Python i Django. Programowanie aplikacji webowych

Autor: Jeff Forcier, Paul Bissex, Wesley Chun
Tłumaczenie: Krzysztof Rychlicki-Kicior
ISBN: 978-83-246-2225-2
Tytuł oryginału: Python Web Development with Django (Developer's Library)
Format: 170x230, stron: 392

Odkryj pełnię niezwykłych możliwości Django i twórz funkcjonalne aplikacje

- Jak w kilka minut założyć blog?
- Jak bez wysiłku tworzyć zaawansowane aplikacje?
- Jak budować modele i używać ich?

Django to stworzony w Pythonie, prosty i nowoczesny framework typu open source. Umożliwia on budowanie funkcjonalnych aplikacji webowych bez potrzeby pisania setek wierszy kodu. Został zaprojektowany w ten sposób, aby doskonale działać jako zwarta całość, ale jego składniki są ze sobą na tyle luźno powiązane, że z łatwością można dokonywać zmian, dodawać i aktualizować funkcjonalności. Ten podręcznik pomoże Ci szybko i efektywnie wykorzystać Django w praktyce.

Książka „Python i Django. Programowanie aplikacji webowych" zawiera opisy podstawowych i bardziej zaawansowanych elementów tego frameworka, a także wiele przykładowych aplikacji, wspartych szczegółowymi wyjaśnieniami dotyczącymi ich budowy. Dzięki temu podręcznikowi dowiesz się, jak wykorzystać elastyczność i szybkość tworzenia aplikacji w Pythonie w celu rozwiązania typowych problemów, które możesz napotkać podczas swojej pracy. Nauczysz się tworzyć formularze, skrypty, własny system zarządzania treścią oraz aplikacje Django zaledwie w kilka minut, a dzięki gotowym projektom poznasz także tajniki zaawansowanego projektowania.

- Python dla Django
- Instrukcje warunkowe
- Funkcje i definicje klas
- Programowanie zorientowane obiektowo
- Tworzenie bloga
- Dynamiczne strony internetowe
- Django – tworzenie i używanie modeli
- URL, obsługa http i widoki
- Szablony i przetwarzanie formularzy
- System zarządzania treścią
- Liveblog
- Zaawansowane programowanie
- Testowanie aplikacji

Poznaj i wykorzystaj nowe możliwości programowania aplikacji!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

4

Tworzenie i używanie modeli

W rozdziale 3. dowiedziałeś się, że model danych w aplikacji WWW stanowi jej fundament. Z tego względu warto rozpocząć bliższe poznawanie Django właśnie od tego komponentu. Mimo że zgodnie z tytułem niniejszy rozdział został podzielony na dwie części — tworzenie i wykorzystywanie modeli — obie są ze sobą w dużej mierze powiązane. W trakcie projektowania powinniśmy zastanowić się, jak zamierzamy wykorzystywać modele, aby wygenerować najbardziej efektywny zestaw klas i relacji. Z drugiej strony, nie będziesz w stanie wykorzystywać modeli w odpowiedni sposób, jeśli nie zrozumiesz zasad ich tworzenia.

Tworzenie modeli

Warstwa danych w Django niezwykle intensywnie wykorzystuje *maper obiektowo-relacyjny* (ORM), dlatego warto się zastanowić, skąd wziął się pomysł wykorzystania tego narzędzia oraz jakie są plusy i minusy takiego rozwiązania. Niniejszy podrozdział zaczniemy więc od omówienia ORM-u zainstalowanego w Django, następnie zanalizujemy pola dostępne w modelach danych, sposoby tworzenia relacji pomiędzy klasami i modelami, a na koniec zapoznamy się z metadanymi modeli klas, używanymi do określania specyficznych zachowań modeli i zastosowania w panelu administracyjnym Django.

Dlaczego ORM?

Django, podobnie jak większość nowoczesnych frameworków sieciowych (podobnie jak wiele innych narzędzi do tworzenia aplikacji), wykorzystuje rozbudowaną warstwę dostępu do danych, która stanowi rodzaj pośrednika pomiędzy relacyjną bazą danych a aplikacjami tworzonymi w Pythonie. ORM-y stanowią przedmiot częstych dyskusji w społecznościach

programistów. Django zostało zaprojektowane z myślą o intensywnym wykorzystaniu ORM-ów; zaprezentujemy poniżej cztery argumenty przemawiające za ich stosowaniem, z naciskiem na maper obecny w Django.

Enkapsulacja użytecznych metod

Obiekty modeli w Django są zdecydowanie najlepszą metodą definiowania kolekcji pól, które zazwyczaj odpowiadają kolumnom w bazie danych. Dzięki temu realizujemy pierwszą i zasadniczą czynność w odwzorowywaniu relacyjnej bazy danych na składniki programowania obiektowego. Zamiast tworzyć zapytanie SQL, np.: `SELECT nazwisko FROM autorzy WHERE id=5`, możesz zażądać obiektu `Author`, którego `id` ma wartość `5`, a następnie sprawdzić pole `author.name` — jest to przykład zdecydowanie bliższy pythonicznemu podejściu do obsługi danych.

Obiekty modeli zawierają o wiele większą funkcjonalność niż w powyższym opisie. ORM zawarty w Django, podobnie jak wiele innych, pozwala na określanie dodatkowych metod, dzięki którym możesz tworzyć następujące mechanizmy:

- Możesz tworzyć kombinacje pól i atrybutów tylko do odczytu, znane często pod nazwą **pól wyliczanych**. Na przykład obiekt `Order`, zawierający atrybuty `count` i `cost`, może zawierać także pole `total`, które stanowi iloczyn dwóch wcześniejszych pól. Zastosowanie typowych dla programowania obiektowego wzorców projektowych — fasad, delegacji — stanie się znacznie prostsze.
- ORM udostępnia możliwość zastąpienia domyślnych metod modyfikacji danych, takich jak zapisywanie czy usuwanie obiektów. Dzięki temu możesz wykonać dodatkowe operacje, zanim Twoje dane zostaną zapisane w bazie. Możesz też upewnić się, że aplikacja „posprząta po sobie", zanim zostanie zrealizowana operacja usunięcia rekordu, niezależnie od tego, gdzie i jak to usunięcie zachodzi.
- Powiązanie z językiem programowania — w naszym przypadku z Pythonem — jest zazwyczaj proste, dzięki czemu Twoje obiekty bazy danych są zgodne z interfejsami i API udostępnianymi przez dany język.

Przenośność

Systemy ORM, będące warstwą kodu wiążącą bazę danych z resztą Twojej aplikacji, oferują niezwykłą przenośność. Większość platform ORM obsługuje wiele baz danych; podobnie jest w przypadku Django. Gdy piszemy tę książkę, warstwa danych Django obsługuje PostegreSQL, MySQL, SQLite i Oracle. Nie jest to, rzecz jasna, lista zamknięta — będzie ona rozbudowywana, w miarę tworzenia modułów obsługi do innych baz danych.

Bezpieczeństwo

W trakcie pracy z Django (lub innym frameworkiem wykorzystującym ORM) niezmiernie rzadko będziesz wykonywał własne zapytania. Dzięki temu nie będziesz musiał obawiać się o źle zaprojektowane i podatne na **wstrzykiwanie kodu** (ang. *SQL injection*) łańcuchy zapytań. ORM-y udostępniają także mechanizmy do zabezpieczania danych wejściowych

(np. za pomocą inteligentnego cytowania znaków specjalnych), dzięki czemu nie musisz się o to martwić. Ten oczywisty pożytek wynika z zastosowanego podziału na warstwy, czego przykładem są frameworki oparte na wzorcu MVC. Gdy kod odpowiedzialny za dane zagadnienie jest dobrze zorganizowany, z pewnością bezpieczeństwo aplikacji wzrośnie, a Ty zaoszczędzisz swój cenny czas.

Ekspresywność

Choć cecha ta nie jest bezpośrednio związana z *definicją* modeli, jedną z największych zalet ORM-ów (i jednocześnie jedną z największych różnic w porównaniu z czystym kodem SQL) jest składnia zapytań wykorzystywanych do pobierania rekordów z bazy danych. Składnia wyższego poziomu jest nie tylko lepsza i łatwiejsza w trakcie pracy; przełożenie mechanizmu zapytań na realia Pythona pozwala na wykorzystywanie licznych jego zalet i technik. Jest na przykład możliwe utworzenie zapytań za pomocą iterowania po strukturach danych; w innej sytuacji zapytania te byłyby nieefektywne. Podejście to jest znacznie bardziej zwięzłe niż odpowiadający mu kod SQL. Pozwala też uniknąć bezsensownej manipulacji łańcuchami znaków, która w innym przypadku mogłaby być konieczna.

Typy pól w Django

Modele danych w Django obsługują wiele typów pól; niektóre z nich są bardzo blisko związane z ich bazodanową implementacją. Istnieją jednakże typy, które zostały zaprojektowane z myślą o formularzach stron internetowych. Większość typów można umieścić gdzieś pomiędzy tymi dwoma charakterystykami. Wyczerpującą listę typów znajdziesz w oficjalnej dokumentacji Django; w tym miejscu omówimy te najczęściej wykorzystywane. Na początku zapoznajmy się z podstawową definicją modelu.

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author)
    length = models.IntegerField()
```

Powyższy przykład powinien być dość czytelny — utworzyliśmy prosty model książki, oparty na licznych zasadach związanych z działaniem baz danych. Powyższy kod nie jest rozbudowany — katalogowanie książek to proces wymagający z reguły więcej informacji niż tytuł, autor i liczba stron — na nasze potrzeby jednak to wystarczy. Co więcej, powyższy model działa bez zarzutu. Mógłbyś dodać ten przykład do pliku *models.py* i być na najlepszej drodze do utworzenia prostego katalogu książek.

Jak widać na powyższym przykładzie, Django wykorzystuje klasy Pythona do reprezentowania obiektów, które odwzorowują tabele SQL, zaś atrybuty tych obiektów odpowiadają kolumnom. Atrybuty te same w sobie również są obiektami, a dokładnie podklasami klasy `Field`. Niektóre z nich są prostymi odpowiednikami typów kolumn SQL, a inne dostarczają pewien poziom abstrakcji. Poniżej omawiamy niektóre z podklas klasy `Field`.

- `CharField` i `TextField`. Są to prawdopodobnie najczęściej używane pola, wykorzystywane do tego samego celu — przechowują tekst. Pola typu `CharField` mają stałą, skończoną długość, zaś `TextField` mogą przechowywać teoretycznie nieskończenie dużo tekstu. Wybór jednego z tych pól zależy od Twoich potrzeb (czy zamierzasz wykorzystywać przeszukiwanie pełnotekstowe (ang. *fulltext search*), czy też zależy Ci na oszczędności pamięci).
- `EmailField`, `URLField` i `IPAddressField`. Wszystkie te pola są w gruncie rzeczy polami `CharField`, jednak oferują one dodatkową walidację. W bazie danych są one przechowywane identycznie jak pole `CharField`, jednak dodatkowy kod zapewnia walidację, dzięki czemu możesz być pewien, że użytkownik wprowadził, odpowiednio, adres e-mail, adres URL lub adres IP. Do modeli danych możesz dodawać także własny kod walidacji, tworząc przez to własne „typy pól" na tym samym poziomie, na którym Django oferuje własne (zob. rozdziały 6. i 7., aby dowiedzieć się więcej o walidacji).
- `BooleanField` i `NullBooleanField`. `BooleanField` to dobry wybór w większości sytuacji, gdy chcesz przechowywać wartość `True` lub `False`. Czasami jednak musisz uwzględnić sytuację, w której nie *znasz* wartości — można wtedy interpretować wartość jako pustą lub `null`. Z takich wniosków powstał pomysł utworzenia pola `NullBooleanField`. To rozróżnienie wynika z faktu, że modelowanie danych musi być poprzedzone procesem analizy modelu — zarówno technicznej, jak i semantycznej. Musisz zastanowić się nie tylko nad tym, jak, ale i co przechowujesz.
- `FileField`. Pole typu `FileField` jest jednym z najbardziej złożonych pól, ponieważ większość pracy nie jest związana z bazą danych, tylko z obsługą żądania. Pole `FileField` przechowuje w bazie danych tylko ścieżkę do pliku, podobnie jak jego mniej funkcjonalny kolega — `FilePathField`. Pole `FileField` wyróżnia się możliwością wysłania pliku za pomocą przeglądarki użytkownika i przechowania go na serwerze. Udostępnia ono także metody odpowiedzialne za dostęp do wysłanego pliku za pomocą adresu URL.

Powyższy wykaz zawiera tylko kilka z dostępnych w definicjach modeli Django. Kolejne wersje frameworka zawierają z reguły coraz bardziej rozbudowaną listę pól. Zapoznaj się z oficjalną dokumentacją Django, w której są umieszczane opisy wszystkich pól i ich klas. Możesz także zanalizować listingi z dalszej części książki, zwłaszcza z części III.

Klucze główne i unikalność

Jedną z najważniejszych konstrukcji związanych z relacyjnymi bazami danymi jest **klucz główny**. Klucz jest polem, którego wartości w całej tabeli muszą być unikalne (w systemach ORM mówimy o całym modelu). Klucze główne są z reguły polami automatycznie inkrementowanymi typu liczbowego, ponieważ inkrementacja stanowi najprostszą i najszybszą metodą sprawdzenia, czy każdy wiersz w tabeli ma unikalną wartość.

Klucze główne są niezwykle przydatne jako punkty odniesienia dla relacji pomiędzy modelami (zostały one opisane w poniższych podrozdziałach) — jeśli dana książka ma `ID = 5` i wiadomo, że istnieje tylko *jedna* książka o takim ID, możemy powiedzieć, że określenie *książka #5* jest absolutnie jednoznaczne.

Powyżej opisany typ klucza głównego jest powszechnie stosowany, dlatego Django tworzy automatycznie taki właśnie klucz, o ile nie określisz go jawnie. Wszystkie modele niezawierające jawnie określonego klucza głównego otrzymują atrybut `id`, będący polem typu `AutoField` (automatycznie inkrementowana liczba całkowita). Pola `AutoField` zachowują się jak zwykłe liczby całkowite; odpowiadający im typ kolumny w bazie danych zależy od wybranego systemu bazodanowego.

Jeśli chcesz zyskać większą kontrolę nad wykorzystaniem kluczy głównych, przypisz wartość `True` właściwości `primary_key` dla pola, które ma zostać Twoim kluczem głównym. Pole to stanie się kluczem i zastąpi generowane pole `id`. Taki wybór oznacza, że wartości tego pola muszą być absolutnie unikatowe. Wykorzystywanie pola zawierającego łańcuch znaków (np. imię i nazwisko lub inne identyfikatory) nie jest dobrym wyborem, o ile nie jesteś pewien na 110%, że w tym polu nie było, nie ma i nie będzie duplikatów!

Jeśli chcesz uczynić dowolne pole unikalnym, jednocześnie nie tworząc klucza głównego, możesz skorzystać z atrybutu `Unique`. Przypisanie temu atrybutowi wartości `True` dla danego pola gwarantuje jego unikalność bez konieczności nadawania klucza głównego.

Relacje pomiędzy modelami

Możliwość tworzenia relacji pomiędzy modelami jest jedną z najważniejszych zalet przemawiających za wykorzystywaniem *relacyjnych* baz danych. W tej kwestii systemy ORM mogą różnić się od siebie dość znacznie. Obecna implementacja w Django jest skupiona wokół baz danych, co oznacza, że relacje są definiowane na poziomie bazy danych, a nie tylko na poziomie aplikacji. Niestety, ze względu na fakt, że SQL udostępnia tylko jedną formę relacji — klucz obcy (ang. *foreign key*) — konieczne jest wprowadzenie dodatkowych mechanizmów na nieco wyższym poziomie w celu utworzenia bardziej zaawansowanych relacji. Najpierw zajmiemy się samym kluczem obcym, a później zastosowaniem go do tworzenia innych typów relacji.

Klucze obce

Zasada działania kluczy obcych jest prosta, dlatego nie inaczej jest w przypadku ich implementacji w Django. Są one reprezentowane za pomocą klasy `ForeignKey` (podklasy klasy `Field`). Pierwszy argument tej klasy stanowi klasę modelu, do którego chcemy się odwołać, jak w poniższym przykładzie:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author)
```

Klasy, do których odwołujemy się za pomocą kluczy obcych, muszą być zadeklarowane przed klasami, w których klucze obce są używane. Inaczej nazwa `Author` nie mogłaby być wykorzystywana w klasie `Book`, w polu typu `ForeignKey`. Możesz też korzystać z łańcucha znaków, podając nazwę klasy (jeśli jest zdefiniowana w tym samym pliku) lub korzystając z notacji kropkowej (np. `'myapp.Author'`). Poniżej znajduje się przykład zapisany z wykorzystaniem klucza obcego zawierającego łańcuch znaków:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey("Author")

class Author(models.Model):
    name = models.CharField(max_length=100)
```

Istnieje możliwość tworzenia kluczy obcych odwołujących się do modelu, w którym zostały zadeklarowane, za pomocą słowa `'self'`. Takie rozwiązanie jest często spotykane przy tworzeniu struktur hierarchicznych (np. klasa `Pojemnik` zawiera atrybut `parent` umożliwiający tworzenie zagnieżdżonych pojemników) lub podobnych konstrukcji (na przykład klasa `Pracownik` zawierająca atrybuty `przełożony` lub `kierownik`).

Mimo że klucz obcy jest definiowany tylko po jednej stronie relacji, druga strona również jest zdolna podłączyć się do relacji. Klucze obce tworzą relację **wiele do jednego** (ang. *many-to-one*), ponieważ wiele obiektów „dzieci" może odwoływać się do tego samego obiektu „rodzica". Jedno dziecko może być powiązane z jednym rodzicem, ale jeden rodzic może dysponować grupą dzieci. Korzystając z powyższego przykładu, możesz wykorzystywać instancje modeli `Book` i `Author` w następujący sposób:

```
# Zdejmij książkę z półki – zapoznaj się z podrozdziałem poświęconym zapytaniom.
book = Book.objects.get(title="Moby Dick")
# Pobierz autora książki – proste, nieprawdaż?
author = Book.author
# Pobierz wszystkie książki danego autora
books = author.book_set.all()
```

Jak widać w powyższym przykładzie, „odwrócenie" relacji od modelu `Author` do `Book` jest reprezentowane za pomocą atrybutu `Author.book_set` (jest to menedżer obiektu, opisany w dalszej części rozdziału), który jest generowany automatycznie przez ORM. Możesz zmienić tę konwencję nazewniczą, modyfikując argument `related_name` obiektu `ForeignKey`; w poprzednim przykładzie mogliśmy zdefiniować atrybut `author` jako `ForeignKey("Author", related_name="books")`. Moglibyśmy wtedy odwoływać się do atrybutu `author.books` zamiast `author.book_set`.

> **Uwaga**
>
> Wykorzystanie atrybutu `related_name` jest opcjonalne w przypadku prostych hierarchii obiektów. W praktyce jest to konieczne w przypadku bardziej złożonych relacji, na przykład gdy dysponujesz wieloma kluczami obcymi wiążącymi jeden obiekt z innymi. W takiej sytuacji ORM zasygnalizuje Ci istnienie dwóch menedżerów odwrotnych relacji przez wyświetlenie komunikatu o błędzie!

Relacje „wiele do wielu”

Klucze obce są wykorzystywane do tworzenia relacji **jeden do wielu** (lub **wiele do jednego**) — w poprzednich przykładach każdy obiekt `Book` ma przypisany jeden obiekt `Author`, a jeden obiekt `Author` może mieć wiele obiektów `Book`. Czasami konieczna jest większa swoboda. Do tej pory zakładaliśmy, że każda książka została napisana przez jednego autora. Cóż jednak począć z książkami, które były pisane przez wiele osób, jak chociażby książka, którą właśnie czytasz?

Taka sytuacja wymaga wykorzystania relacji „wiele” po obu stronach (każda książka może mieć wielu autorów, a każdy autor może mieć wiele książek). Jest to idealny przykład relacji **wiele do wielu**. SQL nie wprowadza definicji takiej relacji, dlatego musimy utworzyć ją, korzystając z kluczy obcych.

Django udostępnia specjalne pole, które ułatwia obsługę takiej sytuacji: `ManyToManyField`. Pod względem składni działa ono identycznie jak `ForeignKey`. Definiuje się je po jednej ze stron relacji (przekazując model drugiej strony relacji jako argument), a ORM automatycznie przyznaje drugiej stronie zestaw niezbędnych metod i atrybutów do obsługi relacji (z reguły polega to na utworzeniu menedżera, podobnie jak w przypadku kluczy obcych). Co ważne, pole typu `ManyToManyField` możesz utworzyć po dowolnej stronie relacji. Wybór strony nie ma znaczenia, ponieważ relacja ta jest symetryczna.

> **Uwaga**
>
> Jeśli zamierzasz wykorzystać panel administracyjny Django, pamiętaj, że formularze dla obiektów w relacji „wiele do wielu” wyświetlają pole formularza tylko po stronie, w której relacja została *zdefiniowana*.

> **Uwaga**
>
> Pola `ManyToManyField` odwołujące się do siebie (tzn. pole, które zdefiniowane w danym modelu odwołuje się do tego samego modelu) są symetryczne, ponieważ relacja jest obustronna. Nie zawsze jest to jednak najlepsze rozwiązanie, dlatego warto zmienić domyślne zachowanie, określając właściwość `symmetrical = False` w definicji pola.

Zaktualizujmy nasz przykład o obsługę książek napisanych przez wielu autorów:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
```

Pola `ManyToManyField` są wykorzystywane podobnie jak strona „wiele” w zwykłej relacji z pojedynczym kluczem obcym:

```
# Zdejmij książkę z półki
book = Book.objects.get(title="Python Web Development Django")
# Pobierz autorów książek
authors = Book.author_set.all()
```

```
# Pobierz książki, których autorem jest trzeci z autorów
books = authors[2].book_set.all()
```

Pole `ManyToManyField` w celu poprawnej obsługi relacji tworzy zupełnie nową tabelę, w której wykorzystujemy znany już mechanizm kluczy obcych. Każdy wiersz tej tabeli zawiera pojedynczą relację między dwoma obiektami, w której skład wchodzą klucze obce do obu obiektów!

Powyższa tabela jest ukryta przed użytkownikiem Django i dostępna jedynie dla ORM-ów. Nie można zatem wywoływać na niej zapytań; jedynym sposobem jej wykorzystywania jest odwoływanie się za pomocą jednej ze stron relacji. Jest jednak możliwe określenie specjalnej opcji w polu `ManyToManyField` (`through`), dzięki której można jawnie zdefiniować klasę modelu pośredniego (własną klasę obsługującą wyżej opisaną tabelę). Dzięki użyciu opcji `through` możesz zdefiniować dodatkowe pola w modelu pośrednim, jednocześnie zachowując możliwość wykorzystywania menedżerów po obu stronach relacji.

Poniższy kod działa identycznie jak przykład z polem `ManyToManyField`, jednak dołączyliśmy jawnie tabelę pośredniczącą `Authoring`, która dodaje pole `collaboration_type` do relacji. Przy tworzeniu relacji uwzględniliśmy też parametr `through` wskazujący na tabelę `Authoring`.

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author, through="Authoring")

class Authoring(models.Model):
    collaboration_type = models.CharField(max_length=100)
    book = models.ForeignKey(Book)
    author = models.ForeignKey(Author)
```

Możesz odpytywać obiekty `Author` i `Book` w ten sam sposób, jak w poprzednich przykładach. Możesz też tworzyć zapytania związane z tabelą `authoring`.

```
# Pobierz wszystkie eseje, których współautorem jest Chun
chun_essay_compilations = Book.objects.filter(
    author__name__endswith='Chun',
    authoring__collaboration_type='essays'
)
```

Mechanizm ten znacząco rozbudowuje elastyczność Django w zakresie tworzenia relacji.

Relacje „jeden do jednego”

Poza najczęściej spotykanymi relacjami „wiele do jednego” i „wiele do wielu” relacyjne bazy danych pozwalają na wykorzystanie trzeciego typu relacji: **jeden do jednego**. Podobnie jak poprzednie typy, tak i ten działa zgodnie ze swoją nazwą — po obu stronach relacji znajduje się tylko jeden obiekt.

Django wykorzystuje tę relację za pomocą pola `OneToOneField`, które również stosuje zasadę działania obiektu `ForeignKey` — pobiera jeden argument, czyli klasę, z którą chcemy utworzyć relację (lub łańcuch znaków `'self'`, jeśli odwołujemy się do tego samego modelu). Podobnie jak w przypadku obiektu `ForeignKey`, możemy podać argument `related_name`, dzięki czemu możemy korzystać z wielu relacji w przypadku dwóch takich samych klas. W przeciwieństwie do pozostałych relacji, pole `OneToOneField` nie udostępnia menedżera do zarządzania odwrotną relacją, ponieważ zawsze istnieje tylko jeden obiekt (w obydwu kierunkach).

Ten typ relacji jest najczęściej wykorzystywany do zdefiniowania obiektów złożonych lub określenia własności (przynależności jednego obiektu do innego). Jest on częściej wykorzystywany w realiach programowania obiektowego niż w świecie rzeczywistym. Zanim Django zaczęło obsługiwać dziedziczenie modeli bezpośrednio, pole `OneToOneField` było wykorzystywane do implementacji relacji pseudodziedziczenia.

Ograniczanie relacji

Na zakończenie warto zwrócić uwagę, że jest możliwe — zarówno w przypadku kluczy obcych, jak i relacji „wiele do wielu" — określenie własności `limit_choices_to`. Argument ten pobiera słownik, którego pary klucz-wartość określają słowa kluczowe zapytania i wartości (o słowach kluczowych zapytań piszemy poniżej). Dzięki temu argumentowi możesz określić zakres możliwych wartości dla relacji, którą tworzysz.

Poniższy model działa poprawnie tylko w odniesieniu do autorów, których nazwisko kończy się łańcuchem `Smith`:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class SmithBook(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

> **Uwaga**
>
> Istnieje możliwość — a czasami jest to zalecane — aby tego typu ograniczenia stosować na poziomie formularza. Przeczytaj opis pola `ModelChoiceField` i `ModelMultipleChoiceField` w rozdziale 6.

Dziedziczenie modeli

Dość niedawno w ORM-ie zainstalowanym w Django zostało wprowadzone dziedziczenie modeli. Poza kluczami obcymi i innymi rodzajami relacji istnieje możliwość definiowania modeli, które dziedziczą po sobie w tradycyjny sposób, znany z innych klas Pythona (przykłady zwykłego dziedziczenia zostały umieszczone w rozdziale 1.).

Na przykład klasa `SmithBook` jest całkowicie samodzielną klasą, która nieprzypadkowo udostępnia takie same pola jak klasa `Book`. Skoro między obydwiema klasami występuje takie podobieństwo, klasę `SmithBook` moglibyśmy uczynić klasą pochodną klasy `Book`. Korzyści są oczywiste — podklasa musi definiować tylko pola, które chce dodać bądź zmienić w stosunku do klasy nadrzędnej.

Nasza klasa `Book` nie jest zbyt skomplikowanym przykładem, jednak można wyobrazić sobie bardziej realistyczny model, zawierający dziesiątki atrybutów i skomplikowane metody. W takiej sytuacji dziedziczenie okazuje się jedną z kluczowych dróg do spełnienia opisanej w rozdziale 3. zasady DRY. Pamiętaj jednak, że w celu osiągnięcia tego samego efektu cały czas możesz korzystać z kluczy obcych i pól `OneToOneField`. To, którą technikę wybierzesz, zależy wyłącznie od Ciebie i Twojej wizji modelu danych.

Django udostępnia dwa sposoby dziedziczenia: **abstrakcyjne klasy bazowe** i **dziedziczenie po wielu tabelach**.

Abstrakcyjne klasy bazowe

Dziedziczenie z użyciem klas bazowych polega w dużym uproszczeniu na wykorzystaniu mechanizmów dziedziczenia opartych tylko na Pythonie — pozwala to na zwykłe dziedziczenie wspólnych pól i metod z klas bazowych. Na poziomie bazy danych i zapytań klasy bazowe nie istnieją, tak więc ich pola są duplikowane w tabelach baz danych ich dzieci.

Powyższe stwierdzenie brzmi jak naruszenie zasady DRY, jednak istnieją sytuacje, w których *nie chcesz* tworzyć dodatkowej tabeli dla klasy bazowej. Może się tak zdarzyć, jeśli z Twojej bazy danych korzystają inne aplikacje. Jest to także sposób na utworzenie bardziej eleganckich definicji klas bez zmiany aktualnej hierarchii obiektów.

Zmieńmy nieco hierarchię modeli `Book` i `SmithBook`, korzystając z abstrakcyjnych klas bazowych.

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
    return self.title

    class Meta:
        abstract = True

class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
})
```

Najważniejsza w powyższym kodzie jest deklaracja `abstract = True` w wewnętrznej klasie `Meta` klasy `Book`. Oznacza ona, że klasa `Book` jest abstrakcyjną klasą bazową i istnieje tylko po to, aby udostępnić swoje atrybuty klasom modeli, które po niej dziedziczą. Zauważ, że klasa `SmithBook` nadpisuje pole `authors` tylko po to, aby uzupełnić je o parametr `limit_choices_to`. Dzięki temu, że klasa `SmithBook` dziedziczy po `Book` (zamiast standardowej `models.Model`), baza danych będzie zawierać kolumny `title`, `genre` i `num_pages`, podobnie jak tabelę pomocniczą do relacji „wiele do wielu". Klasa zawiera także typową dla Pythona metodę `__unicode__`, która zwraca pole `title`, podobnie jak klasa `Book`.

Innymi słowy, zarówno w momencie tworzenia bazy danych, jak i w momencie tworzenia obiektów, odpytywania ORM-u itd. klasa `SmithBook` zachowuje się dokładnie tak, jakby była zdefiniowana następująco:

```
class SmithBook(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
})

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

Powyższa konstrukcja rozszerza mechanizm zapytań podobnie jak atrybuty instancji `SmithBook`, dlatego poniższe zapytanie będzie absolutnie poprawne:

```
smith_fiction_books = SmithBook.objects.filter(genre='Fiction')
```

Nasz przykład nie jest w pełni dostosowany do abstrakcyjnych klas bazowych, ponieważ chciałbyś zapewne tworzyć zarówno egzemplarze klasy `Book`, jak i `SmithBook`. Abstrakcyjne klasy bazowe są, rzecz jasna, abstrakcyjne — nie można tworzyć bezpośrednio egzemplarzy tych klas. Są one najbardziej użyteczne w celu spełniania reguły DRY na poziomie definicji modelu. Dziedziczenie oparte na wielu tabelach jest znacznie lepszym rozwiązaniem dla tej, konkretnej sytuacji.

Na zakończenie zauważmy jeszcze dodatkowe cechy abstrakcyjnych klas bazowych — wewnętrzna klasa `Meta` w klasach pochodnych dziedziczy po klasie `Meta` (lub jest łączona z nią) znajdującej się w klasie bazowej (oczywiście z pominięciem atrybutu `abstract`, który jest ponownie ustawiany na `False`, podobnie jak inne atrybuty, określone dla konkretnej bazy danych, np. `db_name`).

Co więcej, jeśli klasa bazowa wykorzystuje argument `related_name` do określenia relacyjnego pola, takiego jak `ForeignKey`, musisz skorzystać z formatowania łańcuchów, dzięki czemu klasy pochodne nie spowodują konfliktów. Nie korzystaj ze zwykłego łańcucha, np. `'related_pracownicy'`; umieść w nim ciąg `%(class)s`, np. `"related_%(class)s"` (przypomnij sobie fragmenty rozdziału 1., jeśli nie pamiętasz sposobu zamiany łańcuchów znaków). W ten sposób nazwy klas pochodnych są zamieniane poprawnie i nie występują konflikty.

Dziedziczenie po wielu tabelach

Dziedziczenie po wielu tabelach wydaje się tylko nieznacznie różne od abstrakcyjnych klas bazowych, przynajmniej na poziomie definicji. W tym mechanizmie również wykorzystuje się dziedziczenie klas Pythona, jednak pomijamy atrybut `abstract=True` w klasie `Meta`. W trakcie korzystania z egzemplarzy modeli czy wykonywania zapytań na nich dziedziczenie po wielu tabelach ponownie wygląda bardzo podobnie do poprzedniego rozwiązania. Klasa pochodna wydaje się dziedziczyć wszystkie atrybuty i metody klasy bazowej (z wyjątkiem klasy `Meta`, co wyjaśnimy poniżej).

Główna różnica między obydwoma rodzajami dziedziczenia polega na wewnętrznych mechanizmach odpowiadających za działanie. Klasy bazowe w tym przypadku są pełnoprawnymi modelami Django z własnymi tabelami i mogą być tworzone egzemplarze tych klas, podobnie jak ich atrybuty mogą być przekazywane do klas pochodnych. Jest to realizowane za pomocą automatycznego pola `OneToOneField` pomiędzy klasami pochodnymi i klasami bazowymi. Następnie są wykonywane czynności wiążące dwa obiekty i klasa pochodna dziedziczy atrybuty klasy bazowej.

Innymi słowy, dziedziczenie oparte na wielu tabelach stanowi „opakowanie" zwykłej relacji posiadania (sytuacja ta jest znana pod nazwą składania (złożenia) obiektów. Django ze względu na swoją pythoniczność jawnie definiuje ukrytą zazwyczaj relację. Nosi ona nazwę klasy bazowej (pisaną małymi literami) z sufiksem `_ptr`. Na przykład poniższy listing wprowadza atrybut `book_ptr` klasy `SmithBook`, który wskazuje na klasę bazową `Book`.

Poniższy kod przedstawia dziedziczenie wielotabelowe klas `Book` i `SmithBook`:

```
class Author(models.Model):
     name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.title

class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
})
```

Jedyną różnicą w tym momencie jest brak atrybutu `abstract` w klasie `Meta`. Uruchomienie polecenia `manage.py syncdb` na pustej bazie danych z plikiem *models.py* o powyższej treści spowoduje utworzenie trzech głównych tabel, podczas gdy wykorzystanie abstrakcyjnych klas bazowych spowodowałoby utworzenie jedynie dwóch tabel — `Author` i `SmithBook`.

Zauważ, że instancje klasy `SmithBook` otrzymują atrybut `book_ptr`, który wiedzie do instancji klasy `Book`, zaś obiekty klasy `Book`, które należą do określonego obiektu `SmithBook`, otrzymują atrybut `smithbook` (bez sufiksu `_ptr`).

Ta forma dziedziczenia pozwala klasom bazowym na tworzenie własnych egzemplarzy. Z tego względu dziedziczenie klasy `Meta` mogłoby spowodować problemy lub konflikty pomiędzy obydwiema stronami relacji. Z tego względu musisz zdefiniować ponownie większość opcji klasy `Meta`, które w przeciwnym razie były udostępniane pomiędzy obydwiema klasami (mimo że parametry `ordering` i `get_latest_by` są dziedziczone, o ile nie zostały zdefiniowane w klasie pochodnej). Rozwiązanie takie utrudnia spełnienie zasady DRY, ale, jak już wspomnieliśmy, nie zawsze regułę tę można bezwzględnie spełniać.

Mamy nadzieję, że wyjaśniliśmy, dlaczego ten rodzaj dziedziczenia jest lepszy dla naszego modelu z książkami. Możemy tworzyć zarówno egzemplarze klasy `Book`, jak i obiekty `SmithBook`. Jeśli korzystasz z dziedziczenia modeli, aby odwzorować relacje istniejące w realnym świecie, istnieje większa szansa, że skorzystasz z dziedziczenia po wielu tabelach zamiast abstrakcyjnych klas bazowych. Umiejętność wyboru jednego z rozwiązań przychodzi z czasem i nabytym doświadczeniem.

Wewnętrzna klasa Meta

Pola i relacje, które definiujesz w modelach, są używane przy tworzeniu bazy danych. Mają one wpływ na nazwy zmiennych, których używasz przy późniejszym odpytywaniu modelu. Często będziesz też dodawać w modelu takie metody, jak `__unicode__`, `get_absolute_url`, lub zmieniać wbudowane metody `save` albo `delete`. Na końcowy kształt modelu ma również wpływ wewnętrzna klasa, która uwzględnia w Django rozmaite metadane związane z modelem. Jest to klasa `Meta`.

Klasa `Meta`, jak sama nazwa wskazuje, zajmuje się obsługą metadanych związanych z modelem — zarówno dotyczących użycia, jak i wyświetlania, np. jak powinna być wyświetlana jego nazwa w odniesieniu do pojedynczego obiektu, a jak w przypadku wielu obiektów, jaki powinien być domyślny porządek sortowania, jaka jest nazwa tabeli itd.

Klasa `Meta` jest miejscem, w którym określamy także unikalność wielopolową, ponieważ nie miałoby sensu definiowanie takiej własności w deklaracji zwykłego pola. Dodajmy proste metadane do naszego pierwszego przykładu opartego na klasie `Book`.

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

    class Meta:
        # Porządek alfabetyczny
        ordering = ['title']
```

Klasa `Book` jest na tyle prosta, że nie musimy korzystać z większości opcji klasy `Meta`. Gdyby nie zależało nam na porządku sortowania, moglibyśmy całkowicie pominąć deklarację klasy. Klasy `Admin` i `Meta` są całkowicie opcjonalną częścią deklaracji modelu, jednak nie oznacza to, że są rzadko wykorzystywane. Zapoznajmy się z nieco bardziej skomplikowanym przykładem, ponieważ metadane klasy `Book` są dość nudne.

```
class Person(models.Model):
    first = models.CharField(max_length=100)
    last = models.CharField(max_length=100)
    middle = models.CharField(max_length=100, blank=True)

    class Meta:
        # Jest to odpowiednia metoda porządkowania, jeśli założymy, że osoby będą wyświetlane
        # w kolejności nazwisko (last), pierwsze imię (first), drugie imię (middle)
        ordering = ['last', 'first', 'middle']
        # W tym miejscu zakładamy, że nie istnieją dwie osoby o takich samych imionach i nazwiskach
        # Oczywiście w świecie rzeczywistym jest to możliwe, niemniej zakładamy, że jest to
        # świat idealny.
        unique_together = ['first', 'last', 'middle']
        # Według domyślnych ustawień Django dodaje literę 's', aby wyrazić liczbę mnogą. W tym przypadku
        # jest to złe rozwiązanie
        verbose_name_plural = "people"
```

Utworzenie modelu osoby bez wykorzystania klasy `Meta` byłoby niezwykle trudnym zadaniem. Przy sortowaniu musimy uwzględnić wszystkie trzy pola; co więcej, musimy uniknąć powstania duplikatów. Odwoływanie się do grupy osób jako *persons* z pewnością nie jest poprawnym rozwiązaniem.

Więcej szczegółów na temat różnorodnych opcji klasy `Meta`, które możesz wykorzystywać, znajdziesz w oficjalnej dokumentacji Django.

Rejestracja i opcje w panelu administratora

Jeśli korzystasz z aplikacji administratora, dostarczonej razem z Django, z pewnością aktywnie wykorzystujesz obiekty `site` (a dokładnie ich funkcję `register`), podobnie jak klasy pochodne klasy `ModelAdmin`. Klasy te pozwalają na zdefiniowanie sposobu używania modeli w ramach panelu administracyjnego.

Zarejestrowanie klasy modelu w aplikacji administratora (oczywiście po uprzednim włączeniu aplikacji, czynności opisanej w rozdziale 2.) pozwala na obsługę modelu od strony administratora w najprostszy sposób (z wykorzystaniem podstawowych formularzy). Jeśli jednak chcesz uwzględnić dodatkowe opcje, pozwalające na wybór wartości z listy, dopasowanie wyglądu formularzy itd., musisz skorzystać z klas pochodnych klasy `ModelAdmin`.

Możesz także zdefiniować wewnętrzne opcje edycji dla pól relacyjnych, takich jak klucz obcy, tworząc podklasy klasy `Inline` i odwołując się do nich w podklasach klasy `ModelAdmin`. To rozprzestrzenianie się dodatkowych klas na pierwszy rzut oka wygląda jak przerost formy nad treścią. Jednak dzięki takiemu elastycznemu rozwiązaniu każdy model może być reprezentowany na różne sposoby w różnych aplikacjach administratora. Rozszerzenie struktury i hierarchii modeli o opcję wewnętrznej edycji pozwala także na utworzenie wewnętrznego formularza w więcej niż jednej stronie modelu „rodzica”.

Dokładne opisy każdej opcji są umieszczone w oficjalnej dokumentacji — zauważ, że niektóre przykłady działania aplikacji administratora znajdziesz w części III — dlatego poniżej zamieszczamy podstawowe informacje na temat dwóch głównych typów opcji klasy `ModelAdmin`.

- Formatowanie list: `list_display`, `list_display_links`, `list_filter` itp. Opcje te pozwalają na zmianę pól widocznych w widokach list (domyślnie będących reprezentacją egzemplarzy modeli w postaci łańcucha znaków w pojedynczej kolumnie) podobnie jak włączenie pól wyszukiwania i filtrów, dzięki czemu możesz łatwo korzystać ze zbiorów informacji.
- Wyświetlanie formularzy: `fields`, `js`, `save_on_top` itp. Dostarczają one elastycznych mechanizmów, umożliwiających zmianę domyślnego wyglądu formularzy Twojego modelu, podobnie jak dodawanie kodu JavaScript i klas CSS. Mechanizmy są przydatne, jeśli chcesz dopasować wygląd i zachowanie panelu administracyjnego do reszty Twojej strony internetowej.

Jeśli w swojej pracy będziesz wykorzystywał bardzo aktywnie powyższe mechanizmy (do zmiany wyglądu i działania aplikacji administratora), warto rozważyć utworzenie własnego panelu administracyjnego, zamiast modyfikować domyślny, dostępny w Django. Zanim podejmiesz taką decyzję, zapoznaj się z rozdziałem 11., aby dowiedzieć się, jak bardzo możesz dostosować aplikację administratora do własnych potrzeb.

Wykorzystywanie modeli

Poznaliśmy już podstawy i nieco bardziej zaawansowane aspekty tworzenia modeli, więc możemy zająć się utworzeniem i wykorzystaniem bazy danych opartej na tychże modelach. Na koniec zajmiemy się zapytaniami SQL, odpowiadającymi za sprawne działanie całego mechanizmu.

Tworzenie i modyfikowanie bazy danych za pomocą manage.py

W rozdziale 2. zapoznaliśmy się po raz pierwszy ze skryptem *manage.py*, tworzonym dla każdego projektu Django i umożliwiającym pracę z bazą danych. Najczęściej wykorzystywanym poleceniem skryptu *manage.py* jest `syncdb`. Nie daj się zmylić nazwie tego polecenia — nie wykonuje ono pełnej synchronizacji modeli z bazą danych, jak mogą uważać niektórzy. Polecenie sprawdza, czy wszystkie klasy modeli są reprezentowane za pomocą tabel w bazie danych. W przypadku braku niektórych tabel skrypt je utworzy. Jeśli jednak jakaś tabela istnieje — skrypt nie wykona żadnej operacji na tej tabeli.

Jeśli więc utworzysz model i uruchomisz polecenie `syncdb`, aby zapisać zmiany w bazie danych, a następnie wprowadzisz w modelu poprawki i ponownie uruchomisz `syncdb`, skrypt nie zmodyfikuje bazy danych! W takiej sytuacji programista musi wprowadzić zmiany w strukturze bazy ręcznie (np. za pomocą konsolowego klienta), za pomocą przygotowanych skryptów lub zrzucając zawartość bazy (tabeli) do pliku, a następnie tworząc ją od nowa za pomocą polecenia `syncdb`. Na razie najważniejszy dla nas jest fakt, że polecenie `syncdb` stanowi główne narzędzie do konwersji modeli danych na bazę danych.

„Synchronizacja" bazy danych

Uzasadnieniem, jakie podają twórcy Django dla takiego, a nie innego zachowania polecenia `syncdb`, jest ich dogłębne przekonanie, że manipulowanie danymi nigdy nie powinno być procesem w pełni zautomatyzowanym. Wielu projektantów aplikacji WWW i programistów uważa, że zmiany w strukturze bazy danych powinny odbywać się tylko wtedy, gdy programista rozumie zapytanie SQL i zmiany, jakie przyniesie jego wykonanie. Autorzy zgadzają się z tym poglądem; dobre zrozumienie technologii leżących u podstaw wykorzystywanych narzędzi zawsze przynosi dobre rezultaty w trakcie pracy.

Z drugiej strony automatyczne lub półautomatyczne mechanizmy zmiany struktury bazy danych (takie jak migracje w Railsach) mogą przyspieszyć proces tworzenia oprogramowania. Istnieje przynajmniej kilka projektów opartych o Django (niezaliczających się do serca frameworka), które pozwalają na poprawienie tego braku we frameworku.

Poza poleceniem `syncdb` skrypt *manage.py* udostępnia szereg przydatnych funkcji ściśle związanych z obsługą baz danych. Część z nich jest wykonywana w ramach polecenia `syncdb` w celu prawidłowej realizacji tego polecenia. W tabeli 4.1 prezentujemy najczęściej stosowane polecenia. Wśród tych poleceń znajdziesz np. `sql` i `sqlall`, które wyświetlają zapytania `CREATE TABLE` (`sqlall` ładuje również dane początkowe), `sqlindexes` do tworzenia indeksów, `sqlreset` i `sqlclear`, które czyszczą (usuwają zawartość) lub usuwają utworzone tabele, `sqlcustom`, odpowiedzialne za wykonywanie dowolnych poleceń SQL itd.

Tabela 4.1. **Funkcje skryptu manage.py**

Funkcja skryptu manage.py	Opis
`Syncdb`	Tworzy tabele niezbędne do działania aplikacji.
`sql`	Wyświetla wywołania poleceń `CREATE TABLE`.
`sqlall`	Jak wyżej, a ponadto ładuje instrukcje dodające dane do tabel z pliku *.sql*.
`sqlindexes`	Wyświetla instrukcje, które zostały wywołane w celu utworzenia indeksów dla kolumn kluczy głównych.
`sqlclear`	Wyświetla zapytania `DROP TABLE`.
`sqlreset`	Połączenie poleceń `sqlclear` i `sql` (`DROP` i `CREATE`).
`sqlcustom`	Wyświetla polecenia SQL z pliku *.sql*.
`loaddata`	Wczytuje dane (podobnie jak `sqlcustom`, ale bez wykorzystania czystego SQL).
`dumpdata`	Zrzuca zawartość bazy danych do formatu JSON, XML itd.

W przeciwieństwie do polecenia `syncdb`, polecenia rozpoczynające się od liter `sql` nie aktualizują bazy danych. Zamiast tego wyświetlają one polecenia SQL na standardowym wyjściu, dzięki czemu programista wie, co faktycznie robi skrypt `syncdb`. Wypisane polecenia można też zapisać do pliku SQL i wykonać je samodzielnie.

Istnieje możliwość bezpośrednio przekierowania wyjścia polecenia na wejście któregoś z klientów bazy danych (utworzenia znanego z systemów uniksowych potoku), dzięki czemu są one bezpośrednio wykonywane (i faktycznie przypomina to w działaniu polecenie `syncdb`). Możesz także zapisać polecenia do pliku, dokonać zmiany, a następnie wczytać polecenia z pliku do bazy danych (zapoznaj się z dodatkiem A, aby dowiedzieć się więcej o potokach i przekierowaniach).

Więcej informacji na temat polecenia `syncdb` uzyskasz, analizując przykładowe aplikacje z części III i zapoznając się z oficjalną dokumentacją Django.

Składnia zapytań

Odpytywanie Twojej bazy danych (utworzonej na bazie modeli) wymaga wykorzystania dwóch podobnych klas: `Manager` i `QuerySet`. Obiekty klasy `Manager` są zawsze przypisane do klasy modelu, dlatego (o ile nie określisz inaczej) każda z klas modeli udostępnia atrybut `objects`, który stanowi podstawę dla wszystkich zapytań odnoszących się do danego modelu. Klasa `Manager` stanowi główną metodę uzyskiwania informacji z bazy danych. Udostępnia ona trzy metody przeznaczone do wykonywania typowych zapytań:

- `all`. Zwraca obiekt `QuerySet` zawierający wszystkie rekordy z bazy danych określone w danym modelu.
- `filter`. Zwraca obiekt `QuerySet` zawierający rekordy, które spełniają określone warunki.
- `exclude`. Przeciwieństwo metody `filter`. Zwraca rekordy, które *nie* spełniają podanych kryteriów.
- `get`. Pobiera pojedynczy rekord pasujący do zadanych warunków (zwraca błąd, jeśli nie ma takich rekordów lub istnieje więcej niż jeden).

W opisach metod pierwszej z wymienionych klas występuje druga z nich — `QuerySet`. Obiekty tej klasy można traktować jako listy egzemplarzy klas modeli, czyli mówiąc wprost — zbiory wierszy (rekordów) z bazy danych. Taka definicja jest jednak dość krzywdząca — funkcjonalność obiektów `QuerySet` znacznie wykracza poza zwykły dostęp do danych. Instancje klasy `Manager` umożliwiają generowanie zapytań, ale to w obiektach `QuerySet` zachodzi większość istotnych czynności.

Obiekty `QuerySet` wykorzystują dynamikę i elastyczność Pythona, a także *kacze typowanie* (zob. rozdział 1.), aby udostępnić trzy istotne mechanizmy: zapytania bazodanowe, pojemniki i „klocki", z których powstają bardziej zaawansowane konstrukcje — wszystkie połączone w jedną, spójną całość.

QuerySet jako zapytanie bazodanowe

Jak wynika z samej nazwy, obiekt `QuerySet` można traktować jako zapytanie do bazy danych. Może ono być przetworzone na łańcuch znaków, zawierający zapytanie SQL do wykonania w bazie danych. Ze względu na fakt, że zapytania SQL stanowią zbiór instrukcji z powiązanymi parametrami, nic dziwnego, że obiekty `QuerySet` akceptują podobne mechanizmy znane

z Pythona. Dzięki temu do obiektów `QuerySet` możemy przekazywać parametry słów kluczowych, które są automatycznie przetwarzane na odpowiedni kod SQL. Przypomnijmy sobie kod klasy modelu `Book` z tego rozdziału.

```
from myproject.myapp.models import Book
books_about_trees = Book.objects.filter(title__contains="Tree")
```

Słowa kluczowe mają bardzo ciekawą konstrukcję. Są one złożone z nazw pól (tak jak `title`), podwójnego podkreślenia i dodatkowych słów, określających inne warunki: `contains` (zawiera), `gt` (większe niż), `gte` (większe niż lub równe), `in` (testowanie przynależności do zbioru) itd. Prawie każde z tych słów jest odwzorowywane bezpośrednio na odpowiadający mu operator lub słowo kluczowe języka SQL. Zapoznaj się z oficjalną dokumentacją, aby poznać wszystkie możliwe operatory.

Metoda `Book.objects.filter` jest metodą klasy `Manager`, która zwraca obiekty klasy `QuerySet`. W powyższym przykładzie pobraliśmy przy użyciu domyślnego menedżera listę książek, zawierających w tytule słowo *Tree*, a następnie zachowaliśmy tę listę w zmiennej. Obiekty `QuerySet` przedstawia zapytanie SQL, które może wyglądać następująco:

```
SELECT * FROM myapp_book WHERE title LIKE "%Tree%";
```

Nic nie stoi na przeszkodzie, aby tworzyć złożone warunki dla zapytań. Zanalizujmy przykład dla zdefiniowanej wcześniej klasy `Person`:

```
from myproject.myapp.models import Person
john_does = Person.objects.filter(last="Doe", first="John")
```

Takie wywołanie powinno wygenerować następujące zapytanie SQL:

```
SELECT * FROM myapp_person WHERE last = "Doe" AND first = "John";
```

Wywołanie innej z metod klasy `Manager`, np. `all`, spowoduje nieznaczną zmianę powstałego zapytania:

```
everyone = Person.objects.all()
```

Powyższa instrukcja powoduje wygenerowanie następującego zapytania:

```
SELECT * FROM myapp_person;
```

Należy zauważyć, że różnego rodzaju ustawienia, zdefiniowane w klasie wewnętrznej `Meta`, mają wpływ na generowane zapytania SQL, np. atrybut `ordering` jest przekształcany na klauzulę `ORDER BY`. Jak przekonamy się niebawem, dodatkowe metody klasy `QuerySet` także mają wpływ na końcowe zapytanie SQL, które będzie wywołane w bazie danych.

Jeśli dobrze znasz i rozumiesz zasady działania języka SQL, będziesz w stanie tworzyć lepsze zapytania za pośrednictwem systemu ORM (zarówno pod względem jakości zwracanych danych, jak i czasu wykonywania). Co więcej, niedawno wdrożone i właśnie planowane mechanizmy Django w bardzo łatwy sposób pozwalają pobrać generowany przezeń kod SQL. Powstałe zapytania można rozbudowywać i ulepszać we własnym zakresie, dzięki czemu zyskujesz więcej możliwości, niż kiedykolwiek miałeś!

QuerySet jako pojemnik

`QuerySet` przypomina w działaniu listę. Implementuje on częściowo interfejs sekwencji list, dzięki czemu może być iterowany (`for record in queryset:`), indeksowany (`queryset[0]`), kawałkowany (`queryset[:5]`) i mierzony (`len(queryset)`). Skoro poznałeś już listy, krotki i iteratory w Pythonie, bez problemu powinieneś posługiwać się obiektami `QuerySet` w celu pobierania obiektów z modeli danych. Warto zauważyć, że w miarę możliwości do wykonania niektórych z powyższych operacji powinno się wykorzystywać funkcje oferowane przez bazy danych. Na przykład do pobierania fragmentów wyników i (lub) indeksowania warto wykorzystywać słowa kluczowe `LIMIT` i `OFFSET` języka SQL.

Czasami zdarzają się sytuacje, że osiągnięcie pewnych rezultatów przy użyciu obiektu `QuerySet` nie jest możliwe (lub pożądane) z wykorzystaniem ORM-u wbudowanego w Django. W tej sytuacji możesz przekształcić `QuerySet` w listę za pomocą polecenia list, dzięki czemu obiekt `QuerySet` stanie się prawdziwą listą, zawierającą cały zbiór wyników. Choć to rozwiązanie jest czasami konieczne — na przykład gdy chcesz zastosować mechanizm sortowania wbudowany w Pythona — pamiętaj, że przechowywanie tego typu danych kosztuje sporo pamięci, zwłaszcza jeśli zbiór wyników Twojego zapytania zawiera wiele rekordów.

Django stara się udostępnić możliwie jak najwięcej funkcjonalności za pomocą swojego ORM-u, dlatego jeśli chcesz przekształcić zbiór wyników na listę, upewnij się, że takie rozwiązanie jest faktycznie konieczne. Istnieje duża szansa, że Twój problem da się rozwiązać bez pobierania całego obiektu `QuerySet` do pamięci.

QuerySet jako fragment całości

Obiekt `QuerySet` można określić mianem „leniwego" — wykonuje zapytanie do bazy danych tylko wtedy, gdy musi. Dzieje się tak na przykład wtedy, gdy chcemy przekonwertować obiekt na listę lub uzyskać dostęp za pomocą metod omówionych we wcześniejszych akapitach. Takie zachowanie pozwala na wykorzystywanie jednej z największych zalet obiektów `QuerySet` — nie muszą być one wywoływane i przetwarzane osobno, dzięki czemu można z nich *tworzyć* złożone lub zagnieżdżone zapytania. Wynika to z faktu, że obiekty `QuerySet` udostępniają wiele metod klasy `Manager`, takich jak `filter` i `exclude`, a także wiele innych. Podobnie jak ich odpowiedniki z klasy `Manager`, metody te zwracają obiekty `QuerySet` — przy czym w tym przypadku są one ograniczane przez parametry wywołującego je obiektu `QuerySet`. Mechanizm ten bardzo dobrze ilustruje poniższy przykład.

```
from myproject.myapp.models import Person

doe_family = Person.objects.filter(last="Doe")
john_does = doe_family.filter(first="John")
john_quincy_does = john_does.filter(middle="Quincy")
```

Każde kolejne wywołanie powoduje ograniczenie zbioru wyników, dzięki czemu na zakończenie otrzymujemy jeden lub co najwyżej kilka rekordów, w zależności od tego, ilu Johnów Quincy Doesów znajdziemy w naszej bazie. Z małą pomocą Pythona możemy zebrać powyższe instrukcje w jedną:

```
Person.objects.filter(last="Doe").filter(first="John").filter(middle="Quincy")
```

Uważny Czytelnik zauważy, że powyższy ciąg wywołań moglibyśmy zastąpić jednym odwołaniem do funkcji `Person.objects.filter`. Warto zwrócić uwagę, że nie uzyskujemy wcześniejszego obiektu `QuerySet` o nazwie `john_does` za pomocą wywołania funkcji. W tym przypadku nie znamy dokładnej zawartości zapytania, które obsługujemy — ale nie zawsze musimy ją znać.

Załóżmy, że dodaliśmy pole `due_date` do naszego modelu `Book`, dzięki któremu będziemy mogli pobierać zaległe, nieoddane książki (wystarczy znaleźć książki, których data oddania jest określona na dzień wcześniejszy niż bieżący). Możemy pobierać obiekt `QuerySet` zawierający wszystkie książki z biblioteki, tylko beletrystykę lub wszystkie książki danej osoby, zakładając, że jest to ściśle określona kolekcja książek. Jest możliwe pobranie takich kolekcji, a następnie ograniczenie ich tylko do książek, które nas interesują, zwłaszcza tych zaległych.

```
from myproject.myapp.models import Book
from datetime import datetime
from somewhere import some_function_returning_a_queryset

book_queryset = some_function_returning_a_queryset()
today = datetime.now()
# __lt jest przetwarzane na operator "mniejszy niż" (<) w języku SQL
overdue_books = book_queryset.filter(due_date__lt=today)
```

Poza możliwością sortowania składanie obiektów `QuerySet` jest absolutnie konieczne w przypadku bardziej specyficznych konstrukcji, np. wyszukiwania książek napisanych przez autorów o nazwisku `Smith` i nienależących do beletrystyki:

```
nonfiction_smithBook.objects.filter(author__last="Smith").exclude(genre="Fiction")
```

Choć osiągnięcie powyższego efektu jest możliwe z użyciem opcji negacji, np. `__genre__neq` lub podobnej (Django ORM korzystało z tej opcji w przeszłości), wprowadzanie takich wywołań metod obiektów `QuerySet` umożliwia lepszy podział zadań. Zdecydowanie łatwiej jest też czytać taki kod, dzieląc go na ściśle określone kroki.

Sortowanie wyników zapytania

Warto wspomnieć, że obiekty `QuerySet` zawierają zestaw dodatkowych metod, niedostępnych w obiektach klasy `Manager`. Wynika to z faktu, że metody te operują wyłącznie na wynikach zapytania i jednocześnie nie generują nowych zapytań. Najczęściej wykorzystywaną metodą jest `order_by`, która nadpisuje domyślny porządek sortowania w obiekcie `QuerySet`. Załóżmy, że nasza klasa `Person` jest domyślnie sortowana po nazwisku; możemy pobrać osobny obiekt `QuerySet` posortowany według imienia, np.:

```
from myproject.myapp.models import Person
all_sorted_first = Person.objects.all().order_by('first')
```

Powstały obiekt `QuerySet` działa jak każdy inny, jednak w praktyce została dodana klauzula `ORDER BY`. Dzięki zachowaniu normalnego działania obiektu `QuerySet` możemy dalej przetwarzać go tak, jak inne tego typu obiekty. Możemy na przykład pobrać pięć pierwszych osób z listy alfabetycznie, według imion:

```
all_sorted_first_five = Person.objects.all().order_by('first')[:5]
```

Możesz sortować nawet z uwzględnieniem relacji, korzystając z zapisu podwójnego podkreślenia, o którym mówiliśmy wcześniej. Załóżmy, że nasz model `Person` ma klucz obcy (`ForeignKey`) do modelu `Address`, zawierającego m.in. pole `state`. Chcielibyśmy, aby osoby pobrane z bazy danych zostały posortowane najpierw według pola `state`, a następnie według nazwiska. Aby to osiągnąć, wystarczy wykonać poniższe polecenie:

```
sorted_by_state = Person.objects.all().order_by('address__state', 'last')
```

Sortowanie rekordów w porządku odwrotnym jest także możliwe. Wystarczy poprzedzić łańcuch identyfikujący wybrane pole znakiem minus, np. `order_by('-last')`. Możesz „odwrócić" nawet cały obiekt `QuerySet` (jeśli Twój kod przekazywał dalej obiekt `QuerySet` i nie miałeś bezpośredniej kontroli nad poprzednim wywołaniem `order_by`) przy użyciu metody `reverse` obiektu `QuerySet`.

Inne metody modyfikowania zapytań

Obiekty `QuerySet` poza sortowaniem udostępniają wiele ciekawych metod, specyficznych tylko dla nich. Należą do nich metoda `distinct`, która usuwa duplikaty ze zbioru wyników, stosując odpowiedni parametr w zapytaniach SQL (`SELECT DISTINCT`). Ciekawą metodą jest metoda `values`, która pobiera listę pól (z uwzględnieniem pól modeli powiązanych relacjami) i zwraca podklasę klasy `QuerySet`, `ValuesQuerySet`, która zawiera tylko żądane pola, przechowywane jako lista słowników zamiast tradycyjnych klas modeli. Bardzo podobnie działa metoda `values_list`, jednak zamiast słowników (zawierających pary nazwa pola — wartość pola) zwraca ona listę krotek, przechowujących tylko wartości określonych pól. Oto przykład działania obu metod:

```
>>> from myapp.models import Person
>>> Person.objects.values('first')
[{'first' : u'John'}, {'first': u'Jane'}]
>>> Person.objects.values_list('last')
[(u'Doe',), (u'Doe',)]
```

Inną przydatną, lecz często pomijaną metodą jest `select_related`, która pomaga rozwiązać jeden z podstawowych problemów systemów ORM — generowanie dużej liczby zapytań dla względnie prostych operacji. Jeśli na przykład chcesz iterować po zbiorze obiektów `Person`, aby wyświetlić informacje na temat powiązanych z nimi obiektów `Address` (przykład z poprzedniego podrozdziału), Twoja baza danych zostanie odpytana raz na początku, pod kątem listy wszystkich osób, a następnie wiele razy — po jedno zapytanie na każdy obiekt `Address`. Wyobraź sobie efekt działania tej konstrukcji dla setek albo tysięcy osób!

Aby uniknąć takiego, niewątpliwie niepożądanego, zachowania, można skorzystać z metody `select_related`. Metoda ta automatycznie łączy powiązane ze sobą obiekty, dzięki czemu uzyskujesz jedno rozbudowane zapytanie — bazy danych z reguły lepiej radzą sobie z kilkoma złożonymi zapytaniami niż wieloma prostymi. Zauważ, że metoda `select_related` nie sprawdza relacji, gdzie jest ustawiony warunek `null=True`. Pamiętaj o tym, projektując modele z myślą o wydajności.

Działanie metody `select_related` możesz kontrolować za pomocą argumentu `depth`, określającego poziom w łańcuchu relacji, do jakiego będzie „schodzić" metoda `select_related`.

W przypadku rozbudowanej struktury obiektów zapytania generowane przez tę metodę mogą być ogromne, dlatego warto pamiętać o tym argumencie. Metoda `select_related` pozwala też na wybór tylko niektórych relacji, przekazując nazwy ich pól jako argumenty pozycyjne.

Zobaczmy, jak wywołalibyśmy metodę `select_related`, aby pobrać obiekty `Person` wraz z adresami i pominąć inne klucze obce, zarówno `Person`, jak i `Address`:

```
Person.objects.all().select_related('address', depth=1)
```

Nie jest to bardzo innowacyjny przykład, jednak należy pamiętać, że metoda `select_related` (podobnie jak inne specyficzne metody) jest użyteczna wtedy, gdy musisz zrealizować pewne niestandardowe operacje, niewykonywane przez silnik zapytań domyślnie. Jeśli nie pracowałeś nad dużymi lub średnimi aplikacjami WWW, metody te mogą wydać się niezbyt użyteczne. W praktyce okazuje się, że są konieczne, gdy aplikacja jest gotowa i rozpoczyna się proces poprawiania jej wydajności!

Szczegóły na temat działania wszystkich tych funkcji, podobnie jak `order_by` i `reverse`, możesz znaleźć w oficjalnej dokumentacji Django.

Składanie słów kluczowych zapytań z wykorzystaniem Q i ~Q

Kolejne rozszerzenie obiektów `QuerySet` stanowi enkapsulująca parametry słów kluczowych klasa `Q`. Pozwala ona na stosowanie bardziej złożonych mechanizmów, z wykorzystaniem operacji logicznych `AND` i `OR` (są to operatory `&` i `|`, które mimo podobieństwa nie powinny być mylone z Pythonowymi operatorami `and` i `or` lub operatorami bitowymi `&` i `|`). Powstałe obiekty `Q` mogą być używane naprzemiennie z parametrami słów kluczowych wewnątrz wywołań metod `filter` i `exclude`, np.:

```
from myproject.myapp.models import Person
from django.db.models import Q

specific_does = Person.objects.filter(last="Doe").exclude(
    Q(first="John") | Q(middle="Quincy")
)
```

Choć powyższy przykład jest dość dziwny — nieczęsto występują sytuacje, w których szuka się osób o danym pierwszym lub drugim imieniu — powinieneś dzięki niemu zrozumieć, jak działają obiekty `Q`.

Podobnie jak w przypadku klasy `QuerySet`, obiekty `Q` mogą być łączone ze sobą. Wykorzystanie operatorów `&` i `|` w połączeniu z obiektami `Q` powoduje utworzenie nowych obiektów `Q`. Możesz na przykład utworzyć rozbudowane zapytania przy użyciu pętli:

```
first_names = ["John", "Jane", "Jeremy", "Julia"]

first_name_keywords = Q() # Puste zapytanie – możemy umieszczać w nim kolejne warunki
for name in first_names:
    first_name_keywords = first_name_keywords | Q(first=name)

specific_does = Person.objects.filter(last="Doe").filter(first_name_keywords)
```

Utworzyliśmy prostą pętlę for, w której dołączamy kolejne warunki za pomocą operatora |. Powyższy przykład nie stanowi najlepszego rozwiązania problemu — tak prosty problem można by rozwiązać za pomocą operatora zapytań __in — jednak mamy nadzieję, że dobrze ilustruje on siłę tkwiącą w składaniu obiektów Q.

> **Uwaga**
>
> Moglibyśmy zaoszczędzić kilka linijek kodu, korzystając z wbudowanych w Pythona możliwości w zakresie programowania funkcyjnego — a konkretnie rozwinięć list, wbudowanej metody reduce i modułu operator. Moduł operator dostarcza odpowiedniki operatorów w postaci funkcji — or_ dla | i and_ dla &. Trzy wiersze związane z pętlą for mogłyby być zastąpione wywołaniem reduce(or_,[Q(first=name) for name in first_names]). Skoro Django jest tworzone w Pythonie, możemy korzystać z tego rodzaju podejścia w każdej z części frameworka.

W pracy z obiektami Q możesz też wykorzystać operator unarny ~, aby dokonać negacji warunków danego obiektu. Mimo że metoda exclude obiektów QuerySet stanowi częściej stosowane rozwiązanie, ~Q jest jedynym sensownym wyborem, gdy struktura zapytania staje się bardziej skomplikowana. Rozważmy poniższe jednowierszowe wyrażenie, które pobiera wszystkie osoby o nazwisku Does, a także wszystkich o imieniu i nazwisku John Smith, poza tymi, których drugie imię zaczyna się na W.

```
Person.objects.filter(Q(last="Doe") |
    (Q(last="Smith") & Q(first="John") & ~Q(middle__startswith="W"))
)
```

Próba skorzystania z konstrukcji exclude(middle_startswith="W") nie rozwiązałoby problemu — zostałyby wykluczone wszystkie osoby o nazwisku Does z drugim imieniem zaczynającym się na literę W, a takiej sytuacji nie chcemy. Nasz problem całkowicie rozwiązuje zastosowanie konstrukcji ~Q.

Usprawnianie zapytań SQL przy użyciu metody extra

Na zakończenie omawiania mechanizmów zapytań (i w ramach wprowadzenia do podrozdziału, w którym przedstawiamy, czego owe mechanizmy zrealizować nie mogą) rozważymy metodę extra klasy QuerySet. Jest to wszechstronna metoda, która pozwala na modyfikowanie czystych zapytań SQL, generowanych przy użyciu obiektów QuerySet. Przyjmuje ona cztery parametry słów kluczowych, opisane w tabeli 4.2. Zauważ, że przykłady w tym podrozdziale wykorzystują atrybuty modeli niezdefiniowane we wcześniejszych przykładach.

Parametr select przyjmuje słownik, w którym kluczom-identyfikatorom są przypisane wartości-łańcuchy SQL. Pozwala to na dodawanie wybranych atrybutów do powstałych instancji modeli opartych na wybranych klauzulach zapytań SELECT. Mechanizm ten jest przydatny, gdy chcesz pobrać dodatkowe informacje z bazy danych i ograniczyć konieczną do tego celu logikę do kilku fragmentów Twojego kodu (w przeciwieństwie do metod modeli, których zawartość jest wykonywana wszędzie). Niektóre operacje są wykonywane szybciej w bazie danych niż w Pythonie. Pozwala to na osiągnięcie korzyści w trakcie optymalizacji.

Tabela 4.2. **Niektóre parametry przyjmowane przez metodę extra**

Parametry metody extra	Opis
`select`	Modyfikuje fragmenty zapytań `SELECT`.
`where`	Udostępnia dodatkowe klauzule `WHERE`.
`tables`	Udostępnia dodatkowe tabele.
`params`	Bezpiecznie zamienia dynamiczne parametry.

Rozważmy poniższy przykład, wykorzystujący polecenie `select` w celu dodania prostej logiki na poziomie bazy danych jako atrybut metody `extra`:

```
from myproject.myapp.models import Person

# SELECT first, last, age, (age > 18) AS is_adult FROM myapp_person;
the_folks = Person.objects.all().extra(select={'is_adult': "age > 18"})

for person in the_folks:
    if person.is_adult:
        print "%s %s jest dorosły, ponieważ ma %d lat." % (person.first,
            person.last, person.age)
```

Parametr klauzuli `where` pobiera na wejściu listę łańcuchów zawierających czyste klauzule `WHERE`, które są dołączone do zapytań SQL prawie bez zmian (lub prawie bez zmian; przeczytaj fragment poświęcony parametrowi `params`). Parametr `where` jest wykorzystywany w sytuacji, gdy nie możesz poprawnie sformułować zapytania przy użyciu związanych z atrybutami parametrów słów kluczowych, takich jak `__gt` lub `__icontains`. W następnym przykładzie korzystamy z konstrukcji języka SQL, aby zarówno znaleźć, jak i zwrócić połączony łańcuch (łącząc w stylu PostgreSQL, za pomocą znaków `||`)[1]:

```
# SELECT first, last, (first||last) AS username FROM myapp_person WHERE
# first||last ILIKE 'jeffreyf%';
matches = Person.objects.all().extra(select={'username': "first||last"},
    where=["first||last ILIKE 'jeffreyf%'"])
```

Prawdopodobnie najprostszym parametrem metody `extra` jest `tables`, który pozwala na określenie listy dodatkowych nazw `table`. Nazwy te są dołączane za klauzulą `FROM` zapytania, często w połączeniu z instrukcją `JOIN`. Pamiętaj, że Django domyślnie nazywa tabele zgodnie ze schematem `nazwa aplikacji_nazwamodelu`.

Poniższy przykład prezentuje działanie parametru `tables`, który różni się nieco od reszty (i zwraca obiekt klasy `Book` z dodatkowym atrybutem `author_last`) dla zachowania zwięzłości:

```
from myproject.myapp.models import Book

# SELECT * FROM myapp_book, myapp_person WHERE last = author_last
joined = Book.objects.all().extra(tables=["myapp_person"], where=["last =
author_last"])
```

[1] Warto zauważyć, że wykorzystanie niektórych mechanizmów SQL czyni kod nieprzenośnym. Na przykład wykorzystany w poniższym kodzie operator `ILIKE` jest niestandardowym rozszerzeniem bazy PostgreSQL oznaczającym „`LIKE` bez uwzględniania wielkości liter" — *przyp. tłum.*

Na zakończenie zapoznamy się z parametrem `params`. Jedną z dobrych praktyk programistycznych związanych z wykonywaniem zapytań w językach programowania wysokiego poziomu jest właściwe wstawianie dynamicznych parametrów. Najczęstszym błędem początkujących programistów jest zwykłe wstawianie parametrów do łańcucha zapytania, co prowadzi do powstania licznych luk bezpieczeństwa i błędów.

Wykorzystując metodę `extra`, wstaw parametr słowa kluczowego `params`. Stanowi on zwykłą listę wartości zastępujących znaczniki `%s` w łańcuchach klauzuli `where`, np.:

```
from myproject.myapp.models import Person
from somewhere import unknown_input

# Błąd: poniższy kod zadziała, ale jest podatny na wstrzykiwanie kodu SQL i pochodne problemy
# Zauważ, że '%s' jest zastępowane za pomocą zwykłego Pythonowego mechanizmu podstawiania wartości
matches = Person.objects.all().extra(where=["first = '%s'" % unknown_input()])

# Poprawne: dołączy niezbędne cudzysłowy i inne specjalne znaki, w zależności od wykorzystywanej
# bazy danych. Zauważ, że '%s' nie jest zastępowane za pomocą tradycyjnego mechanizmu, tylko przy użyciu
# argumentu 'params'.
matches = Person.objects.all().extra(where=["first = '%s'"],
    params=[unknown_input()])
```

Wykorzystywanie funkcji SQL niedostępnych w Django

Na zakończenie omawiania funkcjonalności Django w zakresie modeli i zapytań należy wspomnieć, że ORM wbudowany w Django nie jest w stanie obsłużyć wszystkich mechanizmów języka SQL. Bardzo niewiele ORM-ów uznaje się za w pełni kompatybilne z bazami danych. Django niestety niektórych funkcji brakuje, choć jego twórcy cały czas starają się rozwijać jego możliwości. Czasami, co dotyczy zwłaszcza doświadczonych programistów, trzeba skorzystać bezpośrednio z bazy danych, bez pomocy ORM-u. Poniższe podrozdziały pomagają zrozumieć działanie kilku takich typowych czynności.

Definicja struktury i wczytywanie plików inicjalizacji SQL

Większość systemów zarządzania bazami danych (RDBMS), poza tabelami i kolumnami, udostępnia dodatkowe mechanizmy, takie jak widoki (tabele zagregowane), wyzwalacze lub kaskady (wykonywanie operacji na tabelach powiązanych w momencie aktualizacji lub usuwania rekordów), a nawet własne typy danych. ORM wbudowany w Django, podobnie jak wiele innych, pomija te mechanizmy, chociaż nie oznacza to, że nie możesz ich wykorzystywać.

Niedawno dodaną funkcją frameworku definicji modeli w Django jest możliwość definiowania dowolnych plików **inicjalizacji SQL**. Są to pliki o rozszerzeniu *.sql*, które muszą się znaleźć w podkatalogu *sql*, np. *mojprojekt/moja aplikacja/sql/triggers.sql*. Każdy plik spełniający te warunki jest automatycznie wykonywany, zawsze gdy uruchamiane są polecenia skryptu *manage.py* związane z językiem SQL, np. `reset` lub `syncdb`. Są one także

dołączane do rezultatów działania poleceń `sqlall` i `sqlreset`. Możesz też wydrukować tylko takie pliki, korzystając z polecenia `sqlcustom`, które (podobnie jak inne polecenia `sql*`) wypisuje znalezione pliki SQL.

Dzięki plikom inicjalizacji SQL możesz przechowywać polecenia tworzące strukturę bazy danych. Masz pewność, że zawsze będą one uwzględniane przez narzędzia Django w trakcie budowania lub przebudowywania bazy danych. Większość poniżej wymienionych elementów może być zrealizowana przy użyciu tej funkcji.

- Widoki. Widoki SQL w praktyce są tabelami tylko do odczytu. Możesz obsługiwać je za pomocą definicji modeli, odwzorowując ich strukturę, a następnie korzystać ze zwykłego API do wykonywania zapytań. Zauważ, że nie możesz korzystać z żadnych poleceń skryptu *manage.py*, które spowodowałyby zapis do bazy danych. Niezależnie od wykorzystywanej biblioteki dostępu do bazy, próba *zapisu* do widoku wygenerowałaby błędy.
- Kaskadowość i wyzwalacze. Obydwa mechanizmy działają poprawnie z metodami ORM-ów wykonujących wstawianie i aktualizowanie. Mogą być one definiowane w plikach inicjalizacji SQL w zależności od używanej bazy danych (powiązania kaskadowe mogą być ręcznie dodane do poleceń wyświetlanych za pomocą `sqlall`, jeśli nie mogą być one utworzone normalnie).
- Własne funkcje i typy danych. Możesz definiować je w plikach inicjalizacji SQL, ale w ramach systemu ORM możesz korzystać z nich tylko za pomocą metody `QuerySet.extra`.

Fikstury — wczytywanie i zrzucanie danych

Choć nie jest to kwestia ściśle związana z językiem SQL, chcielibyśmy w tym miejscu wspomnieć o funkcji Django, pozwalającej na pracę z bazą danych poza działaniem samego ORM-u, czyli o fiksturach. Fikstury, omówione pokrótce w rozdziale 2., są to zbiory danych pochodzące z bazy danych, ale przechowywane w zwykłych plikach, z reguły w formatach innych niż SQL: XML, YAML lub JSON.

Fikstury są najczęściej wykorzystywane do wczytywania danych do bazy danych tuż po jej utworzeniu, np. w celu „wypełnienia" tabel wykorzystywanych do określania kategorii dla danych pobranych od użytkownika lub do ładowania danych testowych, przydatnych w trakcie procesu tworzenia aplikacji. Django wspiera ten mechanizm w podobny sposób, jak w przypadku plików inicjalizacji SQL; każda aplikacja Django przeszukuje podkatalog *fixtures* pod kątem plików o nazwach *initial_data.json* (lub *.xml*, *.yaml* albo innych formatów serializacji). Pliki te są następnie wczytywane przy użyciu modułu serializacji Django (zob. rozdział 9. w celu uzyskania więcej informacji na ten temat), a ich zawartość służy do tworzenia obiektów bazodanowych. Dzieje się tak zawsze wtedy, gdy są uruchamiane polecenia tworzenia (resetowania), takie jak `manage.py syncdb` lub `reset`.

Przeanalizujmy prosty przykład pliku fikstur w formacie JSON, przedstawiającym klasę modelu `Person`:

```
[
    {
        "pk": "1",
        "model": "myapp.person",
        "fields": {
            "first": "John",
            "middle": "Q",
            "last": "Doe"
        }
    },
    {
        "pk": "2",
        "model": "myapp.person",
        "fields": {
            "first": "Jane",
            "middle": "N",
            "last": "Doe"
        }
    }
]
```

Wczytanie powyższego pliku spowoduje wygenerowanie następujących komunikatów:

```
user@example:/opt/code/mojprojekt $ ./manage.py syncdb
Installing json fixture 'initial_data' from '/mojprojekt/mojaaplikacja /fixtures'.
Installed 2 object(s) from 1 fixture(s)
```

Poza dodawaniem do bazy danych początkowych fikstury są użyteczne do zrzucania danych z Twojej bazy. Zastosowanie bardziej neutralnego (niezwiązanego konkretnie z żadną bazą danych) narzędzia jest lepsze niż ściśle powiązane z wybraną bazą danych narzędzie do wykonywania zrzutów (np. `mysqldump`) — dzięki temu możesz zrzucić dane z bazy danych PostgreSQL, a następnie wczytać je do bazy danych MySQL. Nie byłoby to takie proste bez pośredniego procesu tłumaczenia przy użyciu fikstur. Akcje te wykonuje się przy użyciu poleceń `manage.py dumpdata` i (lub) `loaddata`.

W trakcie wykorzystywania poleceń `dumpdata` i `loaddata` nazwa i położenie wykorzystywanych fikstur mogą być bardziej dopasowane niż w przypadku danych inicjalizacyjnych. Pliki te mogą mieć dowolną nazwę (o ile rozszerzenie należy do obsługiwanych formatów) i mogą być zlokalizowane w podkatalogu *fixtures*, dowolnym katalogu określonym w opcji `FIXTURES_DIRS` lub w dowolnym katalogu określonym jawnie w wywołaniu metody `loadddata` lub `dumpdata`. Możesz na przykład zrzucić dwa obiekty `Person` (poprzednio zaimportowane) w następujący sposób (wykorzystujemy opcję `indent`, aby składnia była bardziej dostosowana do czytania).

```
user@example:/opt/code/mojprojekt $ ./manage.py dumpdata –indent=4 mojaaplikacja >
↪/tmp/mojaaplikacja.json
user@example:/opt/code/mojprojekt $ cat /tmp/mojaaplikacja.json
[
    {
        "pk": 1,
        "model": "testapp.person",
        "fields": {
            "middle": "Q",
```

```
            "last": "Doe",
            "first": "John"
        }
    },
    {
        "pk": 2,
        "model": "testapp.person",
        "fields": {
            "middle": "N",
            "last": "Doe",
            "first": "Jane"
        }
    }
]
```

Jak widać, fikstury są bardzo dobrą metodą przechowywania danych w formacie, z którym pracuje się łatwiej niż z czystym SQL.

Własne zapytania SQL

Warto pamiętać, że jeśli ORM (włączając w niego możliwości metody `extra`) nie spełnia Twoich oczekiwań w zakresie zapytań, zawsze możesz wywołać całkowicie dowolne zapytanie SQL, wykorzystując do tego celu niskopoziomowy adapter bazy danych. ORM wykorzystuje te same moduły do własnego dostępu do bazy danych. Moduły te są ściśle zależne od bazy danych, w zależności od Twoich bazodanowych ustawień, i są zgodne ze specyfikacją DB-API. Wystarczy zaimportować obiekt `connection` zdefiniowany w module `django.db`, pobrać kursor bazy danych i wywołać zapytanie.

```
from django.db import connection

cursor = connection.cursor()
cursor.execute("SELECT first, last FROM myapp_person WHERE last='Doe'")
doe_rows = cursor.fetchall()
for row in doe_rows:
    print "%s %s" % (row[0], row[1])
```

> **Uwaga**
>
> Zapoznaj się z dokumentacją DB-API Pythona, przeczytaj rozdział „Database" z książki *Core Python Programming* i dokumentację adapterów bazy danych (zob. *withdjango.com*), aby dowiedzieć się więcej szczegółów na temat składni i metod oferowanych przez te moduły.

Podsumowanie

Omówiliśmy wiele kluczowych elementów w tym rozdziale i mamy nadzieję, że zauważyłeś, ile pracy i analizy należy włożyć w zaprojektowanie dobrych modeli danych. Dowiedziałeś się, dlaczego warto korzystać z ORM-ów, jak definiować proste modele Django i bardziej złożone, zawierające relacje i specjalne klasy wewnętrzne, wykorzystywane do określania

metadanych. Mamy nadzieję, że przekonaliśmy Cię do ogromnych możliwości klasy `QuerySet`, przydatnej do pobierania danych z modeli, i że zapoznaliśmy Cię z metodami dostępu do danych spoza ORM-u.

W następnych dwóch rozdziałach wyjaśnimy, jak można wykorzystywać modele danych w aplikacjach WWW, ustawiając zapytania w kontrolerach logiki (widoki) i wyświetlając pochodzące z nich dane w szablonach.