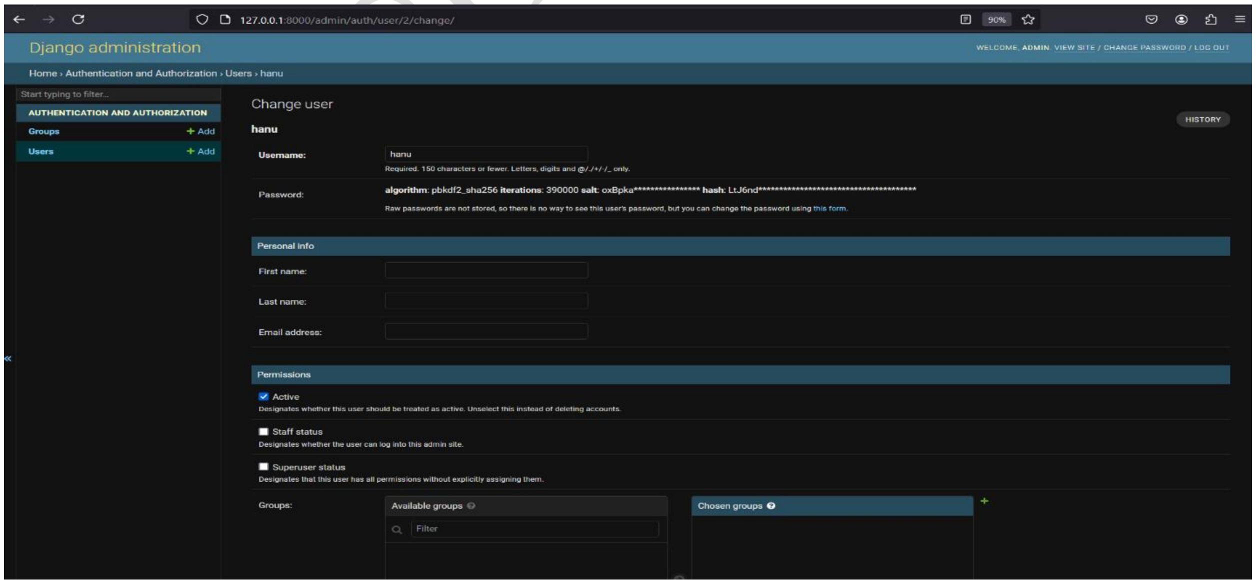
Password confirmation:

Enter the same password as before, for verification.

Figure: - An edit form displaying errors

History:

- When editing an object, a History link appears in the upper-right corner. This logs every change made through the admin interface and allows you to review the change history.



127.0.0.1:8000/admin/auth/user/2/change/

Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Authentication and Authorization › Users › hanu

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

Change user

hanu

Username: hanu

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: **algorithm**: pbkdf2_sha256 **iterations**: 390000 **salt**: oxBpka*************** **hash**: LtJ6nd*************************************

Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Personal info

First name:

Last name:

Email address:

Permissions

☑ Active

Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

☐ Staff status

Designates whether the user can log into this admin site.

☐ Superuser status

Designates that this user has all permissions without explicitly assigning them.

Groups: Available groups | Filter | Chosen groups

Figure: - An object history page

Customizing Admin Interfaces

- In the Django admin site, each field's label is derived from its model field name.
- To customize a label, use the verbose_name attribute in your model field definitions.

Example:

- To change the Author.email field's label to "e-mail":

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')
```

- Alternatively, you can pass verbose_name as a positional argument:

```
class Author(models.Model):

    first_name = models.CharField(max_length=30)

    last_name = models.CharField(max_length=40)

    email = models.EmailField('e-mail', blank=True)
```

Custom Model Admin Classes

- ModelAdmin classes allow customization of how models are displayed and managed in the admin interface.
- Customizing Change Lists
- By default, change lists show the result of the model's __str__ or __unicode__ method. You can specify which fields to display using the list_display attribute.

Example:

To display first_name, last_name, and email for the Author model:

```
from django.contrib import admin

from mysite.books.models import Author

class AuthorAdmin(admin.ModelAdmin):

    list_display = ('first_name', 'last_name', 'email')

admin.site.register(Author, AuthorAdmin)
```

- Adding a Search Bar
- Add search_fields to your AuthorAdmin:

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')
```

- Adding Date Filters
- Add list_filter to your BookAdmin:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
```

- Adding Date Hierarchy
- Add date_hierarchy to BookAdmin:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
```

- Changing Default Ordering
- Use ordering to set the default order of records:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
```

Customizing Edit Forms

- Changing Field Order
- Use the fields option to customize the order of fields:

```
class BookAdmin(admin.ModelAdmin):
    fields = ('title', 'authors', 'publisher', 'publication_date')
```

- Excluding Fields
- Exclude fields by omitting them from the fields list:

```
class BookAdmin(admin.ModelAdmin):
    fields = ('title', 'authors', 'publisher')
```

- Many-to-Many Field Widgets
- Use filter_horizontal for ManyToManyFields:

```
class BookAdmin(admin.ModelAdmin):
    filter_horizontal = ('authors',)
```

- Alternatively, use filter_vertical:

```
class BookAdmin(admin.ModelAdmin):
    filter_vertical = ('authors',)
```

- ForeignKey Field Widgets
- Use raw_id_fields for ForeignKey fields:

```
class BookAdmin(admin.ModelAdmin):
    raw_id_fields = ('publisher',)
```

Reasons to use Admin Interfaces.

When and Why to Use the Admin Interface—and When Not To

When to Use the Admin Interface

1. **For Non-Technical Users:**
   - **Data Entry:** The admin interface is designed to enable non-technical users to easily enter and manage data. For instance, reporters or content creators can input data without needing to know any code.
   - **Example Workflow:**
     1. A reporter meets with a developer to describe the data.
     2. The developer creates Django models based on this data and sets up the admin interface.
     3. The reporter reviews the admin site and suggests any changes to the fields.
     4. The developer updates the models accordingly.
     5. The reporter begins entering data, allowing the developer to focus on building views and templates.

2. **Inspecting Data Models:**
   - **Model Validation:** The admin interface is useful for developers to enter dummy data and validate their data models. This process can help identify modeling errors or inconsistencies early in development.

3. **Managing Acquired Data:**
   - **External Data Sources:** If your application relies on data from external sources (such as user inputs or web crawlers), the admin interface provides an easy way to inspect and edit this data. It acts as a convenient tool for data management, complementing your database's command-line utility.

4. Quick and Dirty Data-Management Apps:
    - **Lightweight Applications:** For personal projects or internal tools that don't require a polished public interface, the admin site can serve as a quick solution. For example, it can be used to track expenses or manage simple data sets, much like a relational spreadsheet.

When Not to Use the Admin Interface

1. **Public Interfaces:**
    - **Security and Usability:** The admin interface is not designed for public use. It lacks the security measures and user-friendly design necessary for a public-facing application.

2. Sophisticated Sorting and Searching:
    - **Advanced Data Handling:** While the admin site provides basic sorting and searching capabilities, it's not built for advanced data manipulation. For complex data queries and manipulations, custom views and tools are more appropriate.

3. **Complex User Interfaces:**
    - **Customization Limits:** The admin interface has limitations in terms of customization and interactivity. If your project requires a highly customized user interface with complex workflows, a custom-built solution will be more suitable.

Form Processing

1. **Introduction to Forms**
    - Django provides a built-in form handling functionality that simplifies the process of handling HTML forms in web applications.
    - Forms in Django are represented by Python classes that map to HTML form fields.

2. **Creating a Form**
    - In Django, forms are created by subclassing the forms.Form class or the forms.ModelForm class (for model-based forms).
    - Each form field is represented by a class attribute, and the type of the field is determined by the corresponding Django Field class.

Example:

```
from django import forms

class ContactForm(forms.Form):

    name = forms.CharField(max_length=100)

    email = forms.EmailField()

    message = forms.CharField(widget=forms.Textarea)
```

3. **Rendering a Form**
    - Forms can be rendered in templates using the {{ form.as_p }} (for paragraph-based rendering) or {{ form.as_table }} (for table-based rendering) template tags.
    - Individual form fields can be rendered using {{ form.field_name }}.

4. **Handling Form Data**
    - In the view function, you can access the submitted form data using the request.POST or request.GET dictionaries.
    - To bind the form data to the form instance, use form = MyForm(request.POST) or form = MyForm(request.GET).
    - After binding the data, you can check if the form is valid using form.is_valid().

5. **Validating Form Data**
    - Django provides built-in validation for common field types (e.g., EmailField, IntegerField, etc.).
    - You can define custom validation rules by overriding the clean_fieldname() method in your form class.
    - For complex validation rules, you can override the clean() method in your form class.

6. **Saving Form Data**
    - For forms not based on models, you can access the cleaned data using form.cleaned_data and handle it as needed.
    - For model-based forms (ModelForm), you can create or update model instances using form.save().

7. **Form Widgets**
    - Django provides a wide range of built-in form widgets (e.g., TextInput, Textarea, CheckboxInput, Select, etc.) for rendering form fields.
    - You can customize the rendering of form fields by specifying the widget argument when defining the field.

8. Form Handling in Views

- In the view function, you typically create a form instance, bind the data to it, and perform validation and data handling.
- After successful form submission, you can redirect the user to another page or render a success message.

9. CSRF Protection

- Django provides built-in protection against Cross-Site Request Forgery (CSRF) attacks by including a CSRF token in forms.
- You need to include the {% csrf_token %} template tag in your form template to generate the CSRF token.

10. Form Inheritance

- Django supports form inheritance, allowing you to create reusable form components and extend or override them as needed.
- You can use the Meta class to specify form-level attributes, such as labels, help_texts, and error_messages.

Creating Feedback forms

Step 1: Create the Feedback Form

Create a new file forms.py in your Django app directory, and add the following code

```
from django import forms

class FeedbackForm(forms.Form):

    name = forms.CharField(max_length=100, label='Your Name')

    email = forms.EmailField(label='Your Email')

    subject = forms.CharField(max_length=200, label='Subject')

    message = forms.CharField(widget=forms.Textarea, label='Your Feedback')
```

Step 2: Create the Feedback Template

Create a new file feedback.html in your Django app's templates directory, and add the following code:

```
<!DOCTYPE html>

<html>

<head>

    <title>Feedback Form</title>

</head>

<body>

    <h1>Feedback Form</h1>

    <form method="post">
```

```
{% csrf_token %}

{{ form.non_field_errors }}

<div>

   {{ form.name.errors }}

   {{ form.name.label_tag }}

   {{ form.name }}

</div>

<div>

   {{ form.email.errors }}

   {{ form.email.label_tag }}

   {{ form.email }}

</div>

<div>

   {{ form.subject.errors }}

   {{ form.subject.label_tag }}

   {{ form.subject }}

</div>

<div>

   {{ form.message.errors }}

   {{ form.message.label_tag }}

   {{ form.message }}
```

```
        </div>

        <input type="submit" value="Submit Feedback">

    </form>

</body>

</html>
```

Step 3: Create the Success Template

Create a new file feedback_success.html in your Django app's templates directory, and add the following code:

```
<!DOCTYPE html>

<html>

<head>

    <title>Feedback Submitted</title>

</head>

<body>

    <h1>Thank you for your feedback!</h1>

    <p>We appreciate your comments and will review them shortly.</p>

</body>

</html>
```

Step 4: Create the View Function

Open your Django app's views.py file and add the following code:

```
from django.shortcuts import render

from .forms import FeedbackForm


def feedback(request):

    if request.method == 'POST':

        form = FeedbackForm(request.POST)

        if form.is_valid():

            # Process the feedback data

            name = form.cleaned_data['name']

            email = form.cleaned_data['email']

            subject = form.cleaned_data['subject']

            message = form.cleaned_data['message']


            return render(request, 'feedback_success.html')

    else:

        form = FeedbackForm()


    return render(request, 'feedback.html', {'form': form})
```

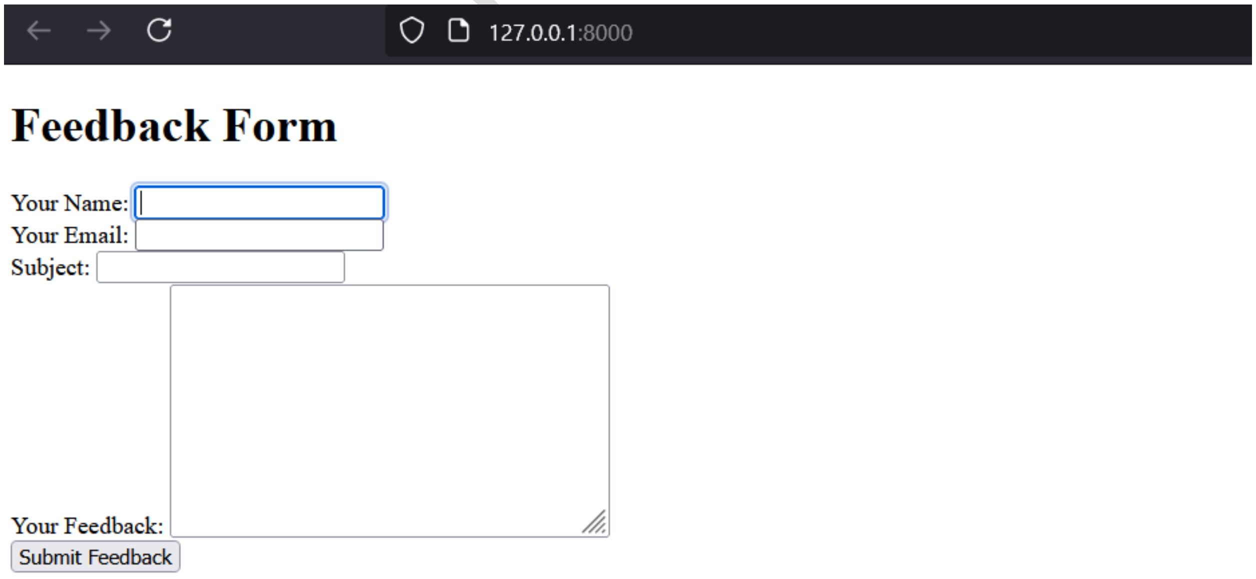
Step 5: Add URL Pattern

Open your Django app's urls.py file and add the following URL pattern:

```
from django.urls import path

from . import views



urlpatterns = [

    # ... other URL patterns

    path('feedback/', views.feedback, name='feedback'),

]
```

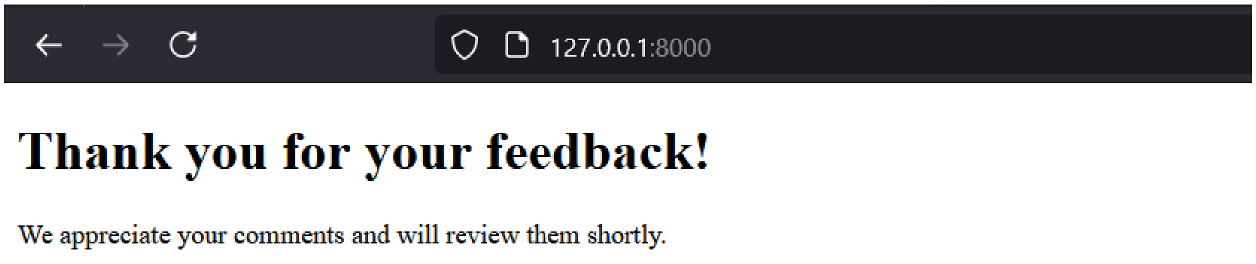
Step 6: Run the Development Server

Start the Django development server and navigate to **http://localhost:8000/feedback/** in your web browser. You should see the feedback form.



Feedback Form

Your Name:
Your Email:
Subject:
Your Feedback:



Thank you for your feedback!

We appreciate your comments and will review them shortly.

Form submissions

1. **Create a Form Class:** Define a form class that inherits from forms.Form or forms.ModelForm. This class defines the fields and validations for your form.

2. **Render the Form in a Template:** In your template, render the form using the {{ form }} template tag or individual field tags like {{ form.field_name }}.

3. **Check Request Method:** In your view function, check if the request method is POST (form submission) or GET (initial form load).

4. **Create Form Instance with Data:** If the request method is POST, create a form instance with the submitted data using form = YourForm(request.POST) or form = YourModelForm(request.POST).

5. **Validate the Form:** Call form.is_valid() to validate the form data against the defined fields and validations.

6. **Process Valid Form Data:** If the form is valid (form.is_valid() returns True), access the cleaned data using form.cleaned_data and perform any necessary operations (e.g., save to the database, send an email, etc.).

7. **Handle Invalid Form Data:** If the form is invalid (form.is_valid() returns False), re-render the form with error messages by passing the form instance to the template context.

8. **Redirect or Render Success Page:** After successful form processing, it's recommended to redirect the user to a success page or a different view to prevent duplicate form submissions on page refresh.

```
# forms.py

from django import forms


class ContactForm(forms.Form):

    name = forms.CharField(max_length=100)

    email = forms.EmailField()

    message = forms.CharField(widget=forms.Textarea)


# views.py

from django.shortcuts import render, redirect

from .forms import ContactForm


def contact(request):

    if request.method == 'POST':

        form = ContactForm(request.POST)

        if form.is_valid():
            # Process the form data
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']

            return redirect('success_url')
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})

<!-- contact.html -->
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Submit">
</form>
```

Custom Validation

1. **Define the Form Class:**
   - Use Django's forms.Form to define the fields of the form

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

2. **Add a Custom Validation Method:**
   - Create a clean_message() method to enforce the minimum word count.

```
def clean_message(self):
    message = self.cleaned_data['message']
    num_words = len(message.split())
    if num_words < 4:
        raise forms.ValidationError("Not enough words!")
    return message
```

- **This method:**
  - Extracts the message from self.cleaned_data.
  - Counts the words using split() and len().
  - Raises a ValidationError if there are fewer than four words.
  - Returns the cleaned message value.

Customizing Form Design

- You can customize the form's appearance using CSS and custom HTML templates for better control over the presentation.

1. **CSS for Error Styling:**
   - Define CSS to style error messages for better visibility

```
<style type="text/css">
   ul.errorlist {
      margin: 0;
      padding: 0;
   }
   .errorlist li {
      background-color: red;
      color: white;
      display: block;
      font-size: 10px;
      margin: 0 0 3px;
      padding: 4px 5px;
   }
</style>
```

2. **Custom HTML Template**:
   - Instead of using {{ form.as_table }}, manually render the form fields for finer control.

```
<html>

<head>

   <title>Contact us</title>

</head>
```

```
<body>

  <h1>Contact us</h1>

  {% if form.errors %}

  <p style="color: red;">

    Please correct the error{{ form.errors|pluralize }} below.

  </p>

  {% endif %}

  <form action="" method="post">

    <div class="field">

      {{ form.subject.errors }}

      <label for="id_subject">Subject:</label>

      {{ form.subject }}

    </div>

    <div class="field">

      {{ form.email.errors }}

      <label for="id_email">Your e-mail address:</label>

      {{ form.email }}

    </div>

    <div class="field">

      {{ form.message.errors }}

      <label for="id_message">Message:</label>
```

```
          {{ form.message }}

        </div>

        <input type="submit" value="Submit">

    </form>

</body>

</html>
```

3. **Advanced Error Handling in Template:**
    - Conditionally add classes and display error messages

```
<div class="field{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
    <ul>
        {% for error in form.message.errors %}
        <li><strong>{{ error }}</strong></li>
        {% endfor %}
    </ul>
    {% endif %}
    <label for="id_message">Message:</label>
    {{ form.message }}
</div>
```

This template:

- Checks for errors and displays them if present.
- Adds an errors class to the <div> if there are validation errors.
- Lists individual error messages in an unordered list.

Creating Model Forms

1. **Define the Model**
    - **Objective:** Create a Django model to represent the data structure.

Example

```
from django.db import models


class Contact(models.Model):
    subject = models.CharField(max_length=100)
    email = models.EmailField(blank=True)
    message = models.TextField()
    def __str__(self):
        return self.subject
```

2. **Create the Model Form**
    - Objective: Use forms.ModelForm to create a form based on the model.

Example

```
from django import forms
from .models import Contact
class ContactForm(forms.ModelForm):
    class Meta:
        model = Contact
        fields = ['subject', 'email', 'message']
```

3. **Add Custom Validation (Optional)**

   Objective: Add custom validation logic specific to form fields.

Example:

```
class ContactForm(forms.ModelForm):

    class Meta:

        model = Contact

        fields = ['subject', 'email', 'message']



    def clean_message(self):

        message = self.cleaned_data['message']

        num_words = len(message.split())

        if num_words < 4:

            raise forms.ValidationError("Not enough words!")

        return message
```

4. **Use the Form in Views**
   - Objective: Handle the form submission and validation within Django views.

Example

```
from django.shortcuts import render, redirect

from .forms import ContactForm



def contact_view(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('success')  # Redirect to a success page or another view
    else:
        form = ContactForm()
    return render(request, 'contact_form.html', {'form': form})
```

5. **Create the Template**
    - Objective: Design an HTML template to render and display the form.

Example

```
<html>
<head>
    <title>Contact Us</title>
</head>
<body>
    <h1>Contact Us</h1>
    {% if form.errors %}
    <p style="color: red;">
        Please correct the error{{ form.errors|pluralize }} below.
```

```
</p>

{% endif %}

<form method="post">

    {% csrf_token %}

    <div class="field">

        {{ form.subject.errors }}

        <label for="id_subject">Subject:</label>

        {{ form.subject }}

    </div>

    <div class="field">

        {{ form.email.errors }}

        <label for="id_email">Your email address:</label>

        {{ form.email }}

    </div>

    <div class="field">

        {{ form.message.errors }}

        <label for="id_message">Message:</label>

        {{ form.message }}

    </div>

    <input type="submit" value="Submit">

</form>
```

```
</body>

</html>
```

1. **Define the Model:** Establish your data structure with a Django model.
2. **Create the Model Form:** Generate a form using forms.ModelForm based on the model.
3. **Add Custom Validation:** Optionally include custom validation methods within the form.
4. **Use the Form in Views:** Implement form handling logic in Django views to process submissions.
5. **Create the Template:** Design an HTML template to display and manage the form interface.

URLConf Ticks

Streamlining Function Imports

1. **Traditional Method: Direct Import of View Functions**

   Example:

```
from django.conf.urls.defaults import *
from mysite.views import hello, current_datetime, hours_ahead
urlpatterns = patterns('',
    (r'^hello/$', hello),
    (r'^time/$', current_datetime),
    (r'^time/plus/(\d{1,2})/$', hours_ahead),
)
```

   - **Disadvantage:** Tedious to manage imports as the application grows.

2. **Importing the Views Module**

Example

```
from django.conf.urls.defaults import *
from mysite import views
urlpatterns = patterns('',
    (r'^hello/$', views.hello),
    (r'^time/$', views.current_datetime),
    (r'^time/plus/(d{1,2})/$', views.hours_ahead),
)
```

- **Advantage:** Simplifies imports, but still requires module import.

3. **Using Strings to Specify View Functions**

Example

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    (r'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(d{1,2})/$', 'mysite.views.hours_ahead'),
)
```

- **Advantage:** No need to import view functions; Django handles imports automatically.

4. **Factoring Out a Common View Prefix**

Example:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',

    (r'^hello/$', 'hello'),

    (r'^time/$', 'current_datetime'),

    (r'^time/plus/(d{1,2})/$', 'hours_ahead'),

)
```

- **Advantage:** Reduces redundancy by factoring out common prefixes.

Choosing Between Methods

- **Advantages of the String Approach:**
  - More compact, no need to import view functions explicitly.
  - More readable and manageable for projects with views in multiple modules.

- **Advantages of the Function Object Approach:**
  - Facilitates easy wrapping of view functions.
  - More "Pythonic," aligning with Python traditions of passing functions as objects.

- **Flexibility:**
  - Both approaches are valid and can be mixed within the same URLconf depending on personal coding style and project needs.

Including Other URLConfs

Purpose and Benefit

1. Purpose: Allows for modular organization of URL patterns by "including" URLconf modules from different parts of the project.
2. Benefit: Enhances reusability and maintainability across multiple Django-based sites.

Basic Usage

- Example URLconf that includes other URLconfs.

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    (r'^weblog/', include('mysite.blog.urls')),
    (r'^photos/', include('mysite.photos.urls')),
    (r'^about/$', 'mysite.views.about'),
)
```

Important Considerations

1. No End-of-String Match Character: Regular expressions pointing to an include() should not have a $ but should include a trailing slash.
2. URL Stripping: When Django encounters include(), it removes the matched part of the URL and sends the remaining string to the included URLconf.

Example Breakdown

- Main URLconf:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',

    (r'^weblog/', include('mysite.blog.urls')),

    (r'^photos/', include('mysite.photos.urls')),

    (r'^about/$', 'mysite.views.about'),

)
```

- Included URLconf (mysite.blog.urls):

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
    (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

Sample Request Handling

1. **Request: /weblog/2007/**
    - First URLconf: r'^weblog/' matches.
    - Action: Strips weblog/.
    - Remaining URL: 2007/.
    - Result: Matches r'^(\d\d\d\d)/$' in mysite.blog.urls.

2. **Request: /weblog//2007/ (with two slashes)**
    - First URLconf: r'^weblog/' matches.
    - Action: Strips weblog/.
    - Remaining URL: /2007/ (with leading slash).
    - Result: Does not match any patterns in mysite.blog.urls.

3. **Request: /about/**
    - First URLconf: Matches r'^about/$'.
    - Result: Maps to mysite.views.about view.

Mixing Patterns

- Flexibility: You can mix include() patterns with non-include() patterns within the same URLconf.