The Django Web Framework – Part III

Hamid Zarrabi-Zadeh
Web Programming – Fall 2013
Outline

• Introduction
• Defining Models
• Relationships Between Models
• Model Inheritance
• Query Syntax
Introduction

• A **model** is the single, definitive source of information about your data
• It contains the essential **fields** and behaviors of the data you are storing
• Generally, each model maps to a single database **table**
The ORM

- Django provides a powerful ORM (Object-Relational Mapper)
- It bridges an underlying relational database with Python’s object-oriented nature
- In Django's ORM:
  - Python classes represent tables
  - objects represent individual rows within those tables
  - the table’s columns are attributes of those objects
Why ORM?

- Most modern Web frameworks use ORMs
- ORMs pros:
  - Integration with the programming language
  - Portability: support for various databases
  - Safety: avoids SQL injection attacks
  - Expressiveness: higher-level query syntax
- ORM cons:
  - Heavy reliance on ORM has been cited as a major factor in producing poorly designed databases
Defining Models
Defining Models

• Each model is a Python class that subclasses `django.db.models.Model`
• Each attribute of the model represents a database field
• Django automatically adds a primary key `id` to the model, which is an auto-incrementing integer
An Example

- This following model defines a **Person**:

```python
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

- This would create a database table like this:

```sql
CREATE TABLE myapp_person (  
    "id" serial NOT NULL PRIMARY KEY,  
    "first_name" varchar(30) NOT NULL,  
    "last_name" varchar(30) NOT NULL
);
```
The most important part of a model is the list of database fields it defines.

Each field in your model should be an instance of the appropriate Field class.

Django uses the field class types to determine:

- The database column type (e.g. INTEGER, VARCHAR)
- The default HTML widget to use when rendering a form field (e.g. `<input type="text">`, `<select>`)  
- The minimal validation requirements, used in Django’s admin and in automatically-generated forms
Field Types

- Django ships with a wide range of field types
- Some common field types include:
  - CharField and TextField
  - EmailField, URLField, and IPAddressField
  - IntegerField and BigIntegerField
  - DateField and DateTimeField
  - BooleanField and NullBooleanField
- The complete list is available in the model field reference
Field Options

- The following arguments are available to all field types, and are all optional
  - `blank`: If True, the field is allowed to be blank
  - `null`: If True, Django stores empty values as NULL in db
  - `choices`: A list of choices available for the field
  - `primary_key`: If True, the field is primary key
  - `unique`: If True, the field must be unique throughout table
  - `default`: The default value for the field
  - `help_text`: Help text to be displayed with the form widget
  - `verbose_name`: A human-readable name for the field
Relationships Between Models
Relationships

• Django offers ways to define the three types of database relationships
  – many-to-one
  – many-to-many
  – one-to-one
Many-to-One

- To define a many-to-one relationship, we simply add a `ForeignKey` to our model.
- `ForeignKey` requires a positional argument: the class to which the model is related.

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

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

- Foreign keys are technically a many-to-one relationship, as multiple child objects can refer to the same parent object
- The child gets a single reference to its parent, but the parent gets access to a set of its children

```python
# Pull a book off the shelf
book = Book.objects.get(title='Shefa')
# Get the book’s author
author = book.author
# Get a set of the books the author has written
books = author.book_set.all()
```
Reverse Relationship

• In the previous example, the “reverse relationship” from Author to Book is represented by the `Author.book_set` attribute, which is automatically added by the ORM.

• It is possible to override this naming scheme by specifying a `related_name` argument to the ForeignKey.

• For example, we could have defined `author` as `ForeignKey(Author, related_name='books')` and would then have access to `author.books` instead of `author.book_set`.
Many-to-Many

- What if a book has more than one author?
- Such a scenario requires a many-to-many relationship

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

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

# sample usage
authors = book.author.all()
books = authors[1].book_set.all()
```
We can define an intermediate table explicitly, and point to it using `through` keyword

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

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

class Authoring(models.Model):
    coll_type = models.CharField(max_length=100)
    book = models.ForeignKey(Book)
    author = models.ForeignKey(Author)
```
One-to-One

• Django implements this concept as a OneToOneField, almost identical to ForeignKey
• This is most useful when an object “extends” another object in some way

```python
class BookInfo(models.Model):
    isbn = models.CharField(max_length=30)

class Book(models.Model):
    title = models.CharField(max_length=100)
    info = models.OneToOneField(BookInfo)

# sample usage
info = book.info
print(info.book.title)
```
Meta Options

- You can add metadata to your model using the `Meta` inner class

```python
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    class Meta:
        db_table = 'contact_list'
        ordering = ['age']
        verbose_name_plural = 'people'
```
Model Inheritance
Model Inheritance

- Model inheritance in Django is almost identical to the normal class inheritance in Python.
- The only decision you have to make is whether
  - the parent models are models in their own right (with their own database tables), or
  - the parents are just holders of common information that will only be visible through the child models.
Model Inheritance

- There are three styles of inheritance in Django
  - If you just want to use the parent class to hold information that you don’t want to have to repeat for each child model, and the class isn’t going to ever be used in isolation, **abstract base classes** are what you’re after
  - If you’re subclassing an existing model and want each model to have its own database table, **multi-table inheritance** is the way to go
  - If you only want to modify the Python-level behavior of a model, without changing the models fields in any way, you can use **proxy models**
Abstract Base Classes

• To define an abstract base class, just put `abstract=True` in the Meta class
• This model will create no database table

```python
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(Person):
    student_id = models.CharField(max_length=10)
```
Multi-Table Inheritance

- Here, each model has its own database table
- The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created OneToOneField)

```python
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField()
    serves_pizza = models.BooleanField()
```
Proxy Models

- You can define a proxy model by setting the `proxy` attribute of the `Meta` class to `True`.
- Proxy class operates on the same database table as its parent class.

```python
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        pass
```
Query Syntax
Making Queries

- To retrieve objects from our database, we construct a **QuerySet** via a **Manager** on our model class
- A QuerySet represents a collection of objects from database, with zero, one or many filters
- **Filters** narrow down the query results based on the given parameters
- In SQL terms, a QuerySet equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT
Managers

- Each model has at least one Manager, called `objects` by default
- Managers are accessible only via model classes, not model instances
- We use our model's Managers to get QuerySets

```python
authors = Author.objects.all()
book = Book.objects.get(title='Shefa')
```
Manager Methods

- Managers have methods for typical queries
  - `all()`: returns a QuerySet containing all the database records for the model in question
  - `filter()`: returns a QuerySet containing the model records matching specific criteria
  - `exclude()`: the inverse of filter—finds records that don’t match the criteria
  - `get()`: returns a single record matching the given criteria (or raises an error if there are either no matches or more than one)
Field Lookups

- Field lookups are how you specify the meat of an SQL WHERE clause.
- They’re specified as keyword arguments to the Manager methods.
- Basic lookup keyword arguments take the form `field__lookuptype=value`.

```python
>>> Author.objects.filter(name__startswith='A')
>>> Person.objects.filter(age__lte=13)
>>> Person.objects.filter(name__contains='Ali')
>>> Person.objects.filter(name__iexact='ali')
>>> Book.objects.filter(pk__in=[1,4,7])
```
Django offers a powerful and intuitive way to follow relationships in lookups, taking care of the SQL JOINs automatically, behind the scenes.

To span a relationship, just use the field name of related fields across models, separated by __, until you get to the field you want.

```python
Book.objects.filter(author__name='Avicenna')
Book.objects.filter(author__name__contains='Av')
Author.objects.filter(book__title='Shefa')
```
QuerySet Modifiers

- QuerySet methods include:
  - order_by()
  - reverse()
  - distinct()
  - value() and values_list()

```python
>>> Author.objects.all().order_by('-name')
>>> Book.objects.all().order_by('author__name')
>>> Book.objects.values('title')
[{'title': 'Shefa'}, {'title': 'Ghanoon'}]
>>> Book.objects.values_list('title', 'author__name')
[('Shefa', 'Avicenna'), ('Ghanoon', 'Avicenna')]
```
Query Composition

- If you need to execute more complex queries, you can use **Q objects**
- A Q object is an object used to encapsulate a collection of keyword arguments
- Q objects can be combined using the & and | operators, and negated by ~ operator

```python
from django.db.models import Q

Person.objects.filter(Q(name='Ali') |
  (Q(age__gt=14) & ~Q(middle__startswith='W')))  
```
Custom SQL Queries

- If the ORM doesn’t meet your query-related needs, it is always possible to execute fully custom SQL by using a lower-level database adapter.

```python
from django.db import connection

cursor = connection.cursor()
cursor.execute('SELECT first, last 
    FROM myapp_person WHERE first="Ali"')
rows = cursor.fetchall()
for row in rows:
    print('{} {}'.format(row[0], row[1]))
```
References

- Django Documentation
- Python Web Development with Django
  - By Jeff Forcier, Paul Bissex, Wesley Chun