

DJANGO CHEAT SHEET — by CosmicLace

MODELS

♦ What are Models?

Models are Python classes that define the structure of your database. Each model class represents a database table, and each attribute represents a database field. Django's ORM (Object-Relational Mapper) handles the translation between Python objects and database records.

♦ Database Normalization & Foreign Keys

Database normalization is the process of structuring a relational database to reduce data redundancy and improve data integrity. It involves dividing large tables into smaller, related tables and defining relationships between them.

Key benefits:

- Eliminates data duplication
- Prevents update anomalies
- Ensures data consistency
- Improves query performance (in most cases)

Normalization forms:

- **1NF**: Eliminate repeating groups, create separate tables for related data
- **2NF**: Remove partial dependencies (attributes that depend on only part of primary key)
- **3NF**: Remove transitive dependencies (attributes that depend on non-key attributes)

Foreign keys are the mechanism that implements these relationships:

- A foreign key is a field (or collection of fields) in one table that refers to the primary key in another table
- They enforce referential integrity (preventing orphaned records)

♦ In Django, **ForeignKey**, **ManyToManyField**, and **OneToOneField** create these relationships:

1. ForeignKey Example (Many-to-One)

```
# models.py
from django.db import models

class Author(models.Model):
```

```

name = models.CharField(max_length=100)
bio = models.TextField()

def __str__(self):
    return self.name

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE, related_name='books')
    published_date = models.DateField()

    def __str__(self):
        return self.title

# Usage
# Create an author and multiple books
author = Author.objects.create(name="J.K. Rowling")
Book.objects.create(title="Harry Potter 1", author=author, published_date="1997-06-26")
Book.objects.create(title="Harry Potter 2", author=author, published_date="1998-07-02")

# Query related objects
author = Author.objects.get(name="J.K. Rowling")
author_books = author.books.all() # Access books via related_name

```

2. ManyToManyField Example (Many-to-Many)

```

# models.py
class Tag(models.Model):
    name = models.CharField(max_length=50)

    def __str__(self):
        return self.name

class Article(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    tags = models.ManyToManyField(Tag, related_name='articles')

    def __str__(self):
        return self.title

# Usage
# Create tags and articles
python_tag = Tag.objects.create(name="Python")
django_tag = Tag.objects.create(name="Django")

article = Article.objects.create(title="Django ORM Tips")

# Add relationships
article.tags.add(python_tag, django_tag)

```

```
# Or create from the other side
new_article = Article.objects.create(title="Python Best Practices")
python_tag.articles.add(new_article)

# Query
django_articles = django_tag.articles.all()
article_tags = article.tags.all()
```

3. OneToOneField Example (One-to-One)

```
# models.py
class User(models.Model):
    username = models.CharField(max_length=100)
    email = models.EmailField()

    def __str__(self):
        return self.username

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, related_name='profile')
    bio = models.TextField(blank=True)
    birth_date = models.DateField(null=True, blank=True)
    avatar = models.ImageField(upload_to='avatars/', null=True, blank=True)

    def __str__(self):
        return f"Profile for {self.user.username}"

# Usage
# Create user and profile
user = User.objects.create(username="johndoe", email="john@example.com")
Profile.objects.create(user=user, bio="Python developer and blogger")

# Access related objects (direct reference in both directions)
user = User.objects.get(username="johndoe")
user_bio = user.profile.bio # Access profile from user

profile = Profile.objects.get(user__username="johndoe")
username = profile.user.username # Access user from profile
```

◆ Defining Models

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User

class Book(models.Model):
    # Basic fields with common options
    title = models.CharField(max_length=200)
    subtitle = models.CharField(max_length=200, blank=True, null=True)
```

```

description = models.TextField(help_text="Full description of the book")
page_count = models.IntegerField(default=0)
price = models.DecimalField(max_digits=6, decimal_places=2)
created_at = models.DateTimeField(auto_now_add=True) # Set once on creation
updated_at = models.DateTimeField(auto_now=True) # Updated on each save
published_date = models.DateField(null=True, blank=True)

# Relationship fields
author = models.ForeignKey('Author', on_delete=models.CASCADE, related_name='books')
co_authors = models.ManyToManyField('Author', related_name='co_authored_books')
publisher = models.ForeignKey('Publisher', on_delete=models.SET_NULL, null=True)

# Choice fields
GENRE_CHOICES = [
    ('FIC', 'Fiction'),
    ('NON', 'Non-Fiction'),
    ('SCI', 'Science Fiction'),
    ('MYS', 'Mystery'),
]
genre = models.CharField(max_length=3, choices=GENRE_CHOICES, default='FIC')

# Boolean and status fields
is_bestseller = models.BooleanField(default=False)
in_stock = models.BooleanField(default=True)

# Other useful fields
slug = models.SlugField(unique=True)
isbn = models.CharField(max_length=13, unique=True)
cover_image = models.ImageField(upload_to='covers/', blank=True)
sample_pdf = models.FileField(upload_to='samples/', blank=True)

# Special fields
metadata = models.JSONField(default=dict, blank=True)
uuid = models.UUIDField(default=uuid.uuid4, editable=False)

class Meta:
    ordering = ['-published_date']
    indexes = [
        models.Index(fields=['title']),
        models.Index(fields=['genre', 'published_date']),
    ]
    constraints = [
        models.UniqueConstraint(fields=['title', 'author'], name='unique_book_per_a')
    ]
    verbose_name = 'Book'
    verbose_name_plural = 'Books'

def __str__(self):
    return self.title

def save(self, *args, **kwargs):

```

```
# Custom save behavior example
if not self.slug:
    self.slug = slugify(self.title)
super().save(*args, **kwargs)

def get_absolute_url(self):
    from django.urls import reverse
    return reverse('book-detail', args=[str(self.id)])

def is_recent(self):
    return self.published_date >= timezone.now().date() - timezone.timedelta(days=3)
```

♦ Field Types

Text Fields

- `CharField`: String field with maximum length
- `TextField`: Unlimited text field
- `SlugField`: For URL-friendly strings
- `EmailField`: Validates as email address
- `URLField`: Validates as URL
- `FilePathField`: For selecting files from a directory

Number Fields

- `IntegerField`: Integer values
- `PositiveIntegerField` / `PositiveSmallIntegerField`: Positive integers only
- `SmallIntegerField` / `BigIntegerField`: Different size integers
- `FloatField`: Floating-point numbers
- `DecimalField`: Fixed-precision decimal numbers

Boolean Fields

- `BooleanField`: True/False field
- `NullBooleanField`: True/False/None field

Date/Time Fields

- `DateField`: Date values
- `TimeField`: Time values
- `DateTimeField`: Date and time values
- `DurationField`: Stores periods of time

Binary Fields

- `BinaryField` : Raw binary data
- `FileField` : File upload field
- `ImageField` : Image upload field with validation

Relationship Fields

- `ForeignKey` : Many-to-one relationship
- `ManyToManyField` : Many-to-many relationship
- `OneToOneField` : One-to-one relationship

Special Fields

- `AutoField` / `BigAutoField` : Auto-incrementing primary key
- `UUIDField` : Stores UUID values
- `JSONField` : Stores JSON-encoded data
- `GenericIPAddressField` : IP address storage
- `ArrayField` : PostgreSQL array field (PostgreSQL only)
- `HStoreField` : Key-value store (PostgreSQL only)

◆ Relationship Fields in Detail

1. ForeignKey (Many-to-One)

A many-to-one relationship where many instances of this model can point to one instance of the related model.

```
# Basic ForeignKey
author = models.ForeignKey('Author', on_delete=models.CASCADE)

# With related_name (to reference books from author)
author = models.ForeignKey(
    'Author',
    on_delete=models.CASCADE,
    related_name='books' # author.books.all()
)

# With on_delete options
publisher = models.ForeignKey(
    'Publisher',
    on_delete=models.SET_NULL, # Set to NULL if publisher is deleted
    null=True,
    blank=True
)

# Other on_delete options:
# models.CASCADE - Delete book when author is deleted
```

```

# models.PROTECT - Prevent deletion of author if they have books
# models.SET_NULL - Set to NULL (requires null=True)
# models.SET_DEFAULT - Set to default value (requires default)
# models.SET(callable) - Set to value returned by callable
# models.DO_NOTHING - Do nothing (may cause DB integrity issues)

# Self-referential ForeignKey
parent_chapter = models.ForeignKey(
    'self',
    on_delete=models.CASCADE,
    null=True,
    blank=True,
    related_name='sub_chapters'
)

# With db_constraint=False (no DB enforced constraint)
reference = models.ForeignKey(
    'Book',
    on_delete=models.SET_NULL,
    null=True,
    db_constraint=False
)

# Using + in related_name to avoid reverse relation
editor = models.ForeignKey(
    'Editor',
    on_delete=models.CASCADE,
    related_name='+' # No reverse relation
)

```

2. ManyToManyField (Many-to-Many)

A many-to-many relationship where multiple instances of this model can be related to multiple instances of another model.

```

# Basic ManyToManyField
categories = models.ManyToManyField('Category')

# With a related_name
tags = models.ManyToManyField(
    'Tag',
    related_name='books' # tag.books.all()
)

# With a through model (for extra fields on the relationship)
class BookAuthor(models.Model):
    book = models.ForeignKey('Book', on_delete=models.CASCADE)
    author = models.ForeignKey('Author', on_delete=models.CASCADE)
    contribution_percentage = models.IntegerField(default=100)
    is_primary = models.BooleanField(default=True)
    date_joined = models.DateField(auto_now_add=True)

```

```

class Meta:
    unique_together = ['book', 'author']

# Then in Book model:
authors = models.ManyToManyField(
    'Author',
    through='BookAuthor',
    related_name='authored_books'
)

# Self-referential ManyToManyField
related_books = models.ManyToManyField('self', blank=True)

# Symmetrical=False for directed relationships in self-referential M2M
influenced_by = models.ManyToManyField(
    'self',
    symmetrical=False,
    related_name='influences',
    blank=True
)

# Using a custom through model with self-referential M2M
class BookRelationship(models.Model):
    from_book = models.ForeignKey('Book', on_delete=models.CASCADE, related_name='relat
    to_book = models.ForeignKey('Book', on_delete=models.CASCADE, related_name='related
    relationship_type = models.CharField(max_length=20, choices=[
        ('sequel', 'Sequel'),
        ('prequel', 'Prequel'),
        ('adaptation', 'Adaptation')
    ])
    created_at = models.DateTimeField(auto_now_add=True)

# Then in Book model:
related_to = models.ManyToManyField(
    'self',
    through='BookRelationship',
    symmetrical=False,
    through_fields=('from_book', 'to_book')
)

```

3. OneToOneField (One-to-One)

A one-to-one relationship that can be used to extend another model or create a strict one-to-one relation.

```

# Basic OneToOneField (e.g., for model extension)
class BookDetail(models.Model):
    book = models.OneToOneField(
        'Book',
        on_delete=models.CASCADE,

```



```

        primary_key=True,
        related_name='detail'
    )
    first_published = models.DateField(null=True, blank=True)
    edition_history = models.TextField(blank=True)
    print_run = models.IntegerField(default=0)

    def __str__(self):
        return f"Details for {self.book.title}"

# With parent_link=True (for model inheritance)
class EBook(Book):
    book_ptr = models.OneToOneField(
        Book,
        on_delete=models.CASCADE,
        parent_link=True,
        primary_key=True
    )
    file_size_mb = models.DecimalField(max_digits=6, decimal_places=2)
    download_url = models.URLField()
    drm_protected = models.BooleanField(default=True)

```

◆ Model Inheritance

1. Abstract Base Classes

Creating a parent class that holds common fields but isn't created as a table.

```

class BaseItem(models.Model):
    title = models.CharField(max_length=100)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True # No table created for this model

class Book(BaseItem):
    author = models.ForeignKey('Author', on_delete=models.CASCADE)
    # Book has title, created_at, updated_at from BaseItem

class Article(BaseItem):
    publication = models.CharField(max_length=100)
    # Article has title, created_at, updated_at from BaseItem

```

2. Multi-table Inheritance

Each model is a separate table with foreign key relationships.

```

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)

```

```

class Book(Product): # Implicitly creates a OneToOneField to Product
    author = models.CharField(max_length=100)
    isbn = models.CharField(max_length=13)

class Electronics(Product):
    brand = models.CharField(max_length=100)
    warranty_years = models.IntegerField(default=1)

```

3. Proxy Models

Allows changing behavior without changing database structure.

```

class Book(models.Model):
    title = models.CharField(max_length=100)
    published_date = models.DateField()

class RecentBook(Book):
    class Meta:
        proxy = True # Uses same table as Book

    def is_recent(self):
        return self.published_date >= timezone.now().date() - timezone.timedelta(days=3)

    @classmethod
    def get_recent(cls):
        return cls.objects.filter(published_date__gte=timezone.now().date() - timezone.

```

◆ Meta Options

The `Meta` class inside a model provides additional settings:

```

class Book(models.Model):
    # fields...

    class Meta:
        # Database table naming
        db_table = 'bookstore_books'

        # Ordering
        ordering = ['-published_date', 'title']

        # Constraints
        unique_together = [['title', 'author']] # OR
        constraints = [
            models.UniqueConstraint(fields=['title', 'author'], name='unique_book_autho
            models.CheckConstraint(check=models.Q(price__gt=0), name='positive_price')
        ]

        # Indexes
        indexes = [
            models.Index(fields=['title']),

```

```

models.Index(fields=['author', 'genre'], name='author_genre_idx')
]

# Permissions
permissions = [
    ('publish_book', 'Can publish book'),
    ('feature_book', 'Can mark book as featured')
]

# Other options
verbose_name = 'Book'
verbose_name_plural = 'Books'
abstract = False # Whether model is abstract base class
proxy = False    # Whether model is proxy model
managed = True   # Whether Django manages the table lifecycle

```

◆ Model Methods

Common custom methods for models:

```

class Book(models.Model):
    # fields...

    # String representation
    def __str__(self):
        return self.title

    # URL pattern name to use with get_absolute_url
    def get_absolute_url(self):
        from django.urls import reverse
        return reverse('book-detail', args=[str(self.id)])

    # Custom save behavior
    def save(self, *args, **kwargs):
        # Do something before saving
        if not self.slug:
            self.slug = slugify(self.title)
        # Call the "real" save method
        super().save(*args, **kwargs)
        # Do something after saving

    # Custom delete behavior
    def delete(self, *args, **kwargs):
        # Do something before deletion
        # Call the "real" delete method
        super().delete(*args, **kwargs)

    # Business logic methods
    def is_on_sale(self):
        return self.discount_percent > 0

```

```
def get_discount_price(self):
    if self.is_on_sale():
        return self.price * (1 - self.discount_percent / 100)
    return self.price
```

◆ Manager Methods

Models can have custom managers with specific query methods:

```
class BookManager(models.Manager):
    # Get queryset with additional defaults
    def get_queryset(self):
        return super().get_queryset().select_related('author')

    # Method to filter published books
    def published(self):
        return self.filter(published=True)

    # Method to find bestsellers
    def bestsellers(self):
        return self.filter(is_bestseller=True)

class Book(models.Model):
    # fields...

    # Set the custom manager
    objects = BookManager()

    # Or add an additional manager
    published_books = BookManager().published()
```

Now you can use `Book.objects.published()` or `Book.bestsellers.all()` in your code.

VIEWS

◆ What they are

Views are Python functions or classes that receive an HTTP request and return an HTTP response. They are the **logic layer** — the glue between the model and the template.

◆ Function-Based View (FBV)

```
from django.shortcuts import render

def book_list(request):
    books = Book.objects.all()
    return render(request, 'books/list.html', {'books': books})
```

- Accepts request, returns `HttpResponse` or `render()`
- Can use `redirect()`, `Http404`, `JsonResponse`, etc.

♦ Class-Based View (CBV)

```
from django.views.generic import ListView
from .models import Book

class BookListView(ListView):
    model = Book
    template_name = 'books/list.html'
    context_object_name = 'books'
```

- Built-in generic views: ListView, DetailView, CreateView, UpdateView, DeleteView
- Powerful **mixin-based dispatching system**

CBVs are composable but harder to trace. Use FBVs when behavior is custom and localized. Use CBVs when logic follows CRUD.

♦ Mixin-based Dispatching System

Django's powerful mixin-based dispatching system is what makes CBVs truly flexible:

- **Composability:** Mix multiple classes to build complex views from simple pieces
- ```
class BookEditView(LoginRequiredMixin, PermissionRequiredMixin, UpdateView):
 permission_required = 'books.change_book'
 model = Book
 fields = ['title', 'author', 'genre']
```
- **Method Resolution Order:** Python's MRO determines which mixin's methods take precedence
- **Request Processing Flow:**
  - 0.1. `dispatch()` receives request first
  - 0.2. Based on HTTP method, calls `get()`, `post()`, etc.
  - 0.3. View-specific methods like `get_object()` or `form_valid()` execute
  - 0.4. Response rendering occurs
- **Common Mixins:**
  - `LoginRequiredMixin`: Restricts to authenticated users
  - `UserPassesTestMixin`: Custom access control logic
  - `FormMixin`: Adds form handling to any view
  - `JSONResponseMixin`

## URLS

### ♦ How Django dispatches requests

URLs map request paths to view functions or classes. Defined in `urls.py`.

```

from django.urls import path
from .views import BookListView

urlpatterns = [
 path('', BookListView.as_view(), name='book-list'),
 path('books/<int:pk>/', BookDetailView.as_view(), name='book-detail'),
]

```

- Use path() for simple routing, re\_path() for regex-based
- include() is used to delegate routing to app-level files

### ◆ Named URLs

Use name='route-name' so you can reverse in templates or views:

```

In views.py
from django.urls import reverse
url = reverse('book-detail', args=[book.pk]) # Returns '/books/1/'
url = reverse('author-books', kwargs={'author_id': 5}) # Returns '/authors/5/books/'

In templates
View Book

```

### ◆ URL Patterns with Path Converters

```

urls.py
urlpatterns = [
 # Integer path converter (matches digits)
 path('books/<int:pk>/', views.book_detail, name='book-detail'),

 # String path converter (matches except '/')
 path('books/<str:genre>/', views.genre_books, name='genre-books'),

 # Slug path converter (matches letters, numbers, hyphens, underscores)
 path('posts/<slug:post_slug>/', views.post_detail, name='post-detail'),

 # UUID path converter (matches formatted UUIDs)
 path('orders/<uuid:order_id>/', views.order_detail, name='order-detail'),

 # Path path converter (matches any non-empty string including /)
 path('files/<path:file_path>/', views.serve_file, name='serve-file'),
]

```

### ◆ Class-Based View URLs

```

Function-based view
path('books/', views.book_list, name='book-list'),

Class-based view - requires .as_view()
path('books/', views.BookListView.as_view(), name='book-list'),

```

```
Class-based view with arguments
path('books/create/', views.BookCreateView.as_view(
 template_name='custom_template.html',
 success_url='/thanks/'
), name='book-create'),
```

## ◆ Advanced URL Configuration

```
Including app URLconfs
path('books/', include('books.urls')),

Including with a namespace
path('books/', include(('books.urls', 'books'), namespace='books')),
Then use: {% url 'books:detail' book.pk %}

URL pattern with optional parameters
re_path(r'^blog/(?>page-(?P<page>\d+)/)?$', views.blog, name='blog'),

Custom path converters
class FourDigitYearConverter:
 regex = '[0-9]{4}'
 def to_python(self, value):
 return int(value)
 def to_url(self, value):
 return str(value)

register_converter(FourDigitYearConverter, 'year')
path('articles/<year:year>/', views.year_archive)
```

## TEMPLATES

### ◆ Template language

Django templates are designed to be **non-Turing complete** — safe for untrusted editors. Minimal logic, mostly display.

```
{% for book in books %}
 <h2>{{ book.title }}</h2>
{% endfor %}
```

### ◆ Key features

- `{{ var }}` — prints variable
- `{% if %} ... {% endif %}` — logic blocks
- `{% for %}` — iteration
- `{% include %}`, `{% extends %}` — template composition
- `{% url 'route-name' arg %}` — reverse link building

- {{ var|filter }} — filters like date, length, truncatechars, default, join, etc.

## ◆ Template inheritance

```
{% extends "base.html" %}

{% block content %}
 <!-- page-specific -->
{% endblock %}
```

## FORMS

### ◆ What they are

Python classes that render HTML forms, validate input, and bind data to models.

```
from django import forms
from django.core.validators import MinValueValidator, RegexValidator
from .models import Book, Author

ModelForm example with custom fields
class BookForm(forms.ModelForm):
 # Text fields
 title = forms.CharField(max_length=100, help_text="Enter the book title")
 subtitle = forms.CharField(required=False, widget=forms.TextInput(attrs={'placeholder': 'Enter subtitle'}))
 description = forms.CharField(widget=forms.Textarea(attrs={'rows': 5}))
 slug = forms.SlugField(help_text="URL-friendly name")

 # Number fields
 page_count = forms.IntegerField(min_value=1, validators=[MinValueValidator(1)])
 price = forms.DecimalField(max_digits=6, decimal_places=2)
 weight = forms.FloatField(required=False)

 # Date and time fields
 published_date = forms.DateField(widget=forms.DateInput(attrs={'type': 'date'}))
 release_datetime = forms.DateTimeField(widget=forms.DateTimeInput(attrs={'type': 'datetime-local'}))
 reading_time = forms.DurationField(help_text="Expected reading time (e.g., 2 hours, 30 minutes)")

 # Boolean fields
 is_bestseller = forms.BooleanField(required=False)
 in_stock = forms.NullBooleanField()

 # Choice fields
 GENRE_CHOICES = [
 ('', 'Select a genre'),
 ('fiction', 'Fiction'),
 ('nonfiction', 'Non-Fiction'),
 ('scifi', 'Science Fiction'),
 ('mystery', 'Mystery'),
]
 genre = forms.ChoiceField(choices=GENRE_CHOICES)
```



```

tags = forms.MultipleChoiceField(choices=[
 ('fantasy', 'Fantasy'),
 ('drama', 'Drama'),
 ('educational', 'Educational')
], widget=forms.CheckboxSelectMultiple)

Model relationship fields
author = forms.ModelChoiceField(queryset=Author.objects.all())
related_books = forms.ModelMultipleChoiceField(
 queryset=Book.objects.all(),
 widget=forms.SelectMultiple,
 required=False
)

File and image fields
cover_image = forms.ImageField(required=False)
sample_chapter = forms.FileField(required=False)

Special fields
email_contact = forms.EmailField(help_text="Contact email for book inquiries")
book_website = forms.URLField(required=False)
isbn = forms.CharField(
 validators=[RegexValidator(r'^\d{13}$', 'ISBN must be 13 digits')],
 help_text="13-digit ISBN number"
)
color_theme = forms.RegexField(regex=r'^#[0-9a-fA-F]{6}$', help_text="Hex color code")

Hidden field
source_id = forms.CharField(widget=forms.HiddenInput(), initial='website')

IP address field
publisher_ip = forms.GenericIPAddressField(protocol='both', unpack_ipv4=True, required=False)

JSON field
metadata = forms.JSONField(required=False)

Typed choices fields
rating = forms.TypedChoiceField(
 choices=[(str(i), str(i)) for i in range(1, 6)],
 coerce=int
)

class Meta:
 model = Book
 fields = ['title', 'slug', 'author', 'published_date'] # Only these fields from Book

def clean(self):
 """Custom validation example."""
 cleaned_data = super().clean()
 published_date = cleaned_data.get('published_date')
 release_datetime = cleaned_data.get('release_datetime')

```

```

 if published_date and release_datetime and published_date > release_datetime.date():
 raise forms.ValidationError("Release date cannot be before publication date")

 return cleaned_data

def clean_title(self):
 """Field-specific validation example."""
 title = self.cleaned_data.get('title')
 if title and title.lower() == 'untitled':
 raise forms.ValidationError("Please provide a meaningful title")
 return title

Regular Form example (not tied to a model)
class ContactForm(forms.Form):
 name = forms.CharField(max_length=100)
 email = forms.EmailField()
 subject = forms.CharField(max_length=200)
 message = forms.CharField(widget=forms.Textarea)
 cc_myself = forms.BooleanField(required=False)

```

## ◆ Form types

- **forms.Form:** Manual definition of all fields

```

class ContactForm(forms.Form):
 name = forms.CharField()
 email = forms.EmailField()

```

- **forms.ModelForm:** Auto-generates fields from a model

```

class ArticleForm(forms.ModelForm):
 class Meta:
 model = Article
 fields = ['title', 'content', 'published'] # Specific fields
 # OR fields = '__all__' # All model fields
 exclude = ['author'] # Exclude specific fields

```

## ◆ Built-in field types

### Text fields

- **CharField:** Text input with max\_length
- **TextField:** Multiline text input
- **EmailField:** Validates email format
- **URLField:** Validates URL format
- **SlugField:** Letters, numbers, underscores, hyphens

- `RegexField` : Validates against a regular expression

### Number fields

- `IntegerField` : Integer values
- `FloatField` : Floating-point numbers
- `DecimalField` : Fixed precision decimal numbers

### Date/Time fields

- `DateField` : Date values
- `TimeField` : Time values
- `DateTimeField` : Date and time
- `DurationField` : Time periods

### Choice fields

- `ChoiceField` : Select from choices
- `MultipleChoiceField` : Select multiple choices
- `TypedChoiceField` : Coerces to specific type
- `ModelChoiceField` : Select from model QuerySet
- `ModelMultipleChoiceField` : Select multiple from QuerySet

### File fields

- `FileField` : File uploads
- `ImageField` : Image file uploads (validates image format)

### Special fields

- `BooleanField` : True/False checkbox
- `NullBooleanField` : True/False/None
- `JSONField` : JSON data
- `UUIDField` : UUID validation
- `GenericIPAddressField` : IPv4/IPv6 validation

### ◆ Form widgets

Widgets control the HTML rendering of form fields.

### Text widgets

- `TextInput` : Standard text input
- `Textarea` : Multiline text area

- `PasswordInput` : Password field (masks input)
- `HiddenInput` : Hidden field

### Selector widgets

- `Select` : Dropdown selector
- `SelectMultiple` : Multiple selection dropdown
- `RadioSelect` : Radio buttons
- `CheckboxSelectMultiple` : Multiple checkboxes

### Date/Time widgets

- `DateInput` : Date input
- `DateTimeInput` : Date and time input
- `TimeInput` : Time input

### File widgets

- `FileInput` : File upload
- `ClearableFileInput` : File upload with clear option

### HTML5 Widgets

- `NumberInput` : HTML5 number input
- `RangeInput` : Slider control
- `EmailInput` : Email field with HTML5 validation
- `URLInput` : URL field with HTML5 validation
- `ColorInput` : Color picker

### Custom widget attributes

```
name = forms.CharField(
 widget=forms.TextInput(attrs={
 'class': 'form-control',
 'placeholder': 'Enter your name',
 'data-validate': 'true',
 'autofocus': True
 })
)
```

### ◆ Form validation

#### Automatic validation

- `form.is_valid()` : Returns True if all validations pass

- `form.errors`: Dictionary of field errors
- `form.cleaned_data`: Validated and converted data

### Custom validation methods

- Field-specific: `clean_<fieldname>()`
- Form-wide: `clean()`

```
def clean_username(self):
 username = self.cleaned_data.get('username')
 if User.objects.filter(username=username).exists():
 raise forms.ValidationError("Username already exists")
 return username

def clean(self):
 cleaned_data = super().clean()
 password = cleaned_data.get('password')
 confirm = cleaned_data.get('confirm_password')
 if password and confirm and password != confirm:
 raise forms.ValidationError("Passwords don't match")
 return cleaned_data
```

### Validators

```
from django.core.validators import MinLengthValidator, EmailValidator

class SignupForm(forms.Form):
 username = forms.CharField(
 validators=[MinLengthValidator(5)]
)
 email = forms.CharField(
 validators=[EmailValidator(message="Enter a valid email")]
)
```

## ADMIN

### ◆ Powerful, but dangerous if misused

```
from django.contrib import admin
from .models import Book

@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
 list_display = ('title', 'published')
 search_fields = ('title',)
```

### ◆ Admin options

- `list_display`, `search_fields`, `list_filter`, `readonly_fields`, `fieldsets`, `autocomplete_fields`

- `get_queryset()`, `get_form()`, `save_model()` — override for behavior

Admin is for **internal staff**, never users. Harden access with 2FA, audit logging, and restricted models.

## SETTINGS

### ◆ Typical core settings

```
DEBUG = False
SECRET_KEY = os.environ.get('SECRET_KEY')
ALLOWED_HOSTS = ['example.com']

DATABASES = {...}
STATIC_URL = '/static/'
MEDIA_URL = '/media/'

INSTALLED_APPS = [...]
MIDDLEWARE = [...]
```

- Use `environ.get()` with fallback defaults
- Put secrets in `.env` or vault
- Never run with `DEBUG=True` in production

## STATIC & MEDIA FILES

- **STATIC**: versioned, read-only assets (JS, CSS)
- **MEDIA**: user-uploaded files

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / 'staticfiles'

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

Use `collectstatic` to gather assets into `STATIC_ROOT` for prod.

## MIDDLEWARE

Middleware is a callable that wraps request/response. Defined in `MIDDLEWARE`.

Examples:

- `SecurityMiddleware`
- `AuthenticationMiddleware`
- `SessionMiddleware`
- Custom:

```
class MyMiddleware:
 def __init__(self, get_response):
 self.get_response = get_response
 def __call__(self, request):
 # do something
 return self.get_response(request)
```

## AUTH SYSTEM

- Users are stored in `django.contrib.auth.models.User`
- Permissions: `is_staff`, `is_superuser`, `has_perm()`
- Login/logout:

```
from django.contrib.auth import authenticate, login, logout
```

- Auth views: `LoginView`, `LogoutView`, `PasswordResetView`
- Custom user: subclass `AbstractUser` or `AbstractBaseUser`

## SIGNALS

Django signals provide a way to allow decoupled applications to get notified when certain actions occur elsewhere in the framework. They're based on the **Observer pattern** - where "senders" notify a set of "receivers" when certain events happen.

When a Django model instance is saved, it emits a `post_save` signal (as the subject). Any registered receiver functions (observers) are then automatically called, allowing operations like cache invalidation, index updates, or related object creation without directly coupling these processes to the model's save method.

Here's a deeper explanation of Django signals:

### Core Concept

Signals let you execute code when specific events occur, without tightly coupling different parts of your application. For example, you might want to create a user profile automatically whenever a new user is created.

### Built-in Signals

Django provides many built-in signals, including:

Model signals:

- `pre_save` / `post_save` : Before/after a model's `save()` method is called
- `pre_delete` / `post_delete` : Before/after a model's `delete()` method
- `m2m_changed` : When a ManyToMany relation changes

1. Request/Response signals:

- `request_started` / `request_finished`
- `got_request_exception`

2. Management signals:

- `pre_migrate` / `post_migrate`

## Signal Implementation

To use signals:

Define a receiver function - A function that will execute when the signal is sent Connect the receiver to a signal - Using the `@receiver` decorator or the `Signal.connect()` method. The receiver gets called when the signal is sent. Loose coupling for actions like `post_save`, `pre_delete`.

```
from django.db.models.signals import post_save
from django.dispatch import receiver

@receiver(post_save, sender=User)
def create_profile(sender, instance, created, **kwargs):
 if created:
 Profile.objects.create(user=instance)
```

## TESTING

- Django uses `unittest.TestCase`
- Models, views, forms all testable
- Use `Client()` for HTTP tests

```
from django.test import TestCase

class BookTest(TestCase):
 def test_view_returns_200(self):
 response = self.client.get('/books/')
 self.assertEqual(response.status_code, 200)
```

## MIGRATIONS

### ♦ What are Migrations?

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make them and when to run them.

### ♦ Key Commands

- `python manage.py makemigrations` - Create new migration files based on changes detected in your models



- `python manage.py migrate` - Apply pending migrations to synchronize the database with your models
- `python manage.py showmigrations` - List all migrations and their status (applied or not)
- `python manage.py sqlmigrate app_name 0001` - Display the SQL that would be run for a migration
- `python manage.py squashmigrations app_name 0001 0004` - Combine multiple migrations into one
- `python manage.py migrate app_name zero` - Revert all migrations for an app
- `python manage.py migrate app_name 0003` - Migrate to a specific migration

## ♦ How Migrations Work

1. **Detection:** When you run `makemigrations`, Django:
  - Compares your current models against the current migration files
  - Identifies changes (new models, changed fields, etc.)
  - Creates new migration files in your app's `migrations/` directory
2. **Migration Files:**
  - Each is a Python file with operations to transform the database
  - Contains a list of `operations` (`CreateModel`, `AddField`, `AlterField`, etc.)
  - Has dependencies on other migrations to ensure correct order
  - Is tracked in Django's `django_migrations` table once applied
3. **Application:** When you run `migrate`, Django:
  - Checks which migrations have been applied
  - Determines the order to apply pending migrations based on dependencies
  - Executes each migration within a transaction (where supported)
  - Records successful migrations in the `django_migrations` table

## ♦ Advanced Migration Techniques

### Data Migrations

```
Create an empty migration
python manage.py makemigrations app_name --empty --name=populate_data

Then edit the migration file:
from django.db import migrations

def populate_initial_data(apps, schema_editor):
```

```

Get the historical model version
Book = apps.get_model('myapp', 'Book')
Create objects
Book.objects.create(title='Django Unleashed', published=True)

class Migration(migrations.Migration):
 dependencies = [
 ('myapp', '0002_auto_20210701_1200'),
]
 operations = [
 migrations.RunPython(populate_initial_data),
]

```

## Handling Circular Dependencies

```

Using SeparateDatabaseAndState operations when models reference each other
migrations.SeparateDatabaseAndState(
 state_operations=[
 migrations.CreateModel(
 name='Book',
 fields=[
 # ...fields
 ('author', models.ForeignKey('Author', on_delete=models.CASCADE)),
],
),
],
 database_operations=[
 migrations.CreateModel(
 name='Book',
 fields=[
 # ...fields without the foreign key
],
),
],
)

```

## Migration Planning

- Always make and apply migrations in development first
- Review migration SQL before applying in production
- Consider using `--plan` flag to see what would be applied: `python manage.py migrate --plan`
- For zero-downtime deployments, ensure migrations are backward-compatible

### ♦ Common Challenges

- **Schema Changes on Large Tables:** Can lock tables during migrations
  - Solution: Use `RunSQL` with custom SQL using database-specific features

- **Modifying Data in Migrations:** Always use the historical model versions

```
WRONG - might break if model changes later
from myapp.models import Book

RIGHT - gets correct model version at migration time
Book = apps.get_model('myapp', 'Book')
```

- **Migration Conflicts:** When multiple developers create migrations simultaneously
  - Solution: Communicate and merge migrations carefully, possibly squashing
- **Rollbacks:** Django doesn't automatically generate "down" migrations
  - Solution: Plan multi-step migrations that can be rolled back safely

## MANAGEMENT COMMANDS

Common Django management commands and their purposes:

```
Development server
python manage.py runserver # Run development server on 127.0.0.1:8000
python manage.py runserver 8080 # Run on port 8080
python manage.py runserver 0.0.0.0:8000 # Listen on all interfaces

User management
python manage.py createsuperuser # Create an admin user
python manage.py changepassword username # Change a user's password

Database operations
python manage.py makemigrations # Create migrations based on model changes
python manage.py migrate # Apply migrations to the database
python manage.py showmigrations # List all migrations and their status
python manage.py sqlmigrate app_name 0001 # Show SQL for a migration
python manage.py dumpdata # Output all data in the database as JSON
python manage.py loaddata fixture.json # Load data from fixture files
python manage.py flush # Empty the database

Static files
python manage.py collectstatic # Gather static files into STATIC_ROOT
python manage.py findstatic filename # Find a static file

Interactive environments
python manage.py shell # Start Python interactive interpreter
python manage.py shell_plus # Enhanced shell (django-extensions)
python manage.py dbshell # Database interactive console

Testing
python manage.py test # Run tests
python manage.py test app_name # Run tests for specific app

Maintenance
```

```
python manage.py clearsessions # Clear expired sessions
python manage.py check # System check framework
python manage.py inspectdb # Create models from existing database
python manage.py sendtestemail # Test email setup

Project information
python manage.py diffsettings # Display differences between settings
python manage.py showmigrations # Show migration status
python manage.py check --deploy # Check deployment readiness
```

Create your own custom management commands:

```
yourapp/management/commands/my_command.py
from django.core.management.base import BaseCommand

class Command(BaseCommand):
 help = 'My custom command description'

 def add_arguments(self, parser):
 parser.add_argument('--option', help='Option description')

 def handle(self, *args, **options):
 self.stdout.write(self.style.SUCCESS('Command executed successfully'))
```

Then run with: