Chapter 10: Advanced Models

In Chapter 5, we presented an introduction to Django's database layer – how to define models and how to use the database API to create, retrieve, update and delete records. In this chapter, we'll introduce you to some more advanced features of this part of Django.

Related Objects

Recall our book models from Chapter 5:

```
from django.db import models


class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    def __unicode__(self):
```

```
        return u'%s %s' % (self.first_name,
self.last_name)


class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()


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

As we explained in Chapter 5, accessing the value for a particular field on a database object is as straightforward as using an attribute. For example, to determine the title of the book with ID 50, we'd do the following:

```
>>> from mysite.books.models import Book
>>> b = Book.objects.get(id=50)
>>> b.title
u'The Django Book'
```

But one thing we didn't mention previously is that related objects – fields expressed as either a ForeignKeyor ManyToManyField – act slightly differently.

Accessing Foreign Key Values

When you access a field that's a ForeignKey, you'll get the related model object. For example:

```
>>> b = Book.objects.get(id=50)
>>> b.publisher
<Publisher: Apress Publishing>
>>> b.publisher.website
u'http://www.apress.com/'
```

With ForeignKey fields, it works the other way, too, but it's slightly different due to the non-symmetrical nature of the relationship. To get a list of books for a given publisher, use publisher.book_set.all(), like this:

```
>>> p = Publisher.objects.get(name='Apress Publishing')

>>> p.book_set.all()

[<Book: The Django Book>, <Book: Dive Into Python>,
...]
```

Behind the scenes, book_set is just a QuerySet (as covered in Chapter 5), and it can be filtered and sliced like any other QuerySet. For example:

```
>>> p = Publisher.objects.get(name='Apress Publishing')

>>> p.book_set.filter(title__icontains='django')

[<Book: The Django Book>, <Book: Pro Django>]
```

The attribute name book_set is generated by appending the lower case model name to _set.

Accessing Many-to-Many Values

Many-to-many values work like foreign-key values, except we deal with QuerySet values instead of model instances. For example, here's how to view the authors for a book:

```
>>> b = Book.objects.get(id=50)

>>> b.authors.all()

[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-
Moss>]

>>> b.authors.filter(first_name='Adrian')

[<Author: Adrian Holovaty>]

>>> b.authors.filter(first_name='Adam')

[]
```

It works in reverse, too. To view all of the books for an author, use author.book_set, like this:

```
>>> a = Author.objects.get(first_name='Adrian',
last_name='Holovaty')

>>> a.book_set.all()
```

```
[<Book: The Django Book>, <Book: Adrian's Other Book>]
```

Here, as with ForeignKey fields, the attribute name book_set is generated by appending the lower case model name to _set.

Making Changes to a Database Schema

When we introduced the syncdb command in Chapter 5, we noted that syncdb merely creates tables that don't yet exist in your database – it does *not* sync changes in models or perform deletions of models. If you add or change a model's field, or if you delete a model, you'll need to make the change in your database manually. This section explains how to do that.

When dealing with schema changes, it's important to keep a few things in mind about how Django's database layer works:

- Django will complain loudly if a model contains a field that has not yet been created in the database table. This will cause an error the first time you use the Django database API to query the given table (i.e., it will happen at code execution time, not at compilation time).
- Django does *not* care if a database table contains columns that are not defined in the model.
- Django does *not* care if a database contains a table that is not represented by a model.

Making schema changes is a matter of changing the various pieces – the Python code and the database itself – in the right order.

Adding Fields

When adding a field to a table/model in a production setting, the trick is to take advantage of the fact that Django doesn't care if a table contains columns that aren't defined in the model. The strategy is to add the column in the database, and then update the Django model to include the new field.

However, there's a bit of a chicken-and-egg problem here, because in order to know how the new database column should be expressed in SQL, you need to look at the output of Django's manage.py sqlallcommand, which requires that the field exist in the model. (Note that you're not *required* to create your column with exactly the same SQL that Django would, but it's a good idea to do so, just to be sure everything's in sync.)

The solution to the chicken-and-egg problem is to use a development environment instead of making the changes on a production server. (You *are* using a testing/development environment, right?) Here are the detailed steps to take.

First, take these steps in the development environment (i.e., not on the production server):

1. Add the field to your model.
2. Run manage.py sqlall [yourapp] to see the new CREATE TABLE statement for the model. Note the column definition for the new field.
3. Start your database's interactive shell (e.g., psql or mysql, or you can use manage.py dbshell). Execute an ALTER TABLE statement that adds your new column.

4. Launch the Python interactive shell with `manage.py shell` and verify that the new field was added properly by importing the model and selecting from the table (e.g., `MyModel.objects.all()[:5]`). If you updated the database correctly, the statement should work without errors.

   Then on the production server perform these steps:

1. Start your database's interactive shell.
2. Execute the `ALTER TABLE` statement you used in step 3 of the development environment steps.
3. Add the field to your model. If you're using source-code revision control and you checked in your change in development environment step 1, now is the time to update the code (e.g., `svn update`, with Subversion) on the production server.
4. Restart the Web server for the code changes to take effect.

   For example, let's walk through what we'd do if we added a `num_pages` field to the `Book` model from Chapter 5. First, we'd alter the model in our development environment to look like this:

```
class Book(models.Model):

    title = models.CharField(max_length=100)

    authors = models.ManyToManyField(Author)

    publisher = models.ForeignKey(Publisher)

    publication_date = models.DateField()

    num_pages = models.IntegerField(blank=True,
null=True)


    def __unicode__(self):

        return self.title
```

(Note: Read the section "Making Fields Optional" in Chapter 6, plus the sidebar "Adding NOT NULL Columns" below for important details on why we included `blank=True` and `null=True`.)

Then we'd run the command `manage.py sqlall books` to see the `CREATE TABLE` statement. Depending on your database backend, it would look something like this:

```
CREATE TABLE "books_book" (

    "id" serial NOT NULL PRIMARY KEY,

    "title" varchar(100) NOT NULL,

    "publisher_id" integer NOT NULL REFERENCES
"books_publisher" ("id"),
```

```
    "publication_date" date NOT NULL,
    "num_pages" integer NULL
);
```

The new column is represented like this:

```
"num_pages" integer NULL
```

Next, we'd start the database's interactive shell for our development database by typing psql (for PostgreSQL), and we'd execute the following statements:

```
ALTER TABLE books_book ADD COLUMN num_pages integer;
```

Adding NOT NULL Columns

There's a subtlety here that deserves mention. When we added the num_pages field to our model, we included the blank=True and null=True options. We did this because a database column will contain NULL values when you first create it.

However, it's also possible to add columns that cannot contain NULL values. To do this, you have to create the column as NULL, then populate the column's values using some default(s), and then alter the column to set the NOT NULL modifier. For example:

```
BEGIN;
ALTER TABLE books_book ADD COLUMN num_pages integer;
UPDATE books_book SET num_pages=0;
ALTER TABLE books_book ALTER COLUMN num_pages SET NOT NULL;
COMMIT;
```

If you go down this path, remember that you should leave off blank=True and null=True in your model (obviously).

After the ALTER TABLE statement, we'd verify that the change worked properly by starting the Python shell and running this code:

```
>>> from mysite.books.models import Book
>>> Book.objects.all()[:5]
```

If that code didn't cause errors, we'd switch to our production server and execute the `ALTER TABLE` statement on the production database. Then, we'd update the model in the production environment and restart the Web server.

Removing Fields

Removing a field from a model is a lot easier than adding one. To remove a field, just follow these steps:

1. Remove the field from your model and restart the Web server.
2. Remove the column from your database, using a command like this:

```
ALTER TABLE books_book DROP COLUMN num_pages;
```

Make sure to do it in this order. If you remove the column from your database first, Django will immediately begin raising errors.

Removing Many-to-Many Fields

Because many-to-many fields are different than normal fields, the removal process is different:

1. Remove the `ManyToManyField` from your model and restart the Web server.
2. Remove the many-to-many table from your database, using a command like this:

```
DROP TABLE books_book_authors;
```

As in the previous section, make sure to do it in this order.

Removing Models

Removing a model entirely is as easy as removing a field. To remove a model, just follow these steps:

1. Remove the model from your `models.py` file and restart the Web server.
2. Remove the table from your database, using a command like this:

```
DROP TABLE books_book;
```

Note that you might need to remove any dependent tables from your database first – e.g., any tables that have foreign keys to `books_book`.

As in the previous sections, make sure to do it in this order.

Managers

In the statement `Book.objects.all()`, `objects` is a special attribute through which you query your database. In Chapter 5, we briefly identified this as the model's *manager*. Now it's time to dive a bit deeper into what managers are and how you can use them.

In short, a model's manager is an object through which Django models perform database queries. Each Django model has at least one manager, and you can create custom managers in order to customize database access.

There are two reasons you might want to create a custom manager: to add extra manager methods, and/or to modify the initial QuerySet the manager returns.

Adding Extra Manager Methods

Adding extra manager methods is the preferred way to add "table-level" functionality to your models. (For "row-level" functionality – i.e., functions that act on a single instance of a model object – use model methods, which are explained later in this chapter.)

For example, let's give our Book model a manager method title_count() that takes a keyword and returns the number of books that have a title containing that keyword. (This example is slightly contrived, but it demonstrates how managers work.)

```
# models.py

from django.db import models

# ... Author and Publisher models here ...

class BookManager(models.Manager):
    def title_count(self, keyword):
        return
self.filter(title__icontains=keyword).count()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True,
null=True)
```

```
    objects = BookManager()


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

With this manager in place, we can now do this:

```
>>> Book.objects.title_count('django')
4
>>> Book.objects.title_count('python')
18
```

Here are some notes about the code:

- We've created a BookManager class that extends django.db.models.Manager. This has a single method,title_count(), which does the calculation. Note that the method uses self.filter(), where selfrefers to the manager itself.
- We've assigned BookManager() to the objects attribute on the model. This has the effect of replacing the "default" manager for the model, which is called objects and is automatically created if you don't specify a custom manager. We call it objects rather than something else, so as to be consistent with automatically created managers.

  Why would we want to add a method such as title_count()? To encapsulate commonly executed queries so that we don't have to duplicate code.

Modifying Initial Manager QuerySets

A manager's base QuerySet returns all objects in the system. For example, Book.objects.all() returns all books in the book database.

You can override a manager's base QuerySet by overriding the Manager.get_query_set() method.get_query_set() should return a QuerySet with the properties you require.

For example, the following model has *two* managers – one that returns all objects, and one that returns only the books by Roald Dahl.

```
from django.db import models


# First, define the Manager subclass.
class DahlBookManager(models.Manager):
```

```
    def get_query_set(self):
        return super(DahlBookManager,
self).get_query_set().filter(author='Roald Dahl')


# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
    # ...


    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-
specific manager.
```

With this sample model, Book.objects.all() will return all books in the database, butBook.dahl_objects.all() will only return the ones written by Roald Dahl. Note that we explicitly set objectsto a vanilla Manager instance, because if we hadn't, the only available manager would be dahl_objects.

Of course, because get_query_set() returns a QuerySet object, you can use filter(), exclude() and all the other QuerySet methods on it. So these statements are all legal:

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

This example also pointed out another interesting technique: using multiple managers on the same model. You can attach as many Manager() instances to a model as you'd like. This is an easy way to define common "filters" for your models.

For example:

```
class MaleManager(models.Manager):
    def get_query_set(self):
```

```
        return super(MaleManager,
self).get_query_set().filter(sex='M')


class FemaleManager(models.Manager):

    def get_query_set(self):

        return super(FemaleManager,
self).get_query_set().filter(sex='F')


class Person(models.Model):

    first_name = models.CharField(max_length=50)

    last_name = models.CharField(max_length=50)

    sex = models.CharField(max_length=1, choices=(('M',
'Male'), ('F', 'Female')))

    people = models.Manager()

    men = MaleManager()

    women = FemaleManager()
```

This example allows you to request Person.men.all(), Person.women.all(), and Person.people.all(), yielding predictable results.

If you use custom Manager objects, take note that the first Manager Django encounters (in the order in which they're defined in the model) has a special status. Django interprets this first Manager defined in a class as the "default" Manager, and several parts of Django (though not the admin application) will use that Managerexclusively for that model. As a result, it's often a good idea to be careful in your choice of default manager, in order to avoid a situation where overriding of get_query_set() results in an inability to retrieve objects you'd like to work with.

Model methods

Define custom methods on a model to add custom "row-level" functionality to your objects. Whereas managers are intended to do "table-wide" things, model methods should act on a particular model instance.

This is a valuable technique for keeping business logic in one place – the model.

An example is the easiest way to explain this. Here's a model with a few custom methods:

```
from django.contrib.localflavor.us.models import
USStateField

from django.db import models


class Person(models.Model):

    first_name = models.CharField(max_length=50)

    last_name = models.CharField(max_length=50)

    birth_date = models.DateField()

    address = models.CharField(max_length=100)

    city = models.CharField(max_length=50)

    state = USStateField() # Yes, this is U.S.-
centric...



    def baby_boomer_status(self):

        "Returns the person's baby-boomer status."

        import datetime

        if datetime.date(1945, 8, 1) <= self.birth_date
<= datetime.date(1964, 12, 31):

            return "Baby boomer"

        if self.birth_date < datetime.date(1945, 8, 1):

            return "Pre-boomer"

        return "Post-boomer"



    def is_midwestern(self):

        "Returns True if this person is from the
Midwest."

        return self.state in ('IL', 'WI', 'MI', 'IN',
'OH', 'IA', 'MO')
```

```
    def _get_full_name(self):
        "Returns the person's full name."
        return u'%s %s' % (self.first_name,
self.last_name)
    full_name = property(_get_full_name)
```

The last method in this example is a "property." Read more about properties athttp://www.python.org/download/releases/2.2/descrintro/#property

And here's example usage:

```
>>> p = Person.objects.get(first_name='Barack',
last_name='Obama')
>>> p.birth_date
datetime.date(1961, 8, 4)
>>> p.baby_boomer_status()
'Baby boomer'
>>> p.is_midwestern()
True
>>> p.full_name  # Note this isn't a method -- it's
treated as an attribute
u'Barack Obama'
```

Executing Raw SQL Queries

Sometimes you'll find that the Django database API can only take you so far, and you'll want to write custom SQL queries against your database. You can do this very easily by accessing the objectdjango.db.connection, which represents the current database connection. To use it, callconnection.cursor() to get a cursor object. Then, call cursor.execute(sql, [params]) to execute the SQL and cursor.fetchone() or cursor.fetchall() to return the resulting rows. For example:

```
>>> from django.db import connection
>>> cursor = connection.cursor()
```

```
>>> cursor.execute("""
...     SELECT DISTINCT first_name
...     FROM people_person
...     WHERE last_name = %s""", ['Lennon'])
>>> row = cursor.fetchone()
>>> print row
['John']
```

connection and cursor mostly implement the standard Python "DB-API," which you can read about athttp://www.python.org/peps/pep-0249.html. If you're not familiar with the Python DB-API, note that the SQL statement in cursor.execute() uses placeholders, "%s", rather than adding parameters directly within the SQL. If you use this technique, the underlying database library will automatically add quotes and escaping to your parameter(s) as necessary.

Rather than littering your view code with these django.db.connection statements, it's a good idea to put them in custom model methods or manager methods. For example, the above example could be integrated into a custom manager method like this:

```
from django.db import connection, models


class PersonManager(models.Manager):
    def first_names(self, last_name):
        cursor = connection.cursor()
        cursor.execute("""
            SELECT DISTINCT first_name
            FROM people_person
            WHERE last_name = %s""", [last_name])
        return [row[0] for row in cursor.fetchone()]


class Person(models.Model):
    first_name = models.CharField(max_length=50)
```

```
    last_name = models.CharField(max_length=50)

    objects = PersonManager()
```

And sample usage:

```
>>> Person.objects.first_names('Lennon')

['John', 'Cynthia']
```