Django Forms: Comprehensive Guide with Examples

Overview

Django provides two main ways to handle forms:

1. **Django Forms**: Manually define each field and its validation.
2. **Django ModelForms**: Automatically generate fields based on a Django model.

This guide includes:

- Examples for all form fields with widgets and configurations.
- A complete explanation of widgets and their customization.
- Examples of both normal forms and ModelForms with corresponding views and HTML templates.

Scenario: Library Book Collection Form

For this example, we'll create forms to manage a library's book collection.

Form Fields and Widgets with Examples

Field Name	Description	Arguments	Example Code
`CharField`	For input of strings.	`max_length`, `min_length`, `strip`, `widget`	`forms.CharField(max_length=100, widget=forms.TextInput(attrs={'placeholder': 'Book Title'}))`
`IntegerField`	For input of integers.	`min_value`, `max_value`, `widget`	`forms.IntegerField(min_value=1, max_value=1000)`
`FloatField`	For input of floating-point numbers.	`min_value`, `max_value`, `widget`	`forms.FloatField(min_value=0.0)`
`EmailField`	For input of email addresses.	`max_length`, `min_length`, `widget`	`forms.EmailField(widget=forms.EmailInput(attrs={'placeholder': 'Email Address'}))`
`URLField`	For input of URLs.	`max_length`, `min_length`, `widget`	`forms.URLField()`
`BooleanField`	Checkbox for True/False values.	`required`, `widget`	`forms.BooleanField(required=False)`
`ChoiceField`	Dropdown for a set of choices.	`choices`, `widget`	`forms.ChoiceField(choices=[('fiction', 'Fiction'), ('nonfiction', 'Non-Fiction')])`
`FileField`	For uploading files.	`widget`	`forms.FileField()`
`DateField`	For input of dates.	`input_formats`, `widget`	`forms.DateField(widget=forms.SelectDateWidget())`

Example Code for a Django Form

```
from django import forms

class PizzaOrderForm(forms.Form):
    name = forms.CharField(
        max_length=100,
        widget=forms.TextInput(attrs={'placeholder': 'Your Name'})
    )
```

```
    email = forms.EmailField(
        widget=forms.EmailInput(attrs={'placeholder': 'Your Email'})
    )
    size = forms.ChoiceField(
        choices=[
            ('small', 'Small'),
            ('medium', 'Medium'),
            ('large', 'Large')
        ],
        widget=forms.RadioSelect
    )
    toppings = forms.MultipleChoiceField(
        choices=[
            ('pepperoni', 'Pepperoni'),
            ('mushrooms', 'Mushrooms'),
            ('onions', 'Onions'),
            ('sausage', 'Sausage'),
            ('bacon', 'Bacon'),
            ('extra_cheese', 'Extra Cheese'),
            ('black_olives', 'Black Olives'),
            ('green_peppers', 'Green Peppers'),
            ('pineapple', 'Pineapple'),
            ('spinach', 'Spinach')
        ],
        widget=forms.CheckboxSelectMultiple
    )
    special_requests = forms.CharField(
        required=False,
        widget=forms.Textarea(attrs={
            'placeholder': 'Any special requests?',
            'rows': 3
        })
    )
```

```
from django.http import HttpResponse
from django.shortcuts import render
from .forms import PizzaOrderForm

def pizza_order_view(request):
    if request.method == "POST":
        form = PizzaOrderForm(request.POST)
        if form.is_valid():
            # Daten aus dem Formular
            order_data = form.cleaned_data
            response_message = f"""
                <h1>Order Received!</h1>
                <p><strong>Name:</strong> {order_data['name']}</p>
                <p><strong>Email:</strong> {order_data['email']}</p>
                <p><strong>Pizza Size:</strong> {order_data['size']}</p>
                <p><strong>Toppings:</strong> {', '.join(order_data['toppings'])}</p>
                <p><strong>Special Requests:</strong> {order_data['special_requests'] or 'None'}
</p>
            """
            return HttpResponse(response_message)
    else:
        form = PizzaOrderForm()
    return render(request, "pizza_order_form.html", {"form": form})
```

HTML

´´´html

Order Your Pizza

{% csrf_token %} {{ form.as_p }} Submit Order
```

## Widget Customizations

Widgets define the HTML representation of form fields. Here are some examples of commonly used widgets:

| Widget Name | Description | Example Code |
|---|---|---|
| `TextInput` | Standard single-line text input. | `forms.CharField(widget=forms.TextInput(attrs={'placeholder': 'Enter value'}))` |
| `Textarea` | Multi-line text input. | `forms.CharField(widget=forms.Textarea(attrs={'rows': 5, 'cols': 40}))` |
| `Select` | Dropdown menu. | `forms.ChoiceField(widget=forms.Select())` |
| `CheckboxInput` | Checkbox for boolean values. | `forms.BooleanField(widget=forms.CheckboxInput())` |
| `PasswordInput` | Input for password (hidden characters). | `forms.CharField(widget=forms.PasswordInput())` |

## Example: Normal Form with View and Template

Model

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
    genre = models.CharField(max_length=20, choices=[
        ('fiction', 'Fiction'),
        ('nonfiction', 'Non-Fiction')
    ])
    published_date = models.DateField()
    pages = models.PositiveIntegerField()

    def __str__(self):
        return self.title
```

Form

```
from django import forms

class LibraryBookForm(forms.Form):
    title = forms.CharField(max_length=100, widget=forms.TextInput(attrs={'placeholder': 'Book Title'}))
    author = forms.CharField(max_length=50, widget=forms.TextInput(attrs={'placeholder': 'Author Name'}))
    genre = forms.ChoiceField(choices=[('fiction', 'Fiction'), ('nonfiction', 'Non-Fiction')], widget=forms.RadioSelect)
    published_date = forms.DateField(widget=forms.SelectDateWidget(years=range(1900, 2024)))
    pages = forms.IntegerField(min_value=1, max_value=5000)
```

View

```
from django.shortcuts import render, redirect
from .models import Book
```
```

```python
from .forms import LibraryBookForm

def library_book_view(request):
    if request.method == "POST":
        form = LibraryBookForm(request.POST)
        if form.is_valid():
            # Save the form data into the model
            book = Book(
                title=form.cleaned_data['title'],
                author=form.cleaned_data['author'],
                genre=form.cleaned_data['genre'],
                published_date=form.cleaned_data['published_date'],
                pages=form.cleaned_data['pages']
            )
            book.save()
            return render(request, "library_success.html", {"data": form.cleaned_data})
    else:
        form = LibraryBookForm()
    return render(request, "library_form.html", {"form": form})
```

Template (library_form.html)

```html
<!DOCTYPE html>
<html>
<head>
    <title>Library Book Form</title>
</head>
<body>
    <h1>Enter Book Details</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Template (library_success.html)

```html
<!DOCTYPE html>
<html>
<head>
    <title>Form Submission Successful</title>
</head>
<body>
    <h1>Book Details Submitted Successfully</h1>
    <ul>
        {% for key, value in data.items %}
            <li><strong>{{ key }}:</strong> {{ value }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Unterschied zwischen Django Forms und Django ModelForms

Django Form

Definition:

`django.forms.Form` ist eine grundlegende Klasse, die für die Erstellung und Validierung von Formularen verwendet wird, unabhängig von einem Datenmodell.

Einsatz:

Django Forms werden verwendet, wenn:

- Das Formular **nicht direkt mit einem Datenmodell** (z. B. einer Datenbank) verknüpft ist.
- Eine benutzerdefinierte Validierung oder zusätzliche Felder benötigt wird, die nicht in einem Datenmodell enthalten sind.

Eigenschaften:

- Felder müssen **manuell definiert** werden.
- Flexibel, aber erfordert **mehr Boilerplate-Code**.
- Wird verwendet, um **beliebige Daten** zu verarbeiten (z. B. Daten, die nicht in der Datenbank gespeichert werden).

Vorteile:

1. Kann für beliebige Datenquellen verwendet werden.
2. Ideal für Formulare, die keine direkte Verbindung zu einem Model haben.
3. Flexibilität bei der Definition von Validierungslogik und zusätzlichen Feldern.

Nachteile:

- Mehr manueller Aufwand, da die Felder und Validierungen vollständig definiert werden müssen.

Beispiel:

Ein Kontaktformular, das nur die Eingaben validiert, ohne die Daten in der Datenbank zu speichern:

```
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)

# Beispielansicht in einer View:
from django.shortcuts import render
from django.http import HttpResponse

def contact_view(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Verarbeitung der Daten
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            message = form.cleaned_data['message']
            return HttpResponse("Vielen Dank für Ihre Nachricht!")
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

Django ModelForm

Definition:

`django.forms.ModelForm` ist eine Unterklasse von `Form`, die direkt mit einem Datenmodell (`django.db.models.Model`) verknüpft ist.

Einsatz:

ModelForms werden verwendet, wenn:

- Das Formular direkt mit einem Datenmodell verknüpft ist.
- Die Eingaben des Benutzers **in einer Datenbank gespeichert** werden sollen.

Eigenschaften:

- Die Felder werden automatisch basierend auf den **Feldern des verknüpften Models** generiert.
- Standardvalidierungen aus dem Model (z. B. `max_length`, `required`) werden automatisch übernommen.
- **Schnelle Entwicklung**, da wenig Boilerplate-Code erforderlich ist.

Vorteile:

1. Automatische Verbindung zwischen Formular und Datenbank.
2. Spart Zeit, da die Felder und Validierungen aus dem Model übernommen werden.
3. Ideal für datenbankbasierte CRUD-Anwendungen.

Nachteile:

- Weniger flexibel, da es auf das verknüpfte Model beschränkt ist.
- Zusätzliche Felder oder Validierungslogik erfordern mehr Anpassungen.

Beispiel:

Ein Kontaktformular, das die Eingaben in einem `Contact` Model speichert:

```
from django import forms
from .models import Contact

class ContactModelForm(forms.ModelForm):
    class Meta:
        model = Contact
        fields = ['name', 'email', 'message']

# Model:
from django.db import models

class Contact(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    message = models.TextField()

# Beispielansicht in einer View:
from django.shortcuts import render
from django.http import HttpResponse

def contact_view(request):
    if request.method == 'POST':
        form = ContactModelForm(request.POST)
        if form.is_valid():
            form.save()  # Speichert die Daten automatisch in der Datenbank
            return HttpResponse("Vielen Dank für Ihre Nachricht!")
    else:
        form = ContactModelForm()
    return render(request, 'contact.html', {'form': form})
```

Unterschied zwischen Django Form und ModelForm

Aspekt	Form	ModelForm
Abhängigkeit	Unabhängig von einem Model	Verknüpft mit einem Django Model
Flexibilität	Sehr flexibel, benötigt mehr Arbeit	Automatisch generiert, weniger flexibel

Aspekt	Form	ModelForm
Verwendung	Für beliebige Datenquellen	Für datenbankbasierte Anwendungen
Validierung	Muss manuell hinzugefügt werden	Automatisch aus dem Model übernommen

---

Konkrete Anwendungsbeispiele

Wann verwende ich **Django Forms**?

- **Kontaktformulare:** Wenn die Eingaben nicht gespeichert, sondern nur verarbeitet werden (z. B. E-Mail-Versand).
- **Dynamische Daten:** Wenn die Eingaben von Benutzern verwendet werden, um andere dynamische Aktionen auszuführen (z. B. eine Suchanfrage).
- **Benutzerdefinierte Felder:** Wenn das Formular Felder enthält, die nicht in einem Model abgebildet sind.

Wann verwende ich **Django ModelForms**?

- **CRUD-Anwendungen:** Zum Erstellen, Bearbeiten und Löschen von Objekten in der Datenbank.
- **Automatisierung:** Wenn die Felder des Formulars direkt den Feldern eines Models entsprechen.
- **Schnelle Prototypen:** Um schnell ein datenbankgestütztes Formular zu erstellen.

Wie füge ich einen Placeholder zu einem ModelForm hinzu?

Um einem Feld (z. B. einem `EmailField`) in einer ModelForm einen Placeholder hinzuzufügen, gibt es zwei Methoden:

1. Mit `Meta.widgets`:

```
class ContactModelForm(forms.ModelForm):
    class Meta:
        model = Contact
        fields = ['name', 'email', 'message']
        widgets = {
            'email': forms.EmailInput(attrs={'placeholder': 'yourname@example.com'})
        }
```

2. Durch Überschreiben des Feldes:

```
class ContactModelForm(forms.ModelForm):
    email = forms.EmailField(widget=forms.EmailInput(attrs={'placeholder':
'yourname@example.com'}))

    class Meta:
        model = Contact
        fields = ['name', 'email', 'message']
```

Fazit

- Verwende **Django Forms**, wenn du die volle Kontrolle über dein Formular benötigst und es nicht direkt an ein Model gebunden ist.
- Verwende **Django ModelForms**, wenn dein Formular direkt mit einem Datenmodell verknüpft ist und du Zeit sparen möchtest.

Beide Ansätze bieten mächtige Werkzeuge, um benutzerfreundliche und robuste Formulare in Django zu erstellen.

Example: ModelForm with View and Template

Model

```
```

```
from django.db import models

class Event(models.Model):
    CATEGORY_CHOICES = [
        ('workshop', 'Workshop'),
        ('seminar', 'Seminar'),
        ('conference', 'Conference'),
    ]

    title = models.CharField(max_length=200)
    date_time = models.DateTimeField()
    location = models.CharField(max_length=255)
    description = models.TextField()
    max_participants = models.PositiveIntegerField()
    category = models.CharField(max_length=50, choices=CATEGORY_CHOICES)

    def __str__(self):
        return self.title
```

ModelForm

```
from django import forms
from .models import Event

class EventForm(forms.ModelForm):
    class Meta:
        model = Event
        fields = ['title', 'date_time', 'location', 'description', 'max_participants',
'category']
        widgets = {
            'description': forms.Textarea(attrs={'rows': 5, 'placeholder': 'Describe the
event...'}),
            'date_time': forms.DateTimeInput(attrs={'type': 'datetime-local'}),
        }
```

View

```
from django.shortcuts import render, redirect
from .models import Event
from .forms import EventForm

def event_create_view(request):
    if request.method == "POST":
        form = EventForm(request.POST)
        if form.is_valid():
            form.save()  # Save the event to the database
            return redirect('event_success')  # Redirect to a success page
    else:
        form = EventForm()
    return render(request, "event_form.html", {"form": form})

def event_success_view(request):
    return render(request, "event_success.html")
```

Template (event_form.html)

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>Create Event</title>
</head>
<body>
    <h1>Create a New Event</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Template (event_success.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Created</title>
</head>
<body>
    <h1>Event Created Successfully!</h1>
    <p>Your event has been added to the system.</p>
    <a href="/">Go back to home</a>
</body>
</html>
```

Conclusion

This guide includes:

- Detailed examples of Django Form fields and widgets.
- Comprehensive comparison between Forms and ModelForms.
- Complete examples with views and templates for normal forms and ModelForms.

Feel free to expand this example by adding more fields, custom validations, and styling.