Django Models

COMP 8347

Slides prepared by Dr. Arunita Jaekel

arunita@uwindsor.ca

Django Models

- Topics
  - Creating simple models
    - Rich field types
    - Model inheritance
    - Meta inner class
  - Relationships between models
    - ForeignKey
    - ManyToManyField
  - Advanced usage
  - Getting a model's data
    - Querysets

Review MTV Architecture

- Represent data organization; defines a table in a database.
- Contain information to be sent to client; help generate final HTML.
- Actions performed by server to generate data.

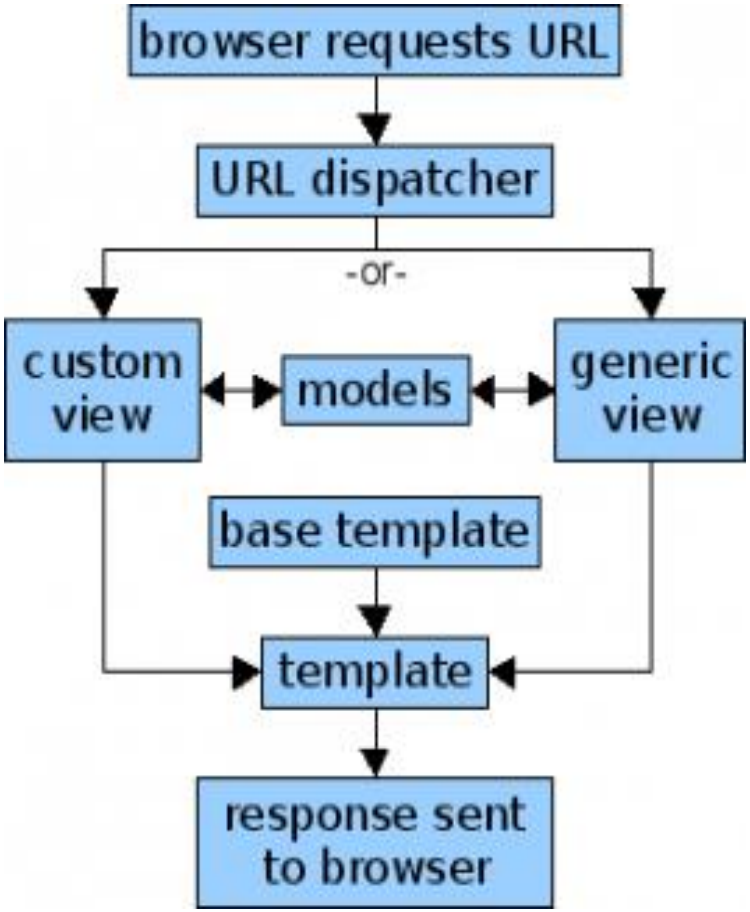


Fig. from [1]

Review MTV Architecture

- Represent data organization; defines a table in a database.
- Contain information to be sent to client; help generate final HTML.
- Actions performed by server to generate data.

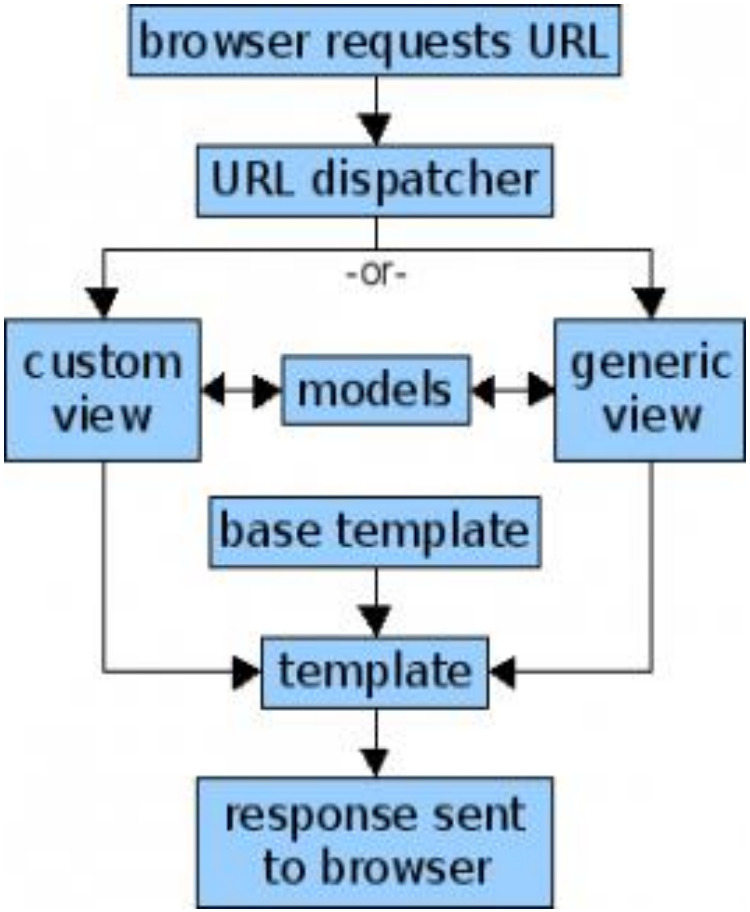


Fig. from [1]

Why Use ORM?

- Django provides rich db access layer
  - bridges underlying relational db with Python's object oriented nature
    - ***Portability***: support multiple database backends
    - ***Safety***: less prone to security issues (e.g. SQL injection attacks) arising from malformed or poorly protected query strings.
    - ***Expressiveness***: higher-level query syntax makes it easier to construct complex queries, e.g. by looping over structures.
    - ***Encapsulation***: Easy integration with programming language; ability to define arbitrary instance methods

Defining Models

- ***Model*** is an object that inherits from Model class.
  - Model → represented by a table in the db
  - Field → represented by a table column
- Models are defined and stored in the APP's ***models.py*** file.
- *models.py* is automatically created when you start the APP
  - Contains one line from django.db import models
  - This allows you to import the base model from Django.

Defining Models

- ***Model*** is an object that inherits from Model class.
  - Model → represented by a table in the db
  - Field → represented by a table column
- Models are defined and stored in the APP's ***models.py*** file.
- *models.py* is automatically created when you start the APP
  - Contains one line from django.db import models
  - This allows you to import the base model from Django.

untitled

File Edit Format Run Options Window Help

```python
from django.db import models

# Create your models here.
class Book(models.Model):
    title = models.CharField(max_length=200)
    length = models.IntegerField()
    website = models.URLField()
    city = models.CharField(max_length=20, blank=True)
    country = models.CharField(max_length=20, default='USA')

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

Ln: 6 Col: 34

Field Types

- Django provides a wide range of built-in field types. Some commonly used field types are given below:
  - *CharField*: character string with a limited number of characters.
    - title = models.CharField(**max_length=**200)
  - *TextField*: character string with unlimited number of characters.
  - *IntegerField*: Integer value.
  - *DateTimeField*: contains date as well as time in hours, minutes, and seconds.

null

If **True**, Django will store empty values as **NULL** in the database. Default is **False**.

blank

If **True**, the field is allowed to be blank. Default is **False**.

Field Types

- Django field types (continued):
  - *DateField*: contains the date only.
  - *BooleanField*: stores True or False values.
  - *NullBooleanField*: Similar to above, but allows empty or null value to specify you don't know yet.
  - *FileField*: stores a file path in the db; provides capability to upload a file and store on server.
  - *EmailField*, *URLField*, *IPAddressField*: stored in db like CharField;
    - has extra validation code to ensure value corresponds to valid email, URL, or IP address.

Primary Keys

- Primary key: A field guaranteed to be unique across the entire table.
  - In ORM terms: unique across the entire model.
  - Using auto-incrementing integers for this field is an effective way of ensuring uniqueness.
  - Useful as reference points for relationships between models.
- By default Django automatically creates this field (of type AutoField)

Primary Keys

- By default Django automatically creates a primary key field.
  - All models without an explicit primary key field are given an **id** attribute (of type AutoField).
    - id = models.AutoField(**primary_key=**True)
  - Autofield: behaves like normal integers; incremented for each new row in table.
  - To define your own primary key:
    - specify **primary_key = True** for one of your model fields.
    - this field becomes the primary key for the table.
    - it is now your responsibility to ensure this field is unique.

Example

- Book Model:

```
class Book(models.Model):
  title = models.CharField(max_length=100)
  length = models.IntegerField()
  pub_date = models.DateField()
```

Example

- Employee Model:

```
class Employee(models.Model):
    emp_no = models.IntegerField(default=999)
    name = models.CharField(max_length=50)
    age = models.IntegerField()
    email = models.EmailField(max_length=100)
    start_date = models.DateField()
```

Migrations

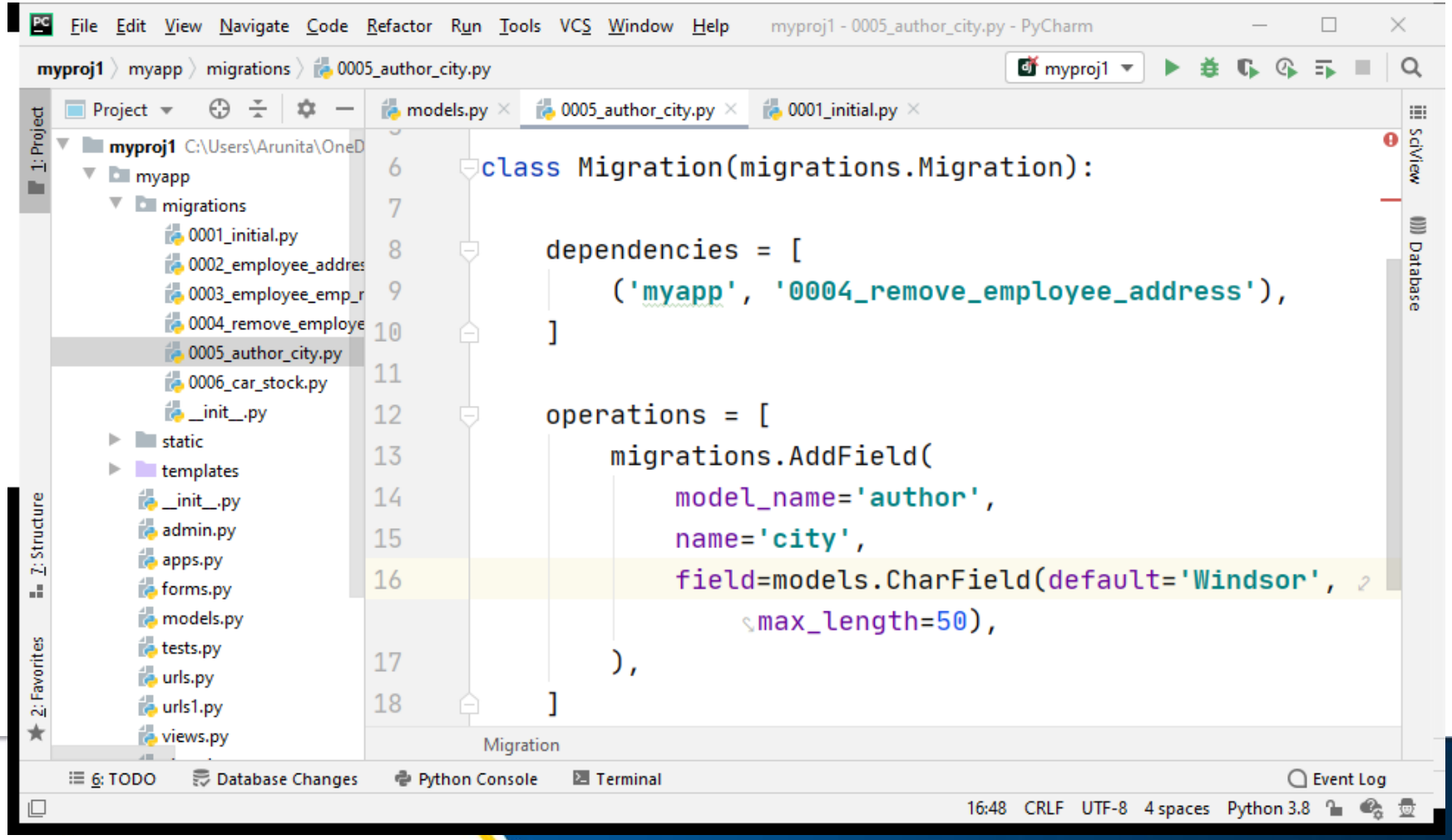
- **Migrations**: propagate changes to your models (adding a field, deleting a model, etc.) into your database schema.
  - Prior to version 1.7, Django only supported adding new models to the database; could not alter or remove existing models.
  - Used the syncdb command (the predecessor to migrate)

Migration Commands

- ***makemigrations*** : responsible for creating new migrations based on the changes made to your models.
- ***sqlmigrate***: displays the SQL statements for a migration.
- ***migrate***: responsible for applying migrations, as well as unapplying and listing their status.

Example of Migrations

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    city = models.CharField(max_length=50, default='Windsor')
```



Relationships Between Models

- Relationships are elements that join our models.
- Types of relationships:
  - **many-to-one**: multiple ‘child’ objects can refer to the same ‘parent’ object; the child gets a single reference to its parent.
  - **many-to-many**: requires a ‘many’ relationship on both sides.
  - **one-to-one**: both sides of the relationship have only a single-related object.

Many-To-One Relationship

- Uses the ForeignKey field
  - Requires a positional argument:
  - the class to which the model is related.
- Explicitly defined on only one side of the relationship.
  - The receiving end is able to follow the relationship backward.
- To create recursive relationship –object having many-to-one relationship with itself – use **models.ForeignKey('self').**

Example

```
class Company(models.Model):
    co_name = models.CharField(max_length=50)

class Car(models.Model):
    type = models.Charfield(max_length=20)
    company = models.ForeignKey(Company,
                on_delete=models.CASCADE)
```

NOTE: The class being referred to must be already defined; otherwise, the variable name would not be available for the Car class.

Example

Alternatively, use a string

– class name if it is defined in same file, or

– dot notation (e.g. ‘myApp.Company’) if defined in another file

```
class Car(models.Model):
    type = models.Charfield(max_length=20)
    company = models.ForeignKey(‘Company’,
     on_delete=models.CASCADE)

class Company(models.Model):
    co_name = models.CharField(max_length=50)
```

Many-to-Many Relationship

- Uses the ManytoManyField.
- Syntax is similar to ForeignKey field.
- Needs to be defined on one side of the relationship only.
  - Django automatically grants necessary methods and attributes to other side.
  - Relationship is symmetrical by default → doesn't matter which side it is defined on.

Example

```
class Book(models.Model):
  title = models.CharField(max_length=100)
  length = models.IntegerField()
  pub_date = models.DateField()

class Author(models.Model):
  name = models.CharField(max_length=50)
  books = models.ManyToManyField(Book)
```

NOTE: The many-to-many relation is only defined in one model.

Constraining Relationships

- Both *ForeignKey* and *ManyToManyField* take a limit_choices_to argument.
  - takes a dictionary as its value
  - dictionary key-value pairs are query keywords and values
  - powerful tool for defining the possible values of the relationship being defined.

```
staff_member = models.ForeignKey(User, on_delete=models.CASCADE,
                                 limit_choices_to={'is_staff': True})
```

One-To-One Relationship

- Uses the one-to-one field
  - Requires a positional argument:
  - the class to which the model is related.
  - Useful when an object “extends” another object in some way
- Explicitly defined on only one side of the relationship.
  - The receiving end is able to follow the relationship backward.
- Model inheritance involves an implicit one-to-one relation.

Model Inheritance

- Models can inherit from one another, similar to regular Python classes.
- Previously defined Empolyee class

```
class Employee(models.Model):
  name = models.CharField(max_length=50)
  age = models.IntegerField()
  email = models.EmailField(max_length=100)
  start_date = models. DateField()
```

- Suppose there are 2 types of employees
  – programmers and supervisors

Model Inheritance

- Option 1: Create 2 different models
  - duplicate all common fields
  - violates DRY principle.
- Option 2: Inherit from **Employee** class

```
class Supervisor(Employee):
    dept = models.CharField(max_length=50)

class Programmer(Employee):
    boss = models. ForeignKey(Supervisor,
                on_delete=models.CASCADE)
```

Adding Methods to Models

- Since a model is represented as a class, it can have *attributes* and *methods*.
- One useful method is the __str__ method
  - It controls how the object will be displayed.

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    length = models.IntegerField()
    pub_date = models.DateField()

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

Meta Inner Class

- ***Meta class*:** Used to inform Django of various metadata about the model.
  - E.g. display options, ordering, multi-field uniqueness etc.

```
class Employee(models.Model):
    name = models.CharField(max_length=50)
    age = models.IntegerField()
    email = models.EmailField(max_length=100)
    start_date = models.DateField()

    class Meta:
        ordering = ['name', 'start_date']
        unique_together = ['name', 'age']
```

Query Syntax

- Querying makes use of two similar classes: Manager and QuerySet
- ***Manager*:** Interface through which database query operations are provided to Django models
  - At least one Manager exists for every model
  - By default, Django adds a Manager with the name objects to every Django model class.

Manager Class

- Manager class has the following methods:
  - ***all***: returns a QuerySet containing all db records for the specified model
  - ***filter***: returns a QuerySet containing model records matching specific criteria
  - ***exclude***: inverse of filter; return records that don't match the criteria
  - ***get***: return a single record (model instance) matching criteria
    - raises error if no match or multiple matches.

Query Examples

```
class Company(models.Model):
    co_name = models.CharField(max_length=50)

class Car(models.Model):
    type = models.Charfield(max_length=20)
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
```

- Get all cars in the db.

  car_list = Car.objects.all()
- Get the car of type ‘Lexus’.

  car1 = Car.objects.get(type=‘Lexus’)
- Get the name of the company that made car1.

  name = car1.company.co_name
- Get all the cars made by ‘Ford’

  company = Company.objects.get(co_name=‘Ford’)
  cars = company.car_set.all()

Your Turn…

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    length = models.IntegerField()
    pub_date = models.DateField()

class Author(models.Model):
   name = models.CharField(max_length=50)
   books = models.ManyToManyField(Book)
```

- Get all the books written by ‘John Smith’
- Get all authors of the 2nd book in the list.
- Print the name of the first author.

QuerySet

- **QuerySet**: Can be thought of as a list of model class instances (records/rows)
  - above is a simplification – actually much more powerful
  - QuerySet as nascent db query:
    - List of all books:

      all_books = Book.objects.all()
    - List of books with the word “Python” in title:

      python_books = Book.objects.filter(title__contains=“Python”)
    - The book with id == 1:

      book = Book.objects.get(id=1)

QuerySet

- **QuerySet** as container: QuerySet implements a partial list interface and can be iterated over, indexed, sliced, and measured.
- Example 1:

```
python_books = Book.objects.filter(title__contains="Python')
for book in python_books:
    print(book.title)
```

- Example 2:

all_books = Book.objects.all()

How many books in db?

num_books = len(all_books) # should use *count* attribute instead

Get the first book:

first_book = all_books[0]

Get a list of first five books:

first_five = all_books[:5]

QuerySet

– QuerySet as building blocks: QuerySets can be *composed* into complex or nested queries.

– Example 1:

```
python_books = Book.objects.filter(title__contains="Python')
short_python_books = python_books.filter(length__lt=100)
```

–Equivalently:

```
short_python_books =
 Book.objects.filter(title__contains="Python').filter(length__lt=100)
```

Summary

- Creating simple models
- Relationships between models
- Setting up database
  - migrate
  - initial data using fixtures
- Retrieving data
  - Managers and QuerySets

References

- [1] https://docs.djangoproject.com/en/3.0/intro/tutorial02/
- [2] docs.djangoproject.com/en/3.0/topics/db/models/